

10_segmentation

May 8, 2022

1 Segmentacja obrazów

1.1 Cel ćwiczenia

- zapoznanie z metodami segmentacji obrazów:
 - segmentacją przez rozrost,
 - segmentacją przez podział i scalanie - zadanie domowe.

1.2 Wstęp

W ramach dotychczas wykonanych ćwiczeń poznaliśmy segmentację z wykorzystaniem binaryzacji (progowania) - tj. na podstawie jasności (koloru) poszczególnych pikseli. Wykonaliśmy dwa warianty metody: globalny i lokalny oraz przetestowaliśmy różne podejścia do automatycznego wyznaczania progu binaryzacji (iteracyjne oraz Otsu).

W tym ćwiczeniu poznamy dwie inne metody podziału obrazu na fragmenty: - segmentację przez rozrost obszaru (ang. *region growing*), - segmentację przez podział i scalanie (ang. *split and merge*).

1.3 Podstawy

Niech R oznacza obszar równy całemu analizowanemu obrazowi. Segmentację możemy uznać za proces podziału R na n podobszarów R_1, R_2, \dots, R_n takich że: 1. $\cup_{i=1}^n R_i = R$ - R_i - składa się z połączonych ze sobą pikseli, - $R_i \cap R_j = \emptyset$ dla wszystkich $i \neq j$, - $Q(R_i) = TRUE$ dla $i = 1, 2, \dots, n$ - $Q(R_i \cap R_j) = FALSE$ dla każdych sąsiednich R_i i R_j .

gdzie: symbole \cup i \cap oznaczają odpowiednio sumę i iloczyn zbiorów, a Q jest pewnym predykatem.

Punkt 1 oznacza, że segmentacja musi być kompletna tj. każdy piksel powinien zostać przyporządkowany do jakiegoś zbioru.

Punkt 2 oznacza, że piksele w ramach jednego podobszaru muszą być ze sobą połączone (na zasadzie sąsiedztwa 4 lub 8 punktowego).

Punkt 3 oznacza, że dowolne różne podobszary muszą być rozłączne.

Punkt 4 oznacza, że wszystkie piksele będące w ramach jednego podobszaru muszą spełniać pewną własność. Przykładowo może to być ten sam lub podobny odcień szarości.

Punkt 5 oznacza, że dwa sąsiednie podobszary muszą być różne w sensie predykatu Q (inaczej powinny zostać uznane za ten sam podobszar).

1.4 Segmentacja przez rozrost obszaru

Pomysł jest następujący. Wybieramy (jak ? - o tym później) piksele startowe (ang. *seed*) i od nich zaczynamy segmentację. Odbyna się ona na zasadzie sprawdzania czy sąsiednie piksele (sąsiedztwo 4 lub 8 punktowe) są podobne do centralnego pod względem jakiejś cechy (predykatu Q). Jeśli tak to oznaczane są jako należące do tej samej klasy co piksel centralny. Ponadto stają się one kolejnymi punktami startowymi metody. Zatem procedura ma charakter rekurencyjny.

Wybór punktów startowych może być podyktowany charakterem problemu (przykładowo wiemy gdzie na pewno zaczynają się obiekty). W ogólnym przypadku trzeba założyć, że pikselem startowym może być każdy piksel, co oczywiście wpływa na złożoność metody.

Kolejnym problemem jest wybór kryterium stopu tj. kiedy nasza procedura rekurencyjna ma się zakończyć. Dla danego podobszaru będzie to moment, kiedy nie istnieją już piksele, które można do niego dołączyć.

Warto w tym miejscu zwrócić uwagę, że stosowanie “sztywnego” warunku - np. różnica jasności pomiędzy pikselem centralnym, a analizowanym jest mniejsza niż 5 - może często dać niepożądane wyniki, gdyż nie uwzględnia pewnych lokalnych właściwości. Przykładowo, może się okazać, że jeśli na obrazie występuje niewielki gradient to za należące do tego samego obszaru uznane zostaną piksele o zupełnie różnych jasnościach. Możliwa jest też sytuacja odwrotna. Duże zróżnicowanie wartości na obrazie spowoduje zbyt duże “poszarpanie” wykrytych obszarów.

Jednym z możliwych rozwiązań jest uzależnienie kryterium podobieństwa (predykatu Q) od własności obrazu np. średniej jasności w obrębie danego obszaru. Można również dodać inne kryteria np. kształt podobszaru itp.

Uwaga. Pojęcie segmentacja przez rozrost to pewna **koncepcja** podejścia do segmentacji, a nie konkretna metoda. Na etapie projektowania algorytmu należy skupić się na konstrukcji kryterium podobieństwa (tj. co i jak ma być ze sobą porównywane) oraz ewentualnym uzupełnianiu metody o dodatkowe kryteria.

1.5 Zadanie: zaprojektować system segmentacji wybranej struktury na obrazie MRI (np. stawu kolanowego).

Punkt startowy wyznaczany będzie “ręcznie” (poprzez kliknięcie na obrazie).

1. Wczytaj obraz *knee.png* (w skali szarości) - MRI stawu kolanowego. Wyświetl go. Załóżmy, że chcemy dokonać segmentacji górnej kości. Przyjeliśmy, że punkt startowy metody wyznaczany będzie w sposób ręczny. Do pobrania położenia kursora myszy na ekranie służy funkcja `plt.ginput`. Uwaga. Jeśli obrazki są “osadzone” w notatniku, funkcja nie działa - proszę zwrócić uwagę na kod `matplotlib.use('TkAgg')`, który powoduje, że obrazki wyświetlane są “samodzielnie”.

Uwaga 1. Pobrane współrzędne należy zaokrąglić (*floor* lub *round*).

Uwaga 2. Dla potrzeb testów dobrze jest wpisać punkt startowy na “sztywno”. Pozwoli to uniknąć dość irytującego klikania przy każdym uruchomieniu skryptu.

2. Metodę zaimplementujemy z wykorzystaniem stosu. Uwaga. Podany poniżej opis jest tylko jedną z możliwych realizacji (niekoniecznie najlepszą). Na początek stworzymy dwie macierze o rozmiarach takich jak analizowany obraz. W jednej będziemy zapisywać odwiedzone lokalizacje (*visited* - typ *boolean*), a w drugiej rezultaty segmentacji (*segmented*). Obie macierze

tworzymy wypełnione zerami (funkcja `np.zeros`). Tworzymy też stos - w Python to po prostu "pusta lista".

3. W pierwszym kroku metody na stos (`stack.append`) odkładamy współrzędne wybranego przez użytkownika piksela. Oznaczamy go również jako odwiedzony (macierz *visited*) i zaliczony do obiektu (macierz *segmented*).
4. Pozostałe działanie odbywać się będzie w pętli `while`, której warunkiem stopu jest obecność elementów na stosie (`len(stack)>0`). W iteracji należy pobrać współrzędne piksela ze stosu. Następnie sprawdzamy, czy dla tego piksela można określić kontekst o rozmiarze 3×3 tj. czy ma wszystkich sąsiadów. Uwaga. Przyjmujemy tutaj uproszczenie - nie segmentujemy brzegu obrazka (ramki o szerokości 1 piksela).
5. W kolejnym kroku rozpisujemy pętlę po otoczeniu 3×3 (x2 `for`). Wewnątrz obliczamy odległość pomiędzy pikselem centralnym, a każdym z kontekstu. Przyjmijmy, że będzie to moduł z różnicy jasności. Jeśli wartość modułu będzie mniejsza od zdefiniowanego progu (proszę przyjąć jako początkową wartość 4) oraz rozpatrywany piksel nie był wcześniej odwiedzany to oznaczamy go jako należący do obiektu oraz jego współrzędne "odkładamy" na stosie. Uwaga. Pierwsza część warunku logicznego to nasz predykat Q. Za piksele podobne uznajemy takie, których różnica w jasności jest mniejsza niż zadany próg. Niezależnie od wyniku testu oznaczamy piksel jako odwiedzony (żeby wielokrotnie nie analizować tych samych lokalizacji).
6. Poza pętlą `while` proszę wyświetlić rezultat segmentacji. Czy wyniki są poprawne? Proszę poeksperymentować z wartością progu.

```
[1]: import os
import cv2
import matplotlib.pyplot as plt
import numpy as np

plt.rcParams["figure.figsize"] = (16,9)

if not os.path.exists("knee.png") :
    !ws1 wget https://raw.githubusercontent.com/vision-agh/poc_sw/master/
    ↪12_Segmentation/knee.png --no-check-certificate

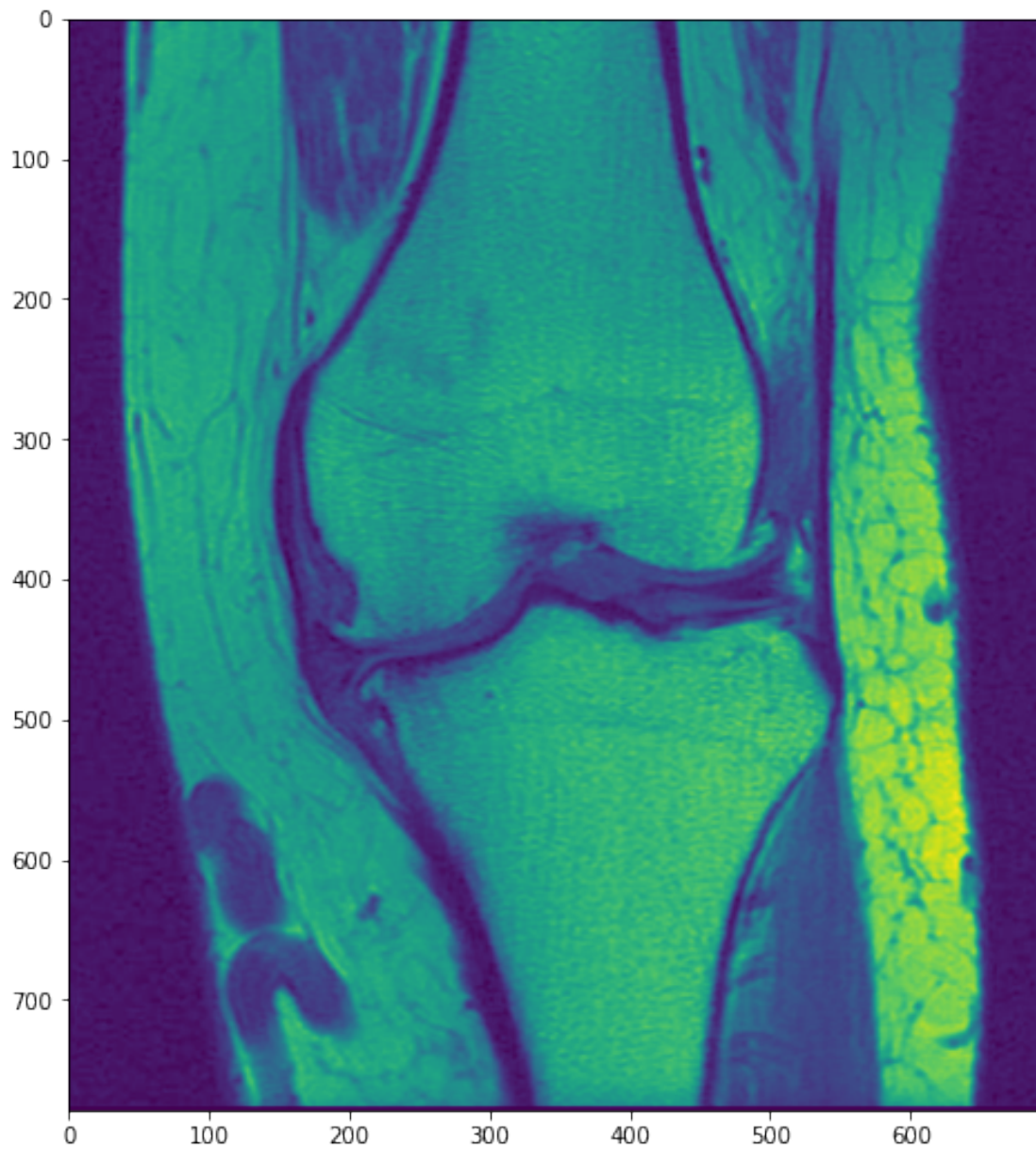
img = cv2.imread('knee.png',cv2.IMREAD_GRAYSCALE)

plt.imshow(img)
plt.show()

# jeżeli chcesz "ręcznie" wybrać punkt startowy:
# [XY] = plt.ginput(1,10)
# X = XY[0]
# Y = XY[1]
# print(X, Y)

# przykładowy punkt startowy
```

```
X = 290  
Y = 259
```



```
[2]: def imread(path):  
      return cv2.imread(path, cv2.IMREAD_GRAYSCALE)
```

```
[3]: def plot_img(img, cmap='gray', title=''):  
      plt.figure(figsize=(img.shape[0]/30, img.shape[1]/30))  
      plt.imshow(img, cmap=cmap, vmin=0, vmax=255)
```

```
plt.axis('off')  
plt.title(title)  
plt.show()
```

```
[4]: plot_img(img)
```



```
[5]: def segmentation_by_aggregating(image, start_x, start_y, threshold=4):  
      visited = np.zeros_like(image, dtype=np.uint8)  
      segmented = np.zeros_like(image, dtype=np.uint8)
```

```

stack = []

stack.append((start_x, start_y))
segmented[start_x, start_y] = 255
while stack:
    x, y = stack.pop()

    if visited[x, y] or (not 0 < x < image.shape[0]-1 or not 0 < y < image.
↪shape[1]-1):
        continue

    visited[x, y] = 1

    center = image[x, y]
    for dx in [-1, 0, 1]:
        for dy in [-1, 0, 1]:
            if dx == dy == 0:
                continue

            xn, yn = x + dx, y + dy

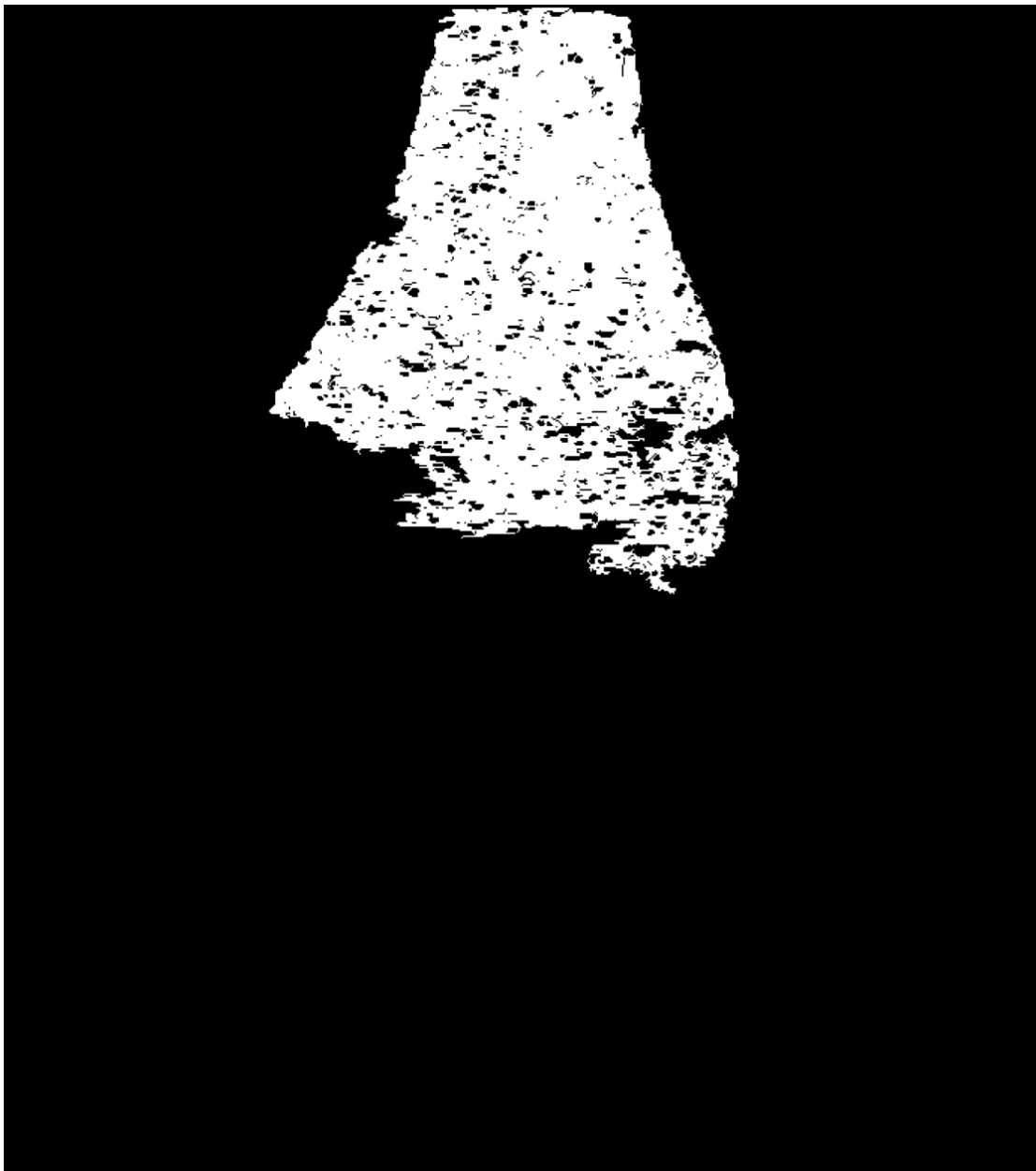
            if abs(int(center) - int(image[xn, yn])) <= threshold:
                segmented[xn, yn] = 255
                stack.append((xn, yn))

return segmented

```

```
[6]: segmented = segmentation_by_aggregating(img, X, Y, threshold=2)
```

```
[7]: plot_img(segmented)
```



1. Powyższy przykład ukazuje wspomniany wcześniej problem z "globalnym" podejściem do predykatu Q . Jeśli próg będzie mały, to wyznaczymy jedynie niewielki fragment kości. Natomiast zwiększenie progu skutkuje segmentacją nadmiarową. Mówiąc kolokwialnie, na obrazie znajdzie się "ścieżka" po której możliwe jest przejście od obszaru jasnego do ciemnego nie "łamiąc" progu odległości pomiędzy sąsiednimi pikselami.
2. Aby zaradzić powyższemu problemowi, można za kryterium podobieństwa przyjąć, nie różnicę jasności względem piksela centralnego, a od globalnie wyznaczonego i aktualizowanego progu. W najprostszym przypadku może to być średnia jasność w wyznaczonym obszarze. W celu implementacji mechanizmu wystarczy dodać dwie zmienne: średnią (mV) oraz licznik pikseli

uznanych za należące do obiektu (nS). Przy każdym zdjęciu ze stosu licznik jest zwiększany o 1. Aktualizacja średniej następuje na podstawie równania:

$$mV_n = \frac{mV_{nS-1}(nS - 1) + I}{nS} \quad (1)$$

Następnie wystarczy tylko zmienić sposób obliczania odległości - zamienić piksel centralny na wartość średnią. Proszę spróbować jak działa metoda z taką modyfikacją. Proszę się liczyć z koniecznością zwiększenia progu (nawet dość znaczną).

3. Poprawić działanie metody może również dodanie filtracji uśredniającej np. filtrem Gaussa.

```
[8]: def segmentation_by_aggregating2(image, start_x, start_y, threshold=4, mV=0, nS=0):
    visited = np.zeros_like(image, dtype=np.uint8)
    segmented = np.zeros_like(image, dtype=np.uint8)

    stack = []

    stack.append((start_x, start_y))
    segmented[start_x, start_y] = 255
    while stack:
        x, y = stack.pop()

        if visited[x, y] or (not 0 < x < image.shape[0]-1 or not 0 < y < image.
        shape[1]-1):
            continue

        visited[x, y] = 1
        center = image[x, y]

        nS += 1
        mV = (mV * (nS - 1) + center) / nS

        for dx in [-1, 0, 1]:
            for dy in [-1, 0, 1]:
                if dx == dy == 0:
                    continue

                xn, yn = x + dx, y + dy

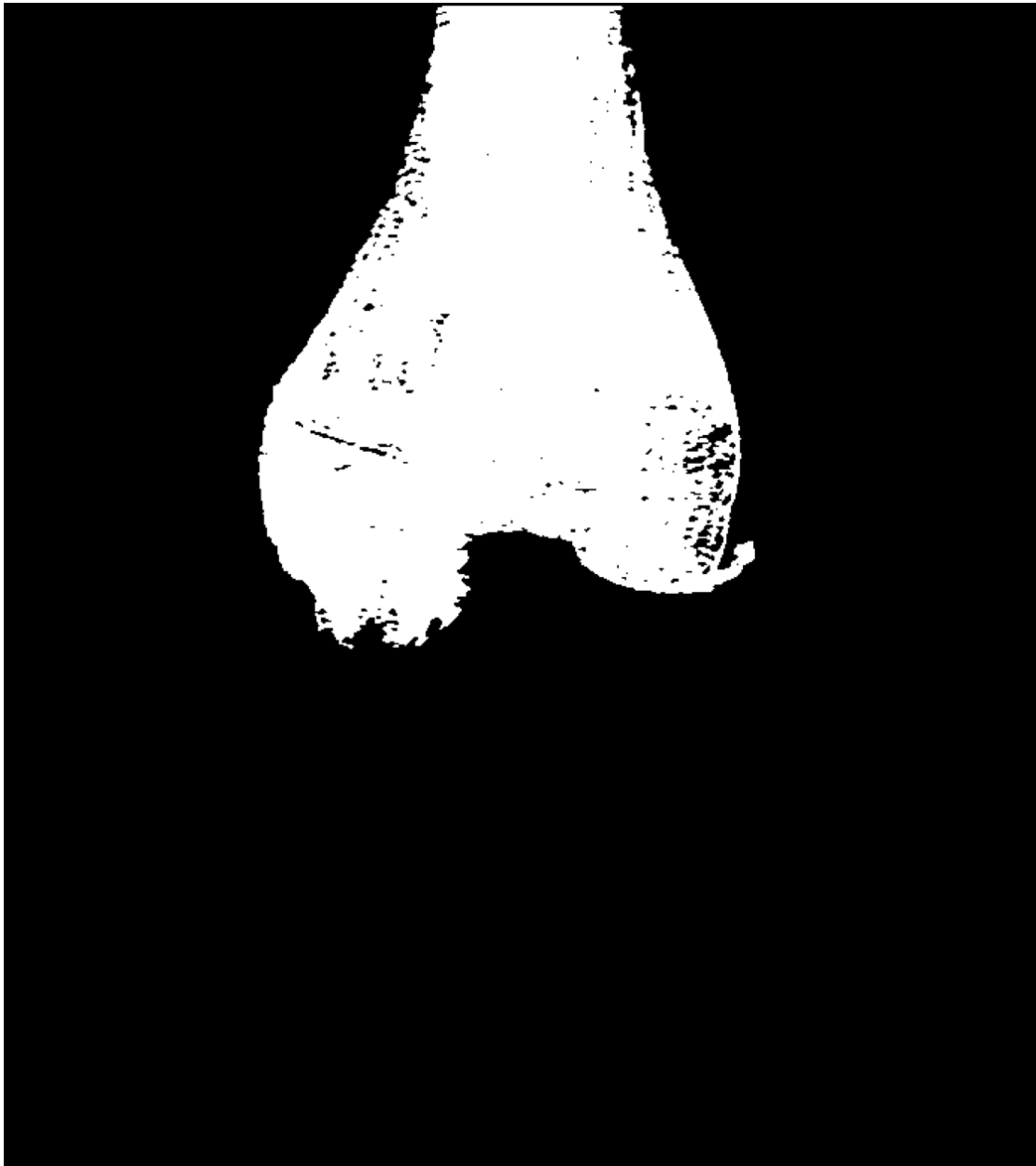
                if abs(mV - image[xn, yn]) <= threshold:
                    segmented[xn, yn] = 255
                    stack.append((xn, yn))

    return segmented
```

```
[9]: segmented = segmentation_by_aggregating2(img, X, Y, threshold=29)
```



```
[10]: plot_img(segmented)
```



```
[11]: segmented = segmentation_by_aggregating2(img, X, Y, threshold=50)
```

```
[12]: plot_img(segmented)
```



```
[13]: plot_img(img)
```



1.6 Zadanie domowe: segmentacja przez podział i łączenie

Omówioną procedurę segmentacji przez rozrost można określić jako podejście z dołu do góry, od szczegółu do ogółu (ang. *bottom-up*) - zaczynamy od pojedynczego piksela i dochodzimy do całego obrazu lub jego fragmentu. Segmentację można również zacząć od całego obrazka i dokonywać jego podziału na fragmenty, które ew. mogą zostać później połączone. Takie podejście można określić jako od góry do dołu, od ogółu do szczegółu (ang. *top-down*).

Niech R oznacza cały obraz, a Q wybrany predykat. Sprawdzamy, czy zachodzi zależność $Q(R) = TRUE$. Zwykle tak nie jest, gdyż to oznaczałoby obecność na obrazie tylko jednego obszaru o

“spójnych” właściwościach, czyli segmentacja w takim przypadku byłaby zbędna (w szczególności zachodziłoby to dla obrazu jednorodnego). Jeśli $Q(R) = FALSE$ dzielimy obszar R na podobszary. Zwykle stosuje się tutaj podział na cztery jednakowe kwadratowe podobszary. Zilustrowano to na poniższym rysunku.

Podział ten można również opisać w formie drzewa, gdzie każdy wierzchołek może mieć dokładnie 4 następniki. W takim ujęciu korzeń to cały obraz. Dla nowo powstałych podobszarów R_1, R_2, R_3, R_4 sprawdzany jest predykat Q . Jeśli nie jest on spełniony to następuje dalszy podział - w przykładzie na $R_{41}, R_{42}, R_{43}, R_{44}$. Procedurę kontynuujemy do momentu kiedy wszystkie podobszary będą spójne. Uwaga. Zwykle określa się minimalny rozmiar podobszaru (większy niż 1 piksel np. blok o rozmiarze 8×8).

W wyniku powyższej procedury otrzymujemy podział obrazka, przy czym często się może zdarzyć, że leżące obok siebie podobszary będą spełniać predykat tj. $Q(R_i \cup R_j) = TRUE$ dla sąsiadujących R_i i R_j . Jest to sprzeczne z podpunktem 5 w warunkach segmentacji. Zjawisko to eliminowane jest w procedurze łączenia. Dla każdego z podobszarów sprawdza się jego sąsiadów i jeśli spełniony jest warunek $Q(R_i \cup R_j) = TRUE$ to następuje ich połączenie. Procedura kontynuowana jest do momentu, kiedy niemożliwe jest dalsze łączenie.

Dla metody możliwe są również uproszczenia. Przykładowo osłabia się warunek na łączenie z $Q(R_i \cup R_j) = TRUE$ na $Q(R_i) = TRUE \wedge Q(R_j) = TRUE$, czyli warunki muszą być spełnione dla każdego podobszaru osobno, ale już nie dla połączenia.

Uwaga. Podobnie jak dla segmentacji przez rozrost, segmentacja przez podział i łączenie to też tylko pewna koncepcja (pewne ramy) algorytmu. Stworzenie konkretnego rozwiązania jest zadaniem dla osoby realizującej dany system wizyjny i wymaga analizy konkretnego problemu.

Uwaga. Podany poniżej sposób implementacji nie jest ani jedynym z możliwych, ani optymalnym obliczeniowo. Ma on za zadanie tylko zilustrować koncepcję tej metody segmentacji.

1.7 Realizacja ćwiczenia

1. Wczytaj obraz *umbrella.png*. Dokonaj konwersji z BGR do RGB. Wyświetl go. Załóżmy, że chcemy dokonać segmentacji poszczególnych fragmentów kolorowego parasola. Nasz algorytm opierać się będzie na podejściu podziału i łączenia. Jako kryterium podziału zastosujemy “jednorodność” danego obszaru, którą można opisać poprzez odchylenie standardowe. Przy scalaniu będziemy brać pod uwagę średni ocień koloru tj. łączyć podobszary o zbliżonym odcieniu.
2. W pierwszym etapie należy dokonać konwersji do przestrzeni barw HSV. Następnie wybrać tylko składową H (tj. barwę). Można ją kontrolnie wyświetlić.
3. Procedurę podziału wygodnie jest zrealizować w formie rekurencji. Utwórz funkcję np. `split`, która jako argumenty przyjmować będzie obraz oraz cztery współrzędne analizowanego podobszaru. W pierwszym kroku należy wyciąć odpowiedni fragment oraz obliczyć dla niego średnią i odchylenie standardowe (`np.mean` oraz `np.std`).
4. Następnie sprawdzamy czy odchylenie jest mniejsze niż zadany przez nas próg (np. 0.05) oraz czy nie osiągnęliśmy limitu podziału (np. bok kwadratu 8 pikseli). Oba potrzebne progi możemy zrealizować za pomocą zmiennych globalnych dostępnych w Python (po prostu zadeklarowanych przed funkcją). Potrzeby rozmiar kwadratu obliczamy na podstawie współrzędnych (przekazanych do funkcji jako argumenty)

Uwaga. Użycie zmiennych globalnych jest złą praktyką programistyczną i zachęcamy do “opakowania” obliczeń w klasę - nie jest to jednak obowiązkowe.

5. Jeśli podobszar nie jest jednorodny (tj. duże odchylenie standardowe) i nie osiągnęliśmy minimalnego rozmiaru podobszaru to dokonujemy podziału na cztery części. Wyznaczamy rozmiar aktualnego podobszaru (na podstawie jego współrzędnych). Z ich wykorzystaniem otrzymujemy cztery identyczne podobszary (I_1, I_2, I_3, I_4).

Następnie dla każdego z nich wywołujemy funkcję `split` - rekurencja. Najtrudniejsze jest odpowiednie podanie współrzędnych. Mają to być **rzeczywiste** współrzędne podobszaru we współrzędnych globalnych (obrazu w pełnej rozdzielczości). Podpowiedź. Trzeba “odpowiednio” wykorzystać współrzędne podobszaru przed podziałem (tj. argumenty funkcji) oraz rozmiar podobszaru. Należy zwrócić uwagę na to, aby nie wystąpił błąd przesunięcia o 1 tj. współrzędna była większa/mniejsza o 1 od rzeczywistej. W razie wątpliwości warto sobie te parametry wypisać na konsolę lub użyć debugera.

6. Jeśli podobszar jest jednorodny lub nie możemy już dalej prowadzić podziału to:
 - zapisujemy indeks danego podobszaru: `seg_res[y1:y2, x1:x2] = index`, gdzie: `seg_res` to globalna macierz o rozmiarze takim jak obraz wejściowy (typ `int16`), `y1,y2,x1,x2` to współrzędne podobszaru. W macierzy `seg_res` zapisywane będą obszary jednorodne. Zmienna `index` jest globalnym licznikiem podobszarów. Powinna zostać zainicjowana wartością 1 przed pierwszym wywołaniem funkcji `split`. Następnie, każdorazowo po przypisaniu należy ją inkrementować. Uwaga. Aby możliwa była modyfikacja zmiennej globalnej, należy wewnątrz funkcji dodać np. linijkę: `global seg_res`
 - zapisujemy średnią podobszaru. Wykorzystujemy macierz globalną `MRes` oraz kod podobny do opisanego powyżej. Wartości te wykorzystamy przy etapie scalania.

Uwaga. Podany sposób zapisu wyników podziału jest dość nieefektywny. Następuje powielenie tej samej informacji. Lepszym pomysłem byłoby wykorzystanie podejścia opartego o grafy, jednak jest ono trudniejsze do “szybkiej” implementacji.

7. Proszę wyświetlić macierze `seg_res` in `m_res`. Pierwsza powinna pokazywać podział na obszary - mniejsze lub większe kwadraty. Druga zasadniczo powinna wyglądać podobnie jak składowa H, tylko bardziej “rozmażany”.

```
[14]: import os
import cv2
import matplotlib.pyplot as plt
import numpy as np

if not os.path.exists("umbrella.png") :
    !ws1 wget https://raw.githubusercontent.com/vision-agh/poc_sw/master/
↪12_Segmentation/umbrella.png --no-check-certificate
```

```
[15]: umbrella = cv2.imread('umbrella.png')
umbrella = cv2.cvtColor(umbrella, cv2.COLOR_BGR2RGB)
```

```
[16]: plot_img(umbrella)
```



```
[17]: umbrella_hsv = cv2.cvtColor(umbrella, cv2.COLOR_RGB2HSV)  
h = umbrella_hsv[:, :, 0]
```

```
[18]: plot_img(h)
```



```
[19]: def segmentation_by_splitting(image, std_threshold=0.05, min_size=4):
    index = 0

    seg_res = np.zeros_like(image, dtype=np.int16)
    m_res = np.zeros_like(image, dtype=np.int16)

    def split(y1, x1, y2, x2):
        nonlocal index

        width = x2 - x1
        height = y2 - y1
        cut = image[y1:y2, x1:x2]
```

```

        if width < min_size*2 or height < min_size*2 or np.std(cut) <
↳std_threshold:
            seg_res[y1:y2, x1:x2] = index
            m_res[y1:y2, x1:x2] = np.mean(cut)
            index += 1
            return

    nw = (width + 1) // 2
    nh = (height + 1) // 2

    split(y1,      x1,      y1 + nh, x1 + nw)
    split(y1,      x1 + nw, y1 + nh, x2)
    split(y1 + nh, x1 + nw, y2, x2)
    split(y1 + nh, x1,      y2, x1 + nw)

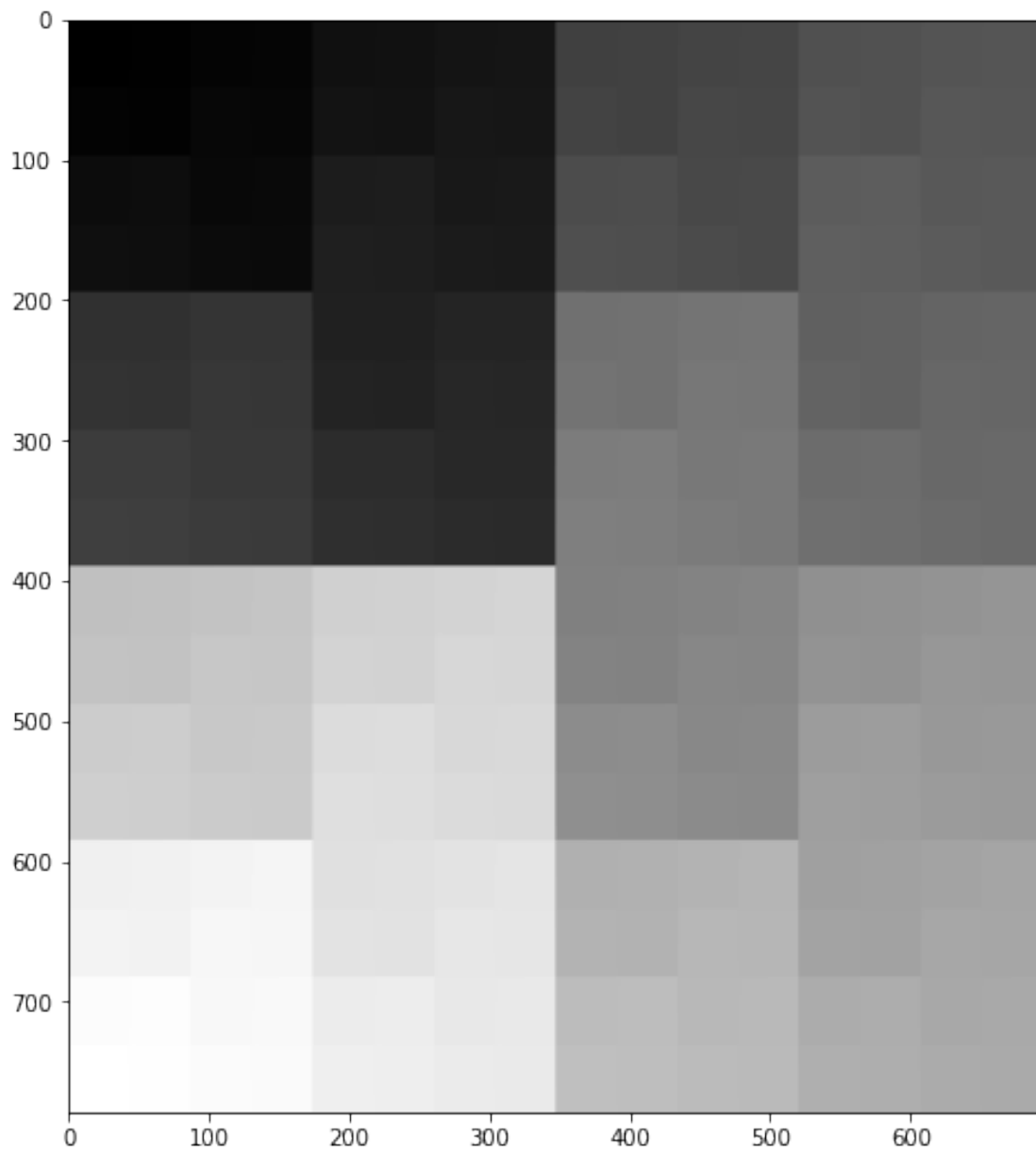
    split(0, 0, image.shape[0], image.shape[1])

    return seg_res, m_res

```

```
[20]: s, m = segmentation_by_splitting(img)
```

```
[21]: plt.imshow(s, cmap='gray')
plt.show()
```

```
[22]: plot_img(m)
```



1. Drugi etap to łączenie. Idea jest nieco zbliżona do segmentacji przez rozrost. Wybieramy dany podobszar i analizujemy sąsiednie podobszary. Jeśli są one podobne to dołączamy je do aktualnie rozpatrywanego. Za kryterium podobieństwa przyjmujemy niewielką różnicę w uśrednionym odcieniu barwy (składowa H).
2. Procedurę realizujemy wewnątrz pętli `while`. Warunkiem jej stopu jest przekroczenie przez licznik (np. `i`) wartości `index`, co oznacza, że przeanalizowane zostały wszystkie znalezione w pierwszym etapie podobszary.
3. W pierwszym kroku ‘wycinamy’ maskę pikseli o rozpatrywanym indeksie tj. `ib = seg_res`

`== i`. Dokonujemy też konwersji na typ `uint8`. Następnie sprawdzamy, czy maska zawiera elementy niezerowe tj. czy nie jest "pusta" - `np.any`. Taki przypadek może zajść jeśli pewien podobszar został dołączony do innego i zmienione zostały jego indeksy. Dla "pustej" maski inkrementujemy licznik `i` i przechodzimy do następnej iteracji - `continue`.

4. Dla rozpatrywanego podobszaru znajdujemy współrzędną lewego górnego rogu. Można to zrobić z wykorzystaniem składni: `(y, x) = np.nonzero(ib)` oraz `first = (y[0], x[0])` - znajdowanie współrzędnych pierwszego niezerowego elementu.
5. Następnie należy znaleźć sąsiadów rozpatrywanego obszaru. Można wykorzystać dylatację maski `ib` z elementem strukturalnym w postaci kwadratu o rozmiarze 3×3 - `dil = cv2.dilate(ib, cv2.getStructuringElement(cv2.MORPH_RECT, (3,3)))` Następnie od maski po dylatacji odejmujemy maskę oryginalną. Otrzymujemy "ramkę", którą wykorzystujemy do "wycięcia" (mnożenie punktowe) fragmentu z macierzy `seg_res`. Z tego fragmentu wybieramy elementy niezerowe (np. w ten sposób `mult_nz = mult[np.nonzero(mult)]`) i unikalne (tj. eliminacja duplikatów) - `mult_uniq = np.unique(mult_nz)`.
6. Mając wektor z indeksami sąsiadów realizujemy pętlę w której dla każdego sąsiada:
 - wycinamy opowiadającą mu maską `i` i znajdujemy jej lewy górny róg - w sposób analogiczny do opisanego powyżej,
 - sprawdzamy, czy moduł z różnicy pomiędzy średnimi odcieniami barwy w dwóch analizowanych podobszarach jest mniejszy od progu (np. $5/255$). Jeśli tak to łączymy obszary tj. sąsiadowi przypisujemy indeks rozpatrywanego `seg_res[ibs] = i`, gdzie `ibs` - maska podobszaru sąsiedniego. Do poprawnego działania konieczna jest jeszcze flaga informująca czy nastąpiło połączenie. Jej wartość ustawiamy na 0 przed pętlą po sąsiednich podobszarach. Jeśli wystąpiło połączenie to wartość ustawiamy na jeden.

Po pętli po sąsiadach sprawdzamy flagę. Jeśli ma ona wartość 0 tj. nie nastąpiło połączenie obszarów to licznik `i` jest inkrementowany.

7. Uwaga. Obliczenia mogą chwilę trwać. Proszę wyświetlić macierz `seg_res`. Jeśli implementacja jest poprawna, to wynik segmentacji powinien być względnie dobry - wszystkie elementy parasolki wyodrębnione.

```
[23]: def segmentation_by_splitting(image, mean_threshold=5, std_threshold=0.05,
    ↪min_size=4):
    index = 0

    seg_res = np.zeros_like(image, dtype=np.int16)
    m_res = np.zeros_like(image, dtype=np.int16)

    def split(y1, x1, y2, x2):
        nonlocal index

        width = x2 - x1
        height = y2 - y1
        cut = image[y1:y2, x1:x2]
```

```

        if width < min_size*2 or height < min_size*2 or np.std(cut) <
↪std_threshold:
            seg_res[y1:y2, x1:x2] = index
            m_res[y1:y2, x1:x2] = np.mean(cut)
            index += 1
            return

    nw = (width + 1) // 2
    nh = (height + 1) // 2

    split(y1,      x1,      y1 + nh, x1 + nw)
    split(y1,      x1 + nw, y1 + nh, x2)
    split(y1 + nh, x1 + nw, y2, x2)
    split(y1 + nh, x1,      y2, x1 + nw)

    split(0, 0, image.shape[0], image.shape[1])

def find(idx):
    mask = (seg_res == idx).astype(np.uint8)
    if not np.any(mask):
        return None, (None, None)

    y, x = np.nonzero(mask)
    return mask, (y[0], x[0])

i = 0
while i < index:
    mask, (y, x) = find(i)

    if mask is None:
        i += 1
        continue

    dil = cv2.dilate(mask, cv2.getStructuringElement(cv2.MORPH_RECT, (3,3)))
    frame = dil - mask
    mult = frame * seg_res
    mult_nz = mult[np.nonzero(mult)]
    mult_uniq = np.unique(mult_nz)

    mean = m_res[y, x]
    pixels = np.sum(mask)
#     print(i, pixels)

    joined = False
    for neighbor in mult_uniq:

```

```

neighbor_mask, (ny, nx) = find(neighbor)
neighbor_pixels = np.sum(neighbor_mask)
all_pixels = pixels + neighbor_pixels
neighbor_mean = m_res[ny, nx]

#         print('n', neighbor, neighbor_pixels, all_pixels)

if abs(mean - neighbor_mean) <= mean_threshold:
    joined = True

    seg_res[neighbor_mask.astype(np.bool)] = i
    new_mean = mean * (pixels / all_pixels) + neighbor_pixels * 1
    ↪(neighbor_pixels / all_pixels)

    mean = new_mean
    pixels = all_pixels

    m_res[mask] = new_mean
    m_res[neighbor_mask] = new_mean

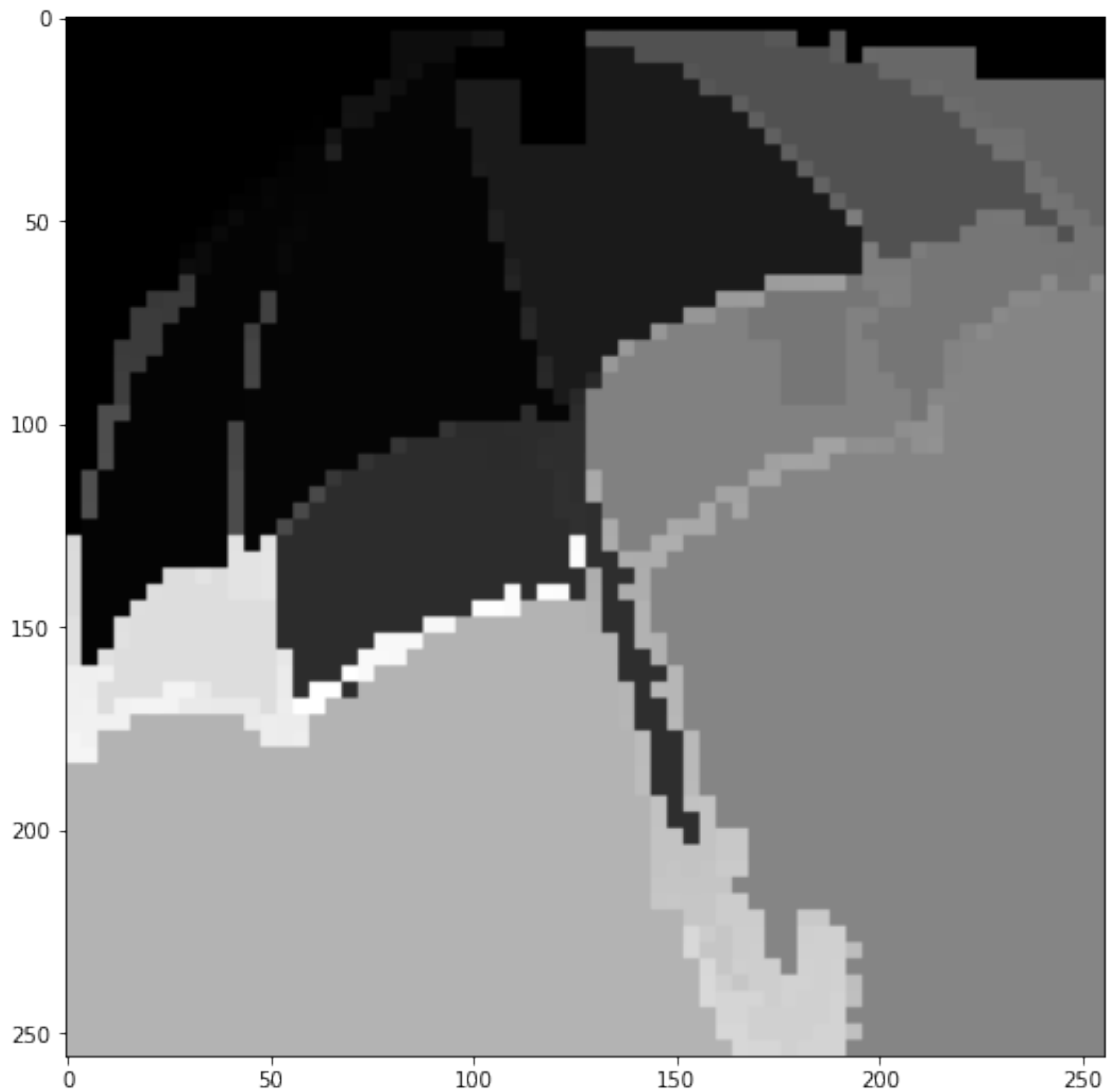
if not joined:
    i += 1

return seg_res

```

```
[24]: s = segmentation_by_splitting(h)
```

```
[25]: plt.imshow(s, cmap='gray')
plt.show()
```



8. Na koniec zastosujemy jeszcze dwie proste metody filtracji wyników. Po pierwsze wyeliminujemy obszary o rozmiarze mniejszym niż zadany (np. 100 pikseli). W tym celu wyznaczamy wektor unikalnych indeksów: `unique = np.unique(seg_res)`. Następnie implementujemy pętlę `for` po tych indeksach. Wewnątrz wycinamy maskę dla rozpatrywanego indeksu i obliczamy jej pole (`sum`). Jeśli jest ono mniejsze niż próg to cały podobszar “zerujemy” - `seg_res[mask] = 0`, gdzie `mask` - maska podobszaru.
9. W drugim kroku przeprowadzamy przeindeksowanie indeksów na pierwsze N liczb całkowitych. Ponownie wyznaczamy unikalne indeksy. Następnie w pętli `for` wycinamy maskę o indeksie `U(ii)`, a do wyniku przypisujemy iterator pętli `seg_res[mask] = ii`.
10. Proszę poeksperymentować z różnymi parametrami. Wyniki proszę zaprezentować prowadzącemu.

```

[26]: def segmentation_by_splitting(image, mean_threshold=5, std_threshold=0.05,
    ↪min_size=4, min_area=100):
    index = 0

    seg_res = np.zeros_like(image, dtype=np.int16)
    m_res = np.zeros_like(image, dtype=np.int16)

    def split(y1, x1, y2, x2):
        nonlocal index

        width = x2 - x1
        height = y2 - y1
        cut = image[y1:y2, x1:x2]

        if width < min_size*2 or height < min_size*2 or np.std(cut) <
    ↪std_threshold:
            seg_res[y1:y2, x1:x2] = index
            m_res[y1:y2, x1:x2] = np.mean(cut)
            index += 1
            return

        nw = (width + 1) // 2
        nh = (height + 1) // 2

        split(y1, x1, y1 + nh, x1 + nw)
        split(y1, x1 + nw, y1 + nh, x2)
        split(y1 + nh, x1 + nw, y2, x2)
        split(y1 + nh, x1, y2, x1 + nw)

    split(0, 0, image.shape[0], image.shape[1])

    def find(id):
        mask = (seg_res == id).astype(np.uint8)
        if not np.any(mask):
            return None, (None, None)

        y, x = np.nonzero(mask)
        return mask, (y[0], x[0])

    i = 0
    while i < index:
        mask, (y, x) = find(i)

        if mask is None:
            i += 1

```

```

        continue

    dil = cv2.dilate(mask, cv2.getStructuringElement(cv2.MORPH_RECT,(3,3)))
    frame = dil - mask
    mult = frame * seg_res
    mult_nz = mult[np.nonzero(mult)]
    mult_uniq = np.unique(mult_nz)

    mean = m_res[y, x]
    pixels = np.sum(mask)
#     print(i, pixels)

    joined = False
    for neighbor in mult_uniq:
        neighbor_mask, (ny, nx) = find(neighbor)
        neighbor_pixels = np.sum(neighbor_mask)
        all_pixels = pixels + neighbor_pixels
        neighbor_mean = m_res[ny, nx]

#         print('n', neighbor, neighbor_pixels, all_pixels)

        if abs(mean - neighbor_mean) <= mean_threshold:
            joined = True

            seg_res[neighbor_mask.astype(np.bool)] = i
            new_mean = mean * (pixels / all_pixels) + neighbor_pixels *
↪(neighbor_pixels / all_pixels)

            mean = new_mean
            pixels = all_pixels

            m_res[mask] = new_mean
            m_res[neighbor_mask] = new_mean

    if not joined:
        i += 1

objects = np.unique(seg_res)

for obj in objects:
    mask, (y, x) = find(obj)
    if mask is None:
        continue

    area = np.sum(mask)
    if area < min_area:
        seg_res[mask.astype(np.bool)] = 0

```



```

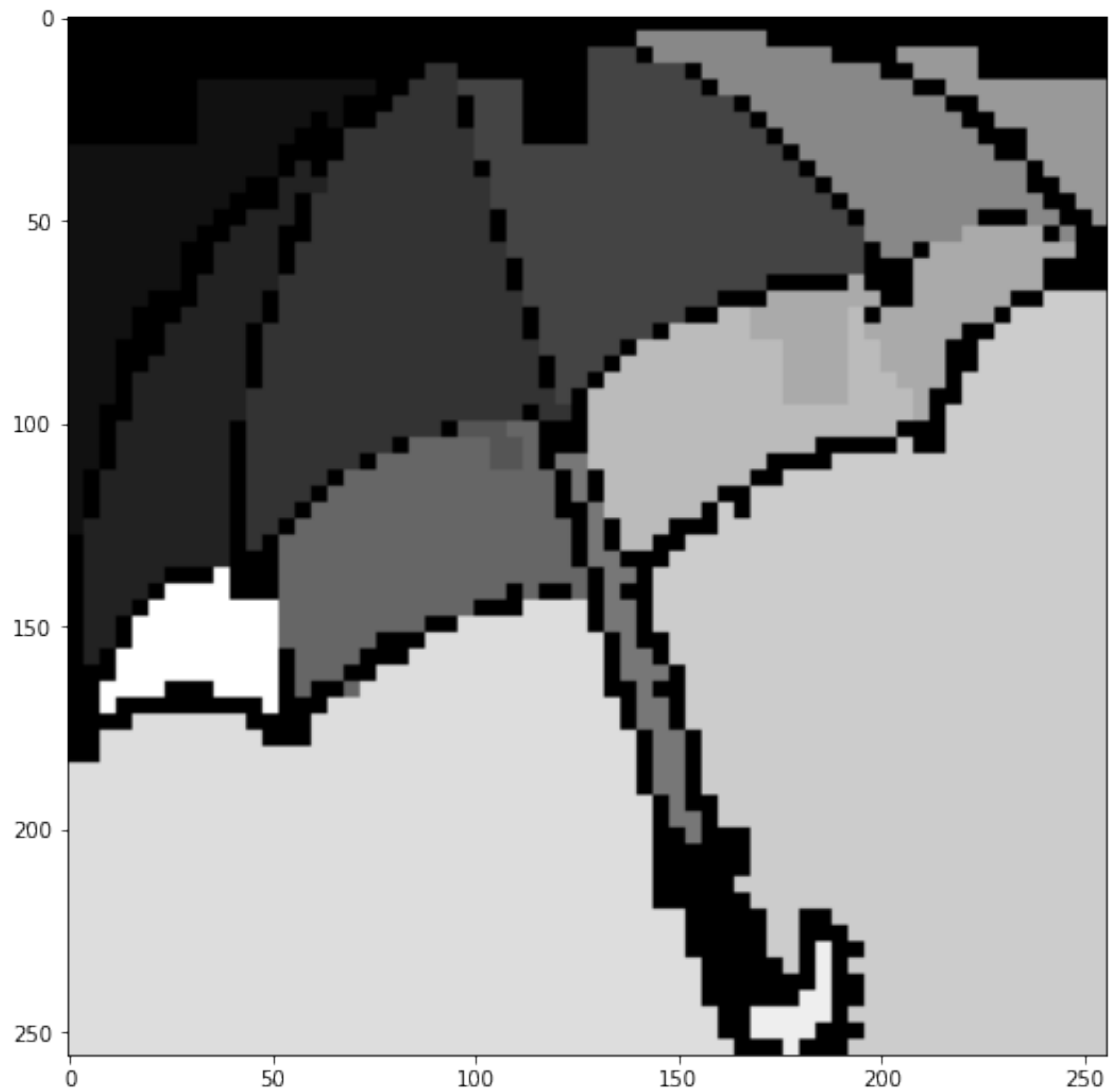
index = 1
for i in np.unique(seg_res):
    mask, _ = find(i)
    if i > 0:
        seg_res[mask.astype(np.bool)] = index
        index += 1

return seg_res

```

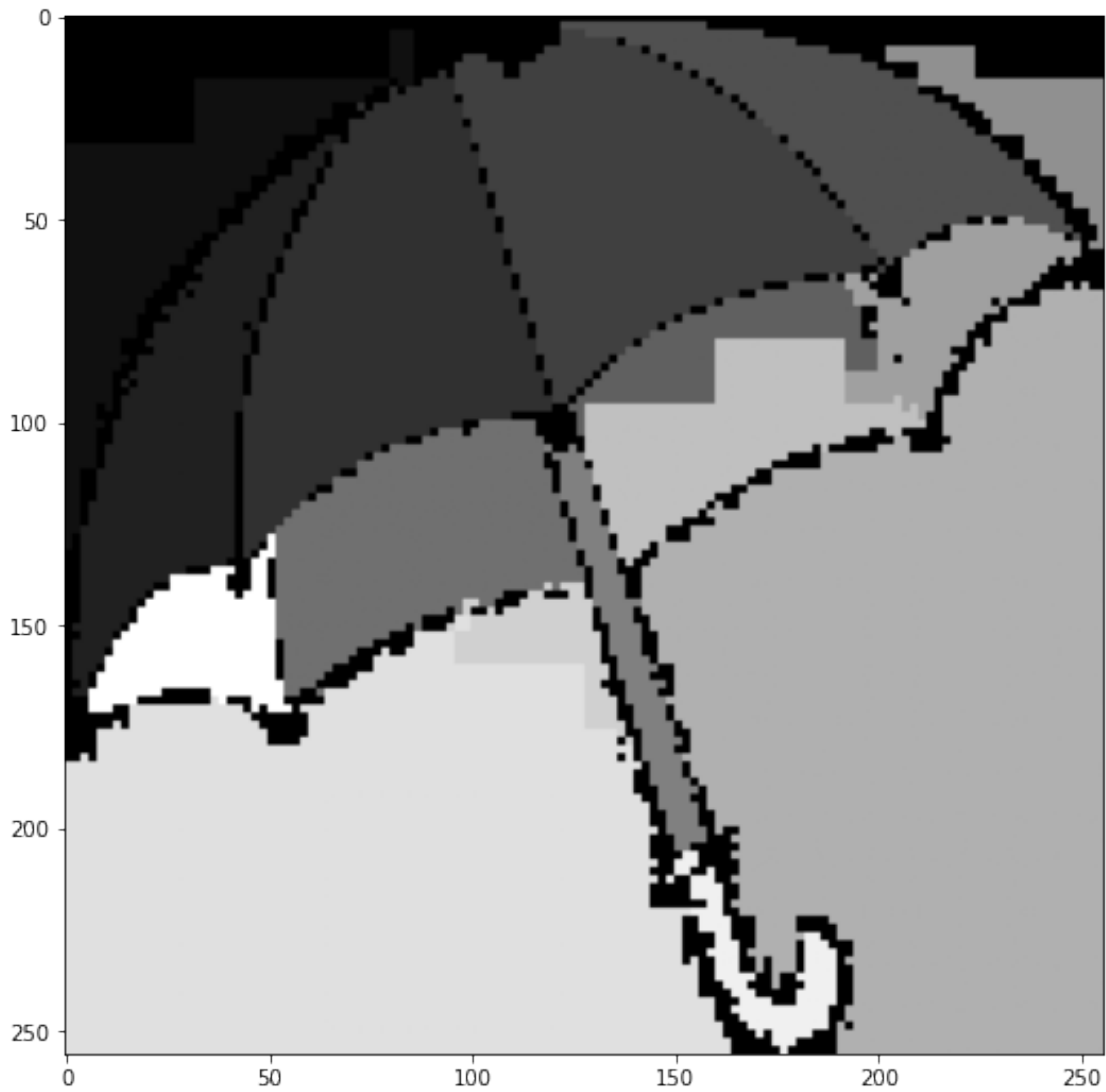
```
[27]: s = segmentation_by_splitting(h, min_size=4)
```

```
[28]: plt.imshow(s, cmap='gray')
plt.show()
```



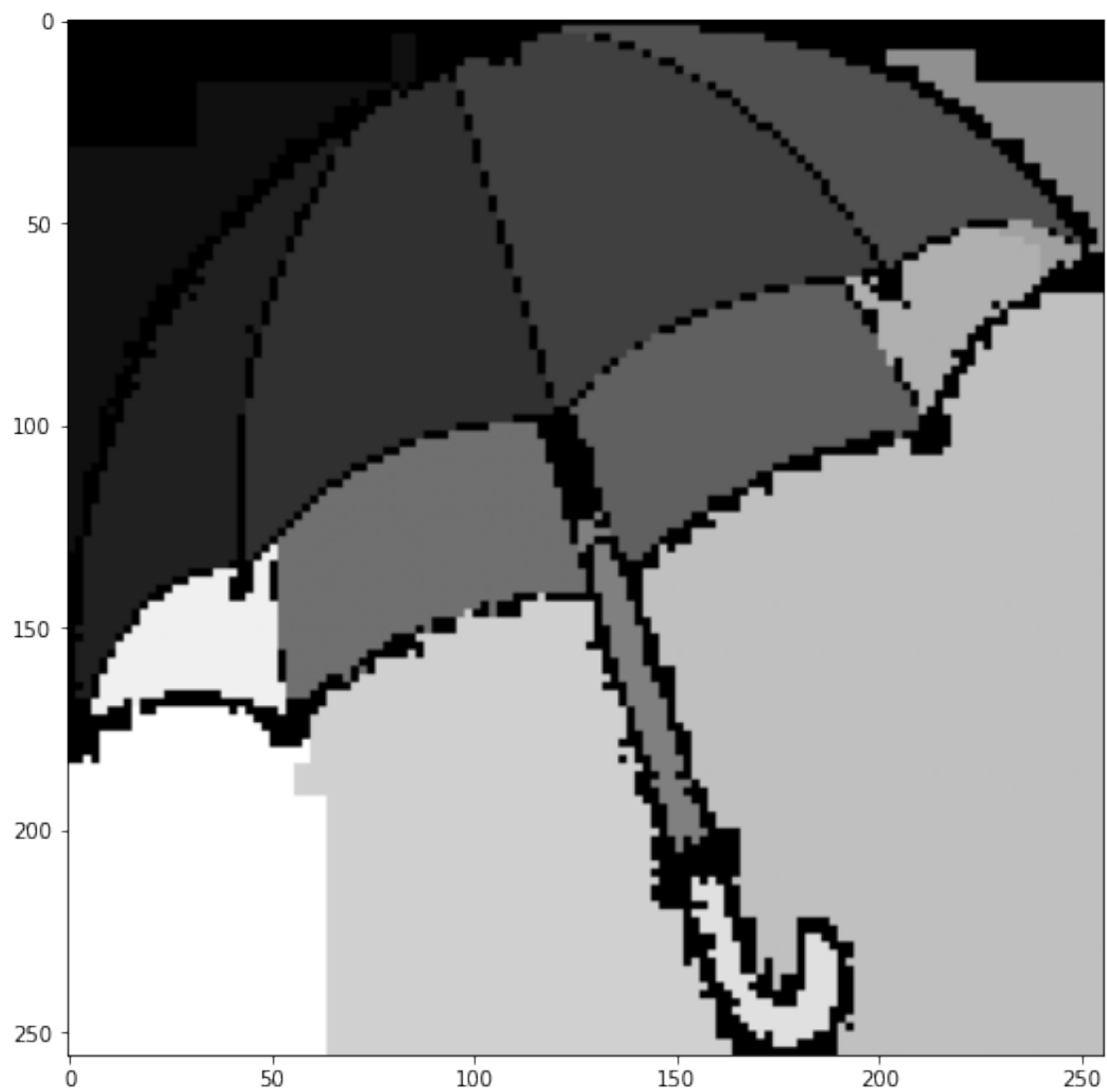
```
[29]: s = segmentation_by_splitting(h, min_size=2, mean_threshold=7, std_threshold=0.  
      ↪03)
```

```
[30]: plt.imshow(s, cmap='gray')  
      plt.show()
```



```
[31]: s = segmentation_by_splitting(h, min_size=2, mean_threshold=2, std_threshold=0.  
      ↪01)
```

```
[32]: plt.imshow(s, cmap='gray')  
plt.show()
```



```
[ ]:
```