Zaawansowne algorytmy wizyjne

materiały do ćwiczeń laboratoryjnych

Piotr Pawlik, Tomasz Kryjak

PUBLISHED BY AGH

First printing, March 2018

Contents

1 — Deskrypcja punktów charakterystycznych

1.1 Cel zajęć

- zapoznanie z zagadnieniem opisywania punktów charakterystycznych
- otoczenie punktu jako deskryptor
- porównanie działania metody Harrisa i SIFT

UWAGA: W dzisiejszym ćwiczeniu do wyświetlania obrazów proszę używać funkcji z matplotlib.pyplot a nie z opencv.

1.2 Operacje na listach w Pythonie

W trakcie zajęć kilkukrotnie pojawi się konieczność operowania na listach (np. punktów). Tradycyjnie operacje te wykonywane sa za pomocą pętli. Jednakże Python (oraz niektóre inne języki) dysponuje pewnymi mechanizmami, które pozwalają na uproszczenie implementacji operacji na listach. Oto niektóre z nich:

- list comprehension ('listy składane') : Zamiast pętli do tworenia listy można użyć konstrukcji przypominającej stworzenie listy przez podanie jej elementów, ale zamiast samych elementów podaje się algorytm ich wyliczenia. Przykładowo, tworząc listę kwadratów liczb, zamast:

  ```
  lst=[]
  for x in range(10):
    lst.append(x*x)
  ```

  można napisać:

  ```
  lst=[x*x for x in range(10)]
  ```

- funkcje wyższego rzędu: Do operowania na listach istnieją funkcje, które wykonają podaną operację na każdym elemencie funkcji. Jako parametr tym funkcjom podaje się funkcję realizującą tę operację. Przykładowo, mając listę z poprzedniego przykładu (kwadratów liczb) anb z niej wybrać liczby podzielne przez 3, zamast:

  ```
  podz3=[]
  for el in lst:
    if el%3 == 0:
    podz3.append(el)
  ```

  można napisać:

```
def przez3(x):
  return x%3 == 0
podz3=list(filter(przez3, lst))
```

Funkcja list jest tu potrzebna, gdyż funkcja filter zwraca iterator, który trzeba przekonwertować na listę.

- funkcje anonimowe: Analizując powyższy przykład mogą Państwo stwierdzić, że wersja z fliter nie jest aż takim uproszczeniem, skoro trzeba pisać niewiele robiącą funkcję *przez3*. Otóż nie trzeba. W przypadkach, kiedy potrzebujemy prostej funkcji (np. jako argument innej funkcji) można użyć funkcji anonimowej. Powyższy przykład z filter i funkcją anonimową sprowadziłby się do jednego polecenia i wyglądałby tak:

```
podz3=list(filter(lambda x:  x%3 == 0, lst))
```

Słowo kluczowe *lambda* rozpoczyna definicję funkcji anonimowej (czyli funkcji nienazwanej, bo nie interesuje nas tu jej nazwa, tylko to, co robi), po nim występują parametry tej funkcji (u nas jest to x). Po dwukropku mamy to, co funkcja ma zwrócić.

- łączenie list: Funkcje wyzszego rzędu działają zazwyczaj na pojedynczej liście. Co zrobić, jeżeli chcemy przeprowadzić jakąś operację jednocześnie na dwóch (lub więcej) listach? Należy je połączyć w jedną listę! Do tego służy funkcja zip. Przykładowo mając dwie listy współrzędnych punktów - jedna dla x-ów, druga dla y-ów możemy stworzyć listę par (x, y) przez:

```
lista_xy = list(zip(lista_x, lista_y))
```

1.3 Prosta deskrypcja punktów charakterystycznych

UWAGA - niniejsze ćwiczenie jest kontynuacją poprzedniego - będą w nim potrzebne zarówno obrazy jak i funkcje z poprzedniego ćwiczenia.

1. Ze strony kursu pobierz archiwum z dodatkowymi danymi do ćwiczenia i rozpakuj je we własnym katalogu roboczym.
2. Do funkcji z ubiegłego tygodnia dodaj funkcję tworzącą opisy punktów charakterystycznych. Opisem będzie wycinek obrazu z otoczenia punktu o rozmiarze podanym przez parametr. Jako parametry funkcja powinna otrzymać obraz, listę współrzędnych punktów charakterystycznych (wynik funkcji z poprzednich zajęć) oraz wielkość otoczenia. Z listy punktów należy usunąć wszystkie punkty, których otoczenia nie mieszczą się w obrazie. Można tu wykorzystać funkcję filter z anonimową funkcją lambda - przykładowo (X,Y to rozmiar obrazu, size to rozmiar otoczenia/wycinka):

```
pts=list(filter(lambda pt: pt[0]>=size and pt[0]<Y-size and pt[1]>=size
     and pt[1]<X-size, zip(pts[0],pts[1])))
```

Następnie należy stworzyć listę opisów punktów po odfiltrowaniu. Przy czym dobrze będzie sprowadzić go do wektora. Jeżeli np. p jest wycinkiem, to za pomocą **p.flatten(**) uzyskuje się z niego wektor. Jako wynik funkcji należy zwrócić listę otoczeń uzupełnionych o współrzędne ich punktów centralnych.
Dygresja - w tym celu można zzipować przefiltrowaną listę współrzędnych z listą otoczeń (funkcja zip) - przykładowo:

```
wynik_funkcji = list(zip(l_otoczen, l_wspolrzednych))
```

3. Dodaj funkcję porównującą opisy punktów charaktereystycznych z dwóch obrazów i znajdującą opisy najbardziej podobne. Funkcja jako parametry powinna otrzymać dwie listy opisów punktów charakterystycznych (z funkcji z punktu 2) oraz liczbę n najbardziej podobnych do siebie opisów. Funkcja może wykorzystać metodę 'każdy z każdym' - każdy

opis punktu z listy pierwszej jest porównywany ze wszystkimi opisami z listy drugiej i wybierany jest opis najbardziej podobny. Można różnie określić miarę podobieństwa. Może to być ich odległość wektorów, suma wartości bezwzględnych ich różnic, lub wynik ich iloczynu skalarnego. Współrzędne pary punktów, odpowiadających parze najbardziej podobnych do siebie opisów/wektorów, mają być umieszczona w liście wynikowej łącznie z miarą podobieństwa. Funkcja powinna zwrócić listę *n* najbardziej podobnych opisów. W tym celu można np. listę posortować (metoda sort) i wziąć jej pierwszych *n* elementów (można też zrobić to 'ręcznie' przez n-krotne wybieranie najbardziej podobnego dopasowania). Metoda *sort* działa 'w miejscu', czyli zastępuje listę listą posortowaną. Aby wykorzystać metodę sort do posortowania listy krotek należy przekazać tej metodzie funkcję, która określi po którym elemencie krotki chcemy sortować. Przykładowo sortowanie po drugim elemencie krotki to:

```
lst.sort(key=lambda x: x[1])
```

Innymi słowy parametr key occzekuje funkcji, która będzie zwracała kolejne elementy do sortowania. Zauważmy też, że x[1] - to odwołanie do **drugiego** elementu krotki, bo numerujemy od 0.
Standardowo metoda sort sortuje rosnaco. Jeżeli zależy nam na sortowaniu w porządku malejącym, to należy parametr *reverse* metody *sort* ustawić na True.

4. Wczytaj obrazy fontanna1.jpg i fontanna2.jpg. Znajdź dla nich punkty charakterystyczne metodą Harrisa za pomocą funkcji z poprzednich zajęć. Dla znalezionych punktów stwórz listy opisów za pomocą funkcji z punktu 2. Rozmiar otoczenia ustaw 15 (możesz poeksperymentować z innymi ustawieniami). Wyszukaj najlepsze dopasowania za pomocą funkcji z punktu 3. Parametr *n* można ustawić na 20
5. Pora na wyświetlenie rezultatów. Wśród danych do ćwiczenia znajduje się plik *pm.py* Można zaimportować go do swojego pliku (*import pm*) i wykorzystać znajdującą się w nim funkcję *pm.plot_matches*. Jednakże może być konieczne dostosowanie tej funkcji do Państwa listy zwróconej przez funkcję z punktu 3 (dostarczona implementacja zakłada, że każda para jest zapisana w postaci ([y1, x1], [y2,x2])).
6. Powtórz operacje z poprzednich punktów dla obrazów 'budynek1.jpg' i 'budynek2.jpg'. Zauważ, że na tych obrazach budynki są położone pod nieco innym kątem. Czy wpłynęło to na jakość dopasowania?
7. Powtórz operacje z poprzednich punktów dla obrazów 'fontanna1.jpg' i 'fontanna_pow.jpg'. Obrazy znaczne różnią się skalą. Czy Harris poradził sobie z taką różnicą?
8. Powtórz operacje z poprzednich punktów dla obrazów 'eiffel1.jpg' i 'eiffel2.jpg'. Obrazy różnią się jasnością i nieco skalą. Czy Harris poradził sobie z taką różnicą?
9. Zmodyfikuj funkcję wyznaczającą opisy punktów (z punktu 2), tak aby uwzględnić afiniczne zmiany jasności:

$$w_{aff} = \frac{w - \bar{w}}{|w - \bar{w}|} \tag{1.1}$$

Przy czym średnią $\bar{w}$ można wyznaczyć jako
np.mean(w)
a $|w - \bar{w}|$ jako odchylenie standardowe za pomocą
np.std(w).
Spróbuj teraz powtórzyć operacje dla wymienionych powyżej obrazów. Czy nastąpiła jakaś poprawa?

1.4 SIFT

1. Wykorzystując implementację metody SIFT z opencv przeprowadź operacje analogiczne do tych z sekcji 1.3 (znalezienie podobnych do siebie otoczeń punktów charakterystycznych). W tym celu wywołaj funkcję *cv2.xfeatures2d.SIFT_create()* która tworzy obiekt realizujący różne operacje na obrazach z wykorzystaniem metody SIFT. Następnie użyj dwukrotnie metody *detectAndCompute* tego obiektu dla dwóch obrazów: **fontanna1.jpg** i **fontanna2.jpg**. Parametrami tej metody powinny być obraz w odcieniach jasności i None. Metoda zwraca dwie listy - listę punktów charakterystycznych i listę ich opisów (deskrypcji). Do porównania opisów użyj 'porównywarki' z opencv - *BFMatcher* i jej metody dla k najbliższych sąsiadów (k=2, gdyż interesuje nas najbardziej podobne otoczenie oraz następne wg. podobieństwa - będzie to wykorzystane przy rysowaniu wyników). Przykładowo, dla list opisów desc1 i desc2 listę pasujących do siebie punktów matches uzyskuje się następująco:

```
bf = cv2.BFMatcher()
matches = bf.knnMatch(desc1,desc2, k=2)
```

Tak uzyskaną listę dopasowań można wyświetlić jako obraz zwracany prze funkcję *cv2.drawMatchesKnn* wywołaną z następującymi parametrami: pierwszy obraz, punkty charakterystyczne pierwszego obrazu, drugi obraz, punkty charakterystyczne drugiego obrazu, lista dopasowanych punktów (z knnMatch), obraz wyjściowy (ustawiamy na None, obraz wyjściowy jest także zwracany przez tę funkcję), flags=2 (bez tego argumentu wyświetlone zostaną wszystkie punkty charakterystyczne).
UWAGA - aby pozostawić tylko najlepsze dopasowania można z listy matches przed zawołaniem *drawMatchesKnn* usunąć te, których drugi sąsiad (z knn) ma dość podobne otoczenie. Realizuje to przykładowa pętla:

```
best_matches = [ ]
for m,n in matches: # m i n - najlepsze i 'drugie najlepsze'
dopasowanie
    if m.distance < 0.5*n.distance:  # najlepsze jest dwukrotnie
lepsze od 'drugiego'
        best_matches.append([m])
```

albo to samo w wersji z list comprehension:

```
best_matches = [ [m] for m,n in matches if m.distance < 0.5*n.distance]
```

2. Ponów operacje z powyższych punktów dla pozostałych obrazów przetwarzanych w sekcji 1.3. Jak SIFT radzi sobie w porównaniu z opisem bazującym na wycinkach (w szczególności dla obrazów **fontanna1.jpg** i **fontanna_pow.jpg**)?
Jeżeli linii jest za dużo i obraz jest przez to nieczytelny - wystarczy zwiększyć wymagania na to o ile najepsze dopasowanie jest lepsze od 'drugiego' (czyli **zmniejszyć** 0.5).

Bibliography