Семинарска работа по предметот Дигитално процесирање на слика на тема:

Алгоритми за креирање на панорамски слики од повеќе слики

Ментор: проф. д-р Ивица Димитровски

Изработила: Ана Алексова

Содржина

1. Вовед

Панорамските слики претставуваат композиција на повеќе фотографии во една широка слика која покрива пошироко поле на видливост од она што може да се добие со една обична фотографија. Тие овозможуваат детално прикажување на сцените и даваат поголема перспектива и длабочина.

Со напредокот на технологијата и софистицираните алгоритми за обработка на слики, процесот на креирање на панорамски слики стана многу полесен и подостапен. Денес, многу апликации и софтверски алатки овозможуваат автоматско спојување на повеќе фотографии во една панорамска слика со минимален напор од корисникот. Основата на овие технологии лежи во напредни алгоритми за детекција на клучни точки, совпаѓање на тие точки и употреба на хомографија за точно спојување на сликите.

Целта на оваа семинарска работа е да се истражат и опишат алгоритмите за креирање на панорамски слики од повеќе слики, како и да се презентира имплементација на таков алгоритам. Со тоа, ќе се овозможи подлабоко разбирање на процесот и техниките кои се користат за оваа цел, а истовремено ќе се демонстрира нивната примена преку практичен пример.

Во продолжение на семинарската работа ќе биде даден теоретски преглед на основните концепти на обработка на слики, детекција и совпаѓање на клучни точки, како и преглед на различни алгоритми за креирање на панорамски слики. Понатаму, ќе биде опишана имплементацијата на еден конкретен алгоритам, заедно со анализа на добиените резултати.

2. Користени алгоритми за креирање на панорамски слики

2.1. SIFT (Scale-Invariant Feature Transform):

SIFT е широко користен алгоритам во компјутерската визија за детектирање и опишување на клучните точки во слики. Со него се идентификуваат клучните точки во сликата и се генерираат дескриптори за секоја клучна точка кој може да се користи за совпаѓање на слични точки во различни слики. Ова го прави SIFT особено ефикасен за задачи како што се препознавање објекти, спојување на слики и 3D реконструкција.

Клучни фази на SIFT алгоритмот:

1. **Детекција на клучни точки**

SIFT прво ги идентификува потенцијалните клучни точки преку детектирање на екстреми во просторот на скали на сликата, кој се генерира со прогресивно заматување на сликата со Gaussian филтер. Со применување на Gaussian филтерот, сликата се заматува и на овој начин се генерираат различни верзии на сликата со различна резолуција.

Клучните точки се детектираат со споредување на секој пиксел со неговите соседи во истиот, претходниот и следниот слој на скалниот простор. Ако пикселот е максимум или минимум (екстремум) во однос на своите соседи, тогаш тој се означува како потенцијална клучна точка.

2. **Прецизирање на клучните точки**

Откако ќе се детектираат потенцијалните клучни точки, алгоритмот ги прецизира, односно ги филтрира и ги отстранува клучните точки со низок контраст и оние кои се слабо дефинирани по должината на рабовите.

Овој чекор осигурува дека остануваат само најстабилните клучни точки, кои се отпорни на бучава и грешки во идентификацијата.

Целта е да се задржат само оние клучни точки кои ќе дадат најпрецизни и најсигурни резултати при понатамошната анализа и споредба на сликите.

3. **Генерирање на дескриптори**

За секоја клучна точка се создава дескриптор преку пресметување на големината и ориентацијата на градиентот во околниот регион. Градиентните информации потоа се користат за градење на хистограм кој ја доловува локалната структура на сликата околу клучната точка. Добиениот дескриптор е високо-димензионален вектор (обично 128 димензии) кој уникатно ја претставува клучната точка.

Предности:

- **Непроменливост на скалата и ротацијата:** SIFT е робустен на промени во скалата и ориентацијата, што значи дека може да совпаѓа клучни точки дури и ако сликите се направени од различни агли или на различни нивоа на зумирање.
- **Дистинктивност:** SIFT дескрипторите се високо дистинктивни, што овозможува сигурно совпаѓање на клучните точки низ слики со различни осветлувања или услови на гледање.

2.2. BFMatcher (Brute Force Matcher)

BFMatcher е алгоритам што се користи за совпаѓање на дескриптори на клучни точки меѓу две или повеќе слики. Тој е еден од најпопуларните методи за совпаѓање во компјутерскиот вид, посебно во задачи како што е спојување на слики за креирање на панорами. BFMatcher користи "brute-force" пристап, што значи дека ги проверува сите можни парови меѓу дескрипторите од двете слики и го наоѓа најдобриот пар.

Клучни фази на BFMatcher:

1. **Споредба на дескрипторите:**

BFMatcher ги споредува сите дескрипторите од првата слика со сите дескриптори од втората слика со пресметување на Евклидово растојание. Растојанието претставува колку се слични два дескриптори, односно помало растојание укажува на поголема сличност.

2. **Совпаѓање на дескрипторите:**

Алгоритамот го идентификува најдобриот пар (или парови) за секој дескриптор, според пресметаните растојанија. Ако се користат K-Најблиски Соседи (KNN) за секој дескриптор се избираат k најблиски соседи, обично се користи k=2, што значи дека се земаат предвид двета најблиски совпаѓања.

3. **Филтрирање на совпаѓањата:**

Совпаѓањата добиени од претходната фаза може да вклучуваат и погрешни парови. Затоа, често се применува дополнителен чекор на филтрирање за да се отстранат несоодветните совпаѓања. Едната техника е cross-check, каде што се проверува дали и обратното совпаѓање (од втората кон првата слика) е валидно. Другиот начин на филтрирање е користење на тестот за однос (Ratio Test) кој ја споредува дистанцата на најблиското совпаѓање и второто најблиско совпаѓање.

Предности:

- **Едноставност:** BFMatcher е едноставен за имплементација и разбирање, што го прави добар избор за мали до средни сетови на податоци
- **Висок степен на разноврсност:** Иако често се користи со SIFT, BFMatcher може да се примени за совпаѓање на кој било тип на дескриптори, вклучувајќи ORB, SURF и други, што го прави разноврсен алат во компјутерската визија.

SIFT и BFMatcher работат заедно во задачи за спојување на слики за да ги детектираат клучните точки, да ги опишат на начин кој е робустен на скала, ротација и промени во осветлувањето, и потоа да ги совпаднат овие клучни точки низ слики. Овој процес на совпаѓање е клучен за точно порамнување на сликите и создавање беспрекорна панорамска слика.

3. Имплементација

3.1. Импортирање на библиотеките

```
1  import numpy as np
2  import cv2
3  import argparse
```

- **numpy**: овозможува работа со вектори, матрици и повеќедимензионални низи, кои се основни елементи за голем број алгоритми во компјутерската наука, особено во областа на машинско учење и компјутерска визија.

- **OpenCV**: се користи за обработка на слики и видеа. Во овој код, OpenCV се користи за читање и обработка на слики, како и за детекција на карактеристики и спојување на слики во панорама.

- **argparse**: се користи за обработка на аргументи од командната линија. Со неа може да се дефинираат аргументи кои програмата ќе ги прифати при извршување, како на пример имиња на датотеки и параметри. Ова овозможува флексибилност и динамичност на програмите, бидејќи аргументите можат да се менуваат без потреба од промена на кодот.

3.2. Промена на големината на сликите

Најпрво треба да се прилагодат влезните слики на одредена ширина, со цел да се осигура дека сите слики што ќе се обработуваат имаат исти димензии или барем се одржува истиот сооднос на пропорции. Ова е важно за процесите како детекција на карактеристични точки и спојување на слики, бидејќи големината и сооднос на пропорциите на сликите може да влијаат на прецизноста.

```
6  def resize_image(image, target_width=500):
7      (h, w) = image.shape[:2]
8      scale = target_width / float(w)
9      new_dim = (target_width, int(h * scale))
10     resized = cv2.resize(image, new_dim)
11     return resized
```

Со помош на функцијата *resize_image* ја менуваме големината на сликите, со цел секогаш, сите слики да се со иста големина.

- Со користење на *(h, w) = image.shape[:2]* ги добиваме димензиите на сликата, *h* – висина, *w* – ширина.
- Потоа со делење на посакуваната ширина што сакаме да ја има сликата, во овој случај 500 пиксели, со вистинската должина на сликата, ќе се добие скалата на ширината.
- Новата висина ќе ја добиеме така што ќе ја помножиме висината на сликата со скалата *(h * scale)*, и заедно со посакуваната должина ќе ги зачуваме во *new_dim*.
- Потоа со помош на функцијата *cv2.resize* од OpenCV ќе се промени големината на сликата според посакуваните димензии.

3.3. Процесирање на сликите

```
13  def process_images(image_paths, target_width=500):
14      images = []
15      for image in image_paths:
16          img = cv2.imread(image)
17          if img is not None:
18              resized_img = resize_image(img, target_width)
19              images.append(resized_img)
20          else:
21              print(f"Could not load image {image}")
22      return images
```

- За да може да се процесираат сликите кои треба да се обработат, најпрво иницијализираме празна листа, *images*, која ќе се користи за чување на обработените слики.
- Потоа, со помош на функцијата *cv2.imread(image)* од OpenCV, се чита сликата.
- Ако сликата е успешно прочитана, што се проверува со условот *if img is not None*, се продолжува со промена на големината на сликата со помош на функцијата *resize_image (img, target_width)*. Променетата слика се додава во листата *images*.
- Ако сликата не може да се прочита, се печати порака што информира дека сликата не може да се лоцира од дадената патека.
- На крај, функцијата ја враќа листата со сите обработени слики.

3.4. Детекција на клучни точки помеѓу сликите, наоѓање совпаѓања и прикажување на совпаѓањата

Функцијата detect_match_draw е дизајнирана да ги детектира, спореди, и визуелно да ги прикаже клучните точки помеѓу две последователни слики од листата *images* користејќи ги алгоритмите SIFT и BFMatcher.

```
23  def detect_match_draw(images, top_n=80):
24      sift = cv2.SIFT_create()
25      bf = cv2.BFMatcher()
```

- *sift = cv2.SIFT_create():* Создава објект на SIFT (Scale-Invariant Feature Transform) детекторот, кој се користи за детекција и опишување на клучните точки на слика.
- *bf = cv2.BFMatcher():* Создава објект на Brute-Force Matcher, кој ќе се користи за споредување на дескриптори на клучните точки помеѓу две слики.

Конверзија во сиви тонови

```
28      for i in range(len(images) - 1):
29          gray1 = cv2.cvtColor(images[i], cv2.COLOR_BGR2GRAY)
30          gray2 = cv2.cvtColor(images[i + 1], cv2.COLOR_BGR2GRAY)
```

- Циклусот for *i in range(len(images) - 1):* поминува низ секој пар на соседни слики од листата *images*.
- *gray1 = cv2.cvtColor(images[i], cv2.COLOR_BGR2GRAY):* Конвертирање на првата слика во сиви тонови.
- *gray2 = cv2.cvtColor(images [i + 1], cv2.COLOR_BGR2GRAY):* Конвертирање на втората слика во сиви тонови.

Придобивки од користење на сиви тонови

- **Зголемена ефикасност на обработката:**
  - Помалку податоци: Сивата слика има само еден канал (интензитет) во споредба со три канали (црвен, зелен, и сина) во бојата. Ова значи дека обработката на сивата слика бара помалку податоци и помалку пресметки, што ја зголемува брзината на алгоритмите.
- **Независност од боја**:
  - Фокус на интензитет: Алгоритмите како SIFT се дизајнирани да идентификуваат клучни точки и да ги опишат нивните карактеристики на основа на контрасти и текстури, кои се претставени со интензитети на светлина, а не со бои. Претворањето во сиви тонови го елиминира влијанието на бојата, што помага во конзистентна идентификација на клучните точки.

- Отпорност на промени во боја: Претварањето во сиви тонови ги елиминира разликите во боја кои можат да се појават поради различни осветлувања или камери, што ја зголемува отпорноста на алгоритмите на варијации во бојата.

- **Фокусираност на текстура и контрасти:**

  - Ефикасна детекција на карактеристики: Сивите слики често ги нагласуваат текстурите и контрастите, што е корисно за алгоритми кои се фокусирани на откривање на клучни точки како SIFT.

Детекција на клучни точки и дескриптори

```
32      keypoints1, descriptors1 = sift.detectAndCompute(gray1, None)
33      keypoints2, descriptors2 = sift.detectAndCompute(gray2, None)
```

Функцијата *detectAndCompute* го врши процесот на детекција на клучни точки и пресметување на дескриптори во еден чекор. Прво ги наоѓа сите карактеристични точки на сликата, а потоа за секоја точка генерира дескриптор кој ја опишува нејзината околина.

- keypoints1, descriptors1 = sift.detectAndCompute(gray1, None): Откривање на клучни точки и добивање на нивните дескриптори за првата сива слика.
- keypoints2, descriptors2 = sift.detectAndCompute(gray2, None): Откривање на клучни точки и добивање на нивните дескриптори за втората сива слика.

Споредба на дескрипторите користејќи BFMatcher

```
35      matches = bf.knnMatch(descriptors1, descriptors2, k=2)
36      good_matches = []
37      for m, n in matches:
38          if m.distance < 0.75 * n.distance:
39              good_matches.append(m)
```

- *bf.knnMatch* е функција која користи Brute-Force Matcher (BFMatcher) за споредба на дескрипторите помеѓу две слики.
- Дескрипторите (*descriptors1* и *descriptors2*) се математички претставувања на клучните точки во секоја слика, добиени со *sift.detectAndCompute*.

- *k=2* значи дека за секој дескриптор од првата слика, ќе се пронајдат двата најблиски дескриптори од втората слика. Ова создава парови на најдобри совпаѓања.
- Резултатот *matches* е листа од овие парови, каде секој пар содржи две совпаѓања (најдоброто и второто најдобро совпаѓање).

Принцип на "Ratio Test" за филтрирање на совпаѓања

for m, n in matches значи дека за секое од две совпаѓања (*m* и *n*) во листата *matches*, ќе се примени следното:

- *m* е најдоброто совпаѓање за даден дескриптор од првата слика.
- *n* е второто најдобро совпаѓање.
- if *m.distance < 0.75 * n.distance*: проверува дали растојанието (distance) помеѓу првиот и вториот најблизок дескриптор е значително различно.
- Принципот "Ratio Test" сугерира дека ако најдоброто совпаѓање (m) е многу поблиско до даден дескриптор отколку второто најдобро совпаѓање (n), тогаш ова е добро совпаѓање.
- 0.75 е фактор кој е емпириски утврден и ја одредува разликата помеѓу растојанијата; ако растојанието на најдоброто совпаѓање е помало од 75% од растојанието на второто најдобро собпаѓање, тогаш се смета за добро совпаѓање.
- *good_matches.append(m)* ги додава само оние совпаѓања (m) кои ги задоволуваат овие критериуми во листата *good_matches*.

Сортирање на совпаѓањата

```
41        top_matches = sorted(good_matches, key=lambda x: x.distance)[:top_n]
```

top_matches = sorted (good_matches, key=lambda x: x.distance)[:top_n]:

- **Сортирање според растојание (distance)**: Оваа линија го сортира списокот на добри совпаѓања (*good_matches*) според нивното растојание (*x.distance*). Растојанието мери колку се слични двата дескриптори кои одговараат на парот на клучни точки. Помало растојание укажува на поголема сличност меѓу дескрипторите, што значи дека точките се веројатно поточни за совпаѓање.
- **Избор на најдобрите совпаѓања (top_n)**: По сортирањето, кодот ги зема само првите top_n совпаѓања (најдобрите), кои најверојатно претставуваат реални совпаѓања меѓу двете слики.

Предности при избор на најдобри совпаѓања

- **Намалување на лажните совпаѓања**: Кога се совпаѓаат карактеристични точки, не се сите совпаѓања точни. Селектирањето само на најдобрите совпаѓања ги намалува шансите за користење на лажни совпаѓања, што би можело да резултира во погрешно порамнување на сликите.
- **Подобрување на прецизноста**: Овие најдобри совпаѓања се користат за финално порамнување на сликите, што директно влијае на квалитетот на крајната панорамска слика. Ако се користат премногу или лошо избрани совпаѓања, порамнувањето може да биде погрешно, што ќе резултира во слика со погрешни или изместени делови.
- **Перформанси**: Со селекција на ограничен број совпаѓања (*top_n*), исто така се подобруваат перформансите на алгоритмот, бидејќи помалку податоци се обработуваат во наредните чекори, како што е цртање на совпаѓањата или спојување на сликите.

Визуелизација

```
42          img_matches = cv2.drawMatches(
43              images[i], keypoints1, images[i + 1], keypoints2,
44              top_matches, None, flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS
45          )
46          cv2.imshow(f'Matches between Image {i + 1} and {i + 2}', img_matches)
47          cv2.waitKey(0)
```

- *cv2.drawMatches* е функција од OpenCV која создава слика на која се прикажуваат совпаѓањата помеѓу две слики.
- *images[i]* и *images [i + 1]* се две слики помеѓу кои се наоѓаат клучните точки.
- *keypoints1* и *keypoints2* се листи од клучни точки детектирании во првата и втората слика соодветно.
- *top_matches* е листата од најдобри совпаѓања помеѓу клучните точки на двете слики.
- *flags=cv2.DrawMatchesFlags_NOT_DRAW_SINGLE_POINTS* е параметар кој ја контролира визуелизацијата. цртајќи само линии кои ги поврзуваат совпаѓањата, но не и самите клучни точки.
- `cv2.imshow` е функција која ја покажува создадената слика во нов прозорец на екранот.
  - Првиот параметар е наслов на прозорецот и покажува кој пар на слики се прикажува. Променливата `i` се користи за идентификација на редоследот на сликите.
  - Вториот параметар е самата слика што треба да се прикаже (`img_matches`), која содржи визуелизација на совпаѓањата помеѓу двете слики.

- `cv2.waitKey` е функција која чека интеракција од корисникот.
  - Параметарот `0` значи дека програмата ќе чека бесконечно време за кориснички влез, односно, прозорецот ќе остане отворен додека не се притисне било кое копче
  - Ова е корисно за прикажување на сликата додека корисникот не е подготвен да продолжи со следната слика или дел од програмата.

3.5. Пост-обработка на добиената панорамска слика

Пост-обработката се постигнува со примена на различни техники за обработка на слики, како што се додавање рамка, конвертирање во сива боја, примена на праг, детекција на контури, и конечно сечење на сликата, со што значително се подобрува визуелната привлечност и квалитетот на панорамската слика.

Додавање рамка на панорамската слика

```
50  def post_process_stitched_image(stitched_img):
51      stitched_img = cv2.copyMakeBorder(stitched_img, 10, 10, 10, 10, cv2.BORDER_CONSTANT, value=(0, 0, 0))
```

Најпрво се додава рамка од 10 пиксели на секоја страна од споената панорамска слика (*stitched_img*). Функцијата *cv2.copyMakeBorder* од OpenCV е корисна за да се обезбеди простор околу сликата, што е потребно за следните чекори каде што се детектираат контури. Без рамката, може да се изгубат делови од сликата при процесот на отсранување на празните делови.

Конвертирање во сива боја и примена на праг(thresholding)

```
53      gray = cv2.cvtColor(stitched_img, cv2.COLOR_BGR2GRAY)
```

Споената панорамска слика се конвертира од BGR (Blue, Green, Red) формат во сива боја со помош на функцијата *cv2.cvtColor*. Тоа се посочува со аргументот *cv2.COLOR_BGR2GRAY*. Ова е неопходно бидејќи следниот чекор (примена на праг) бара црно-бела слика. Конвертирањето во сива боја ги поедноставува пресметките и го намалува бројот на информации што треба да се обработат, бидејќи сликата веќе не содржи информации за боја, туку само интензитет на светлината.

```
55      thresh_img = cv2.threshold(gray, 0, 255, cv2.THRESH_BINARY)[1]
```

Со оваа линија се применува праг врз сивата слика. Функцијата *cv2.threshold* го претвора секој пиксел во сликата во бел (255) или црн (0) зависно од неговиот интензитет. Прагот 0 значи дека OpenCV автоматски ќе го одреди најдобриот праг користејќи го алгоритмот *cv2.THRESH_BINARY*. Индексирањето со [1] укажува на тоа дека не интересира вториот резултат од функцијата - бинарната слика. На резултантната слика *thresh_img*, белите области претставуваат предмети или региони од интерес, а црните се позадина.

Детекција на контури

```
56    contours = cv2.findContours(thresh_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[0]
57    areaOI = max(contours, key=cv2.contourArea)
58    x, y, w, h = cv2.boundingRect(areaOI)
```

- Со функцијата *cv2.findContours* се детектираат контурите на белите региони во сликата thresh_img.
- Со *cv2.RETR_EXTERNAL* ќе бидат детектирани само надворешните контури, а сите внатрешни контури ќе се игнорираат.
- Со *cv2.CHAIN_APPROX_SIMPLE* се поедноставуваат контурите, оставајќи ги само основните точки што го дефинираат обликот на контурите.
- Со индексирањето [0] ја добиваме листата од контури

areaOI = max(contours, key=cv2.contourArea) ја наоѓа најголемата контура (област од интерес) од сите детектирани контури, според површината која е пресметана со помош на функцијата *cv2.contourArea* од OpenCV, која се користи за пресметување на површината на дадена контура.

x, y, w, h = cv2.boundingRect(areaOI): Го пресметува најмалиот правоаголник што ја опкружува контурата на areaOI (областа од интерес).

Создавање маска

```
60    mask = cv2.erode(cv2.rectangle(np.zeros_like(thresh_img), (x, y), (x + w, y + h), 255, -1), None)
61    sub = cv2.subtract(mask, thresh_img)
```

- Создава празна маска (матрица) со исти димензии како бинарната слика *thresh_img.*
- Матрицата е со нули, што значи дека сите пиксели во почетната маска се црни
- *thresh_img.shape*: го зема обликот (димензиите) на бинарната слика *thresh_img*, која е резултат од претходниот чекор (thresholding). Таа форма е обично во формат (висина, ширина).
- *cv2.rectangle(mask, (x, y), (x + w, y + h), 255, -1):* Со оваа линија, се создава правоаголна маска која ги обележува границите на areaOI. Правоаголникот ја дефинира областа што ќе биде задржана или обработена понатаму во сликата. На овој начин, други делови од сликата што се надвор од правоаголникот останууваат непроменети и може да бидат игнорирани во следните чекори на обработка.

```
63      while cv2.countNonZero(sub) > 0:
64          mask = cv2.erode(mask, None)
65          sub = cv2.subtract(mask, thresh_img)
```

while cv2.countNonZero(sub) > 0: Со овој услов се проверува разликата помеѓу намалената маска *mask* и бинарната слика *thresh_img*.

- *mask= cv2.erode(mask, None)*: Се намалува големината на белите области во сликата
- *sub = cv2.subtract(mask, thresh_img)*: Го одзема бинарниот приказ на *thresh_img* од намалената маска mask и го зачувува резултатот во sub.

Детекција на контури по намалувањето

```
66      areaOI=max(cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)[0], key=cv2.contourArea)
```

- Оваа линија ја избира најголемата контура од сите најдени контури

```
67      x, y, w, h = cv2.boundingRect(areaOI)
```

Оваа линија го пресметува најмалиот можен правоаголник кој целосно ја опфаќа избраната контурна област areaOI со помош на *cv2.boundingRect*.

- x, y: координатите на горниот лев агол на правоаголникот. x е хоризонталната координата, а y е вертикалната
- w, h: ширината (w) и висината (h) на правоаголникот. Со нив се дефинираат димензиите на правоаголникот што ја опфаќа целата контура.

Правоаголникот што го опфаќа целата контура се користи за да се утврди минималниот простор потребен за да се обиколи избраната контура. Ова е корисно затоа што дозволува прецизно да се исече делот од сликата што е важен, а непотребните празни делови да се отстранат.

```
68      return stitched_img[y:y + h, x:x + w]
```

- Оваа линија ја исекува оригиналната слика stitched_img така што останува само делот дефиниран од правоаголникот што го опфаќа контурната област areaOI.

- stitched_img[y:y + h, x:x + w] е операција со која се одредува делот од оригиналната слика што треба да се задржи:
    - y: y + h ги дефинира вертикалните граници
    - x:x + w ги дефинира хоризонталните граници

Преку ова исекување, сликата *stitched_img* се модифицира за да се задржи само делот кој ја содржи најголемата и најрелевантна област. Ова е последниот чекор во процесот на обработка, кој обезбедува финалната слика да биде уредна, без празни делови и да е прикажан само најрелевантниот дел.

3.6. Main функција

Функцијата main() во овој код служи како главна точка за извршување на програмата и е одговорна за контролирање на целиот процес на создавање панорамска слика од повеќе влезни слики.

Обработка на аргументи

```
70  def main():
71      parser = argparse.ArgumentParser(description="Stitch multiple images to create a panorama.")
72      parser.add_argument('images', nargs='+', help='Paths to input images')
73      args = parser.parse_args()
```

- **argparse.ArgumentParser** е библиотека за обработка на аргументи што се предаваат преку командната линија. Описот го објаснува основниот концепт на програмата - спојување на повеќе слики за да се создаде панорама.
- **parser.add_argument('images', nargs='+', help='Paths to input images')** додава аргумент наречен images кој ќе содржи патеки до една или повеќе слики. Параметарот nargs='+' значи дека се очекуваат една или повеќе патеки. help параметарот дава краток опис на аргументот
- **args = parser.parse_args()** ги обработува аргументите и ги складира во објектот args.

Промена на Големината на Сликите и Верификација

```
75      target_width = 500
76      images = process_images(args.images, target_width)
```

- **target_width = 500** задава целна ширина за промената на големината на сликите. Сите слики ќе бидат променети на оваа ширина за да се обезбеди конзистентност при понатамошната обработка.
- **images = process_images(args.images, target_width)** повикува функција за обработка на сликите (process_images), која ги чита сликите, ја менува нивната големина на зададената ширина, и ги складира во листа images.

```
78      if len(images) < 2:
79          print("Need at least two images to stitch")
80          return
```

- **if len(images) < 2:** проверува дали има најмалку две слики, бидејќи за спојување и креирање на панорама се потребни најмалку две слики.
- **print ("Need at least two images to stitch")** ако има помалку од две слики, се печати порака и програмата престанува со извршување преку return.

Детекција, спојување и приказ на клучни точки

```
82      detect_match_draw(images, top_n=80)
```

Се повикува функција која детектира клучни точки во сликите, ги споредува и ги црта совпаѓањата меѓу сите парови слики. Ова е визуелизација на клучните точки кои ќе се користат за спојување на сликите во панорама.

Спојување на сликите со cv2.Stitcher

```
84      imageStitcher = cv2.Stitcher_create()
85      error, stitched_img = imageStitcher.stitch(images)
```

- **imageStitcher = cv2.Stitcher_create()** креира објект од класата Stitcher, кој е дел од OpenCV и служи за автоматско спојување на слики.
- **error, stitched_img = imageStitcher.stitch(images)** ја спојува листата images во една панорамска слика. error е вратен код што индицира дали спојувањето било успешно. Ако е 0, спојувањето е успешно и резултира во stitched_img, што е финалната споена слика.

Обработка на панорамската слика

```
87      if not error:
88          stitched_img = post_process_stitched_image(stitched_img)
89
90          cv2.imshow("Panorama Image Processed", stitched_img)
91          cv2.waitKey(0)
```

- **if not error:** проверува дали нема грешка во спојувањето на сликите. Ако нема грешка, продолжува со обработка на споената слика.
- **cv2.imshow("Panorama Image Processed", stitched_img)** го прикажува финалниот резултат на споената и обработена панорамска слика.
- **cv2.waitKey(0)** чека кориснички влез за да затвори прозорецот.

```
92      else:
93          print("Images could not be stitched")
```

- **else:** ако спојувањето не е успешно, печати порака дека сликите не можат да се спојат. Ова може да се случи поради разни причини, како на пример недостаток на доволно совпаѓања меѓу сликите.

4. Добиени резултати

Сега ќе ги разгледаме резултатите добиени од примената на алгоритамите за креирање панорамски слики. Со користење на напредни техники за детекција и совпаѓање на клучни точки, заедно со ефективни методи за спојување на слики, постигнато е создавање на високо квалитетни панорамски слики.

Преку анализа на визуелните примери, ќе можеме да ја видиме успешноста на применетите методи. Овие примери ја демонстрираат ефикасноста на алгоритмите и ги покажуваат успешно креираните квалитетни панорамски слики.

Извршување на скриптата

Скриптата се извршува преку командната линија со внесување на наредбата *python skripta.py* (името на скриптата) и потоа патеките на сликите од кои што сакаме да добиеме панорамска слика.

```
PS C:\Users\Administrator\Desktop\panoramski_sliki> python skripta.py img1.png img2.png img3.png
```

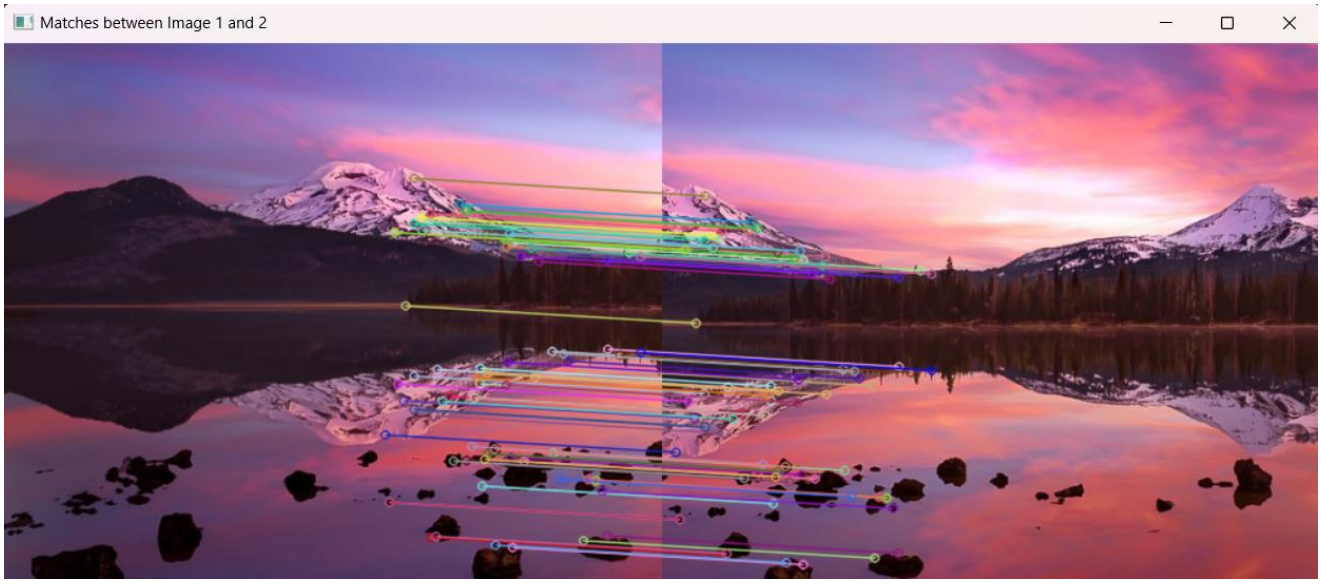
Со извршување на кодот, скриптата ги обработува овие слики, ги детектира, споредува и прикажува клучните точки , ги порамнува сликите и на крајот генерира финална панорамска слика која ги обединува сите влезни слики. Овие резултати ќе ги покажеме преку пример на три влезни слики.

Слика 1 Прва влезна слика

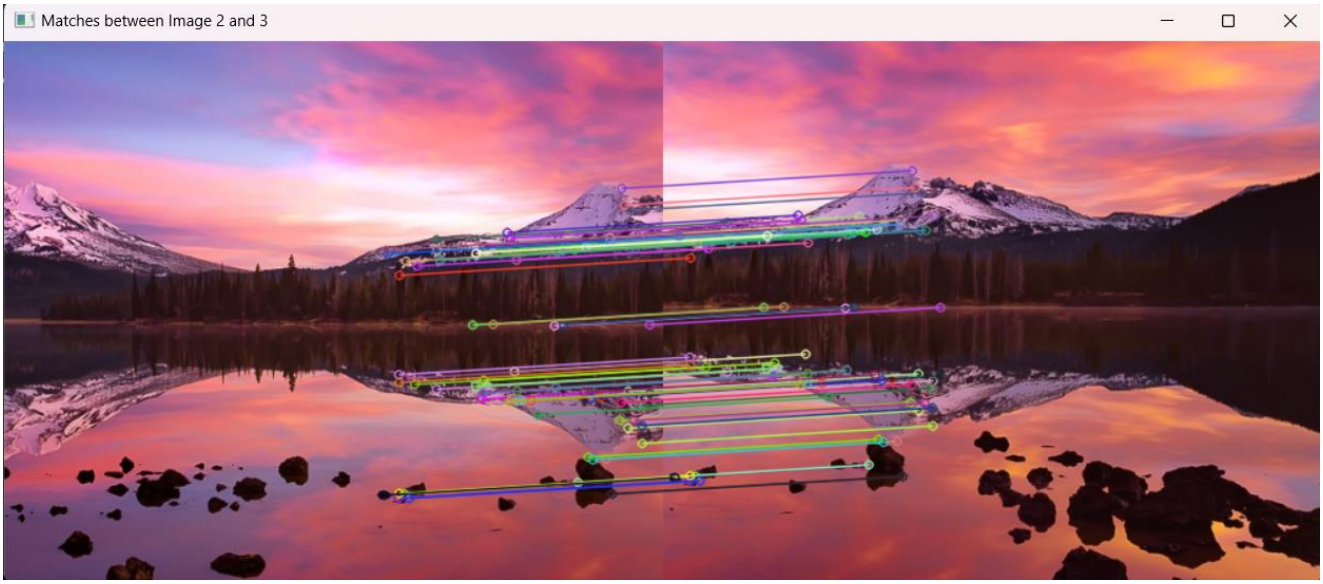
Слика 2 Втора влезна слика

Слика 3 Трета влезна слика

Резултати

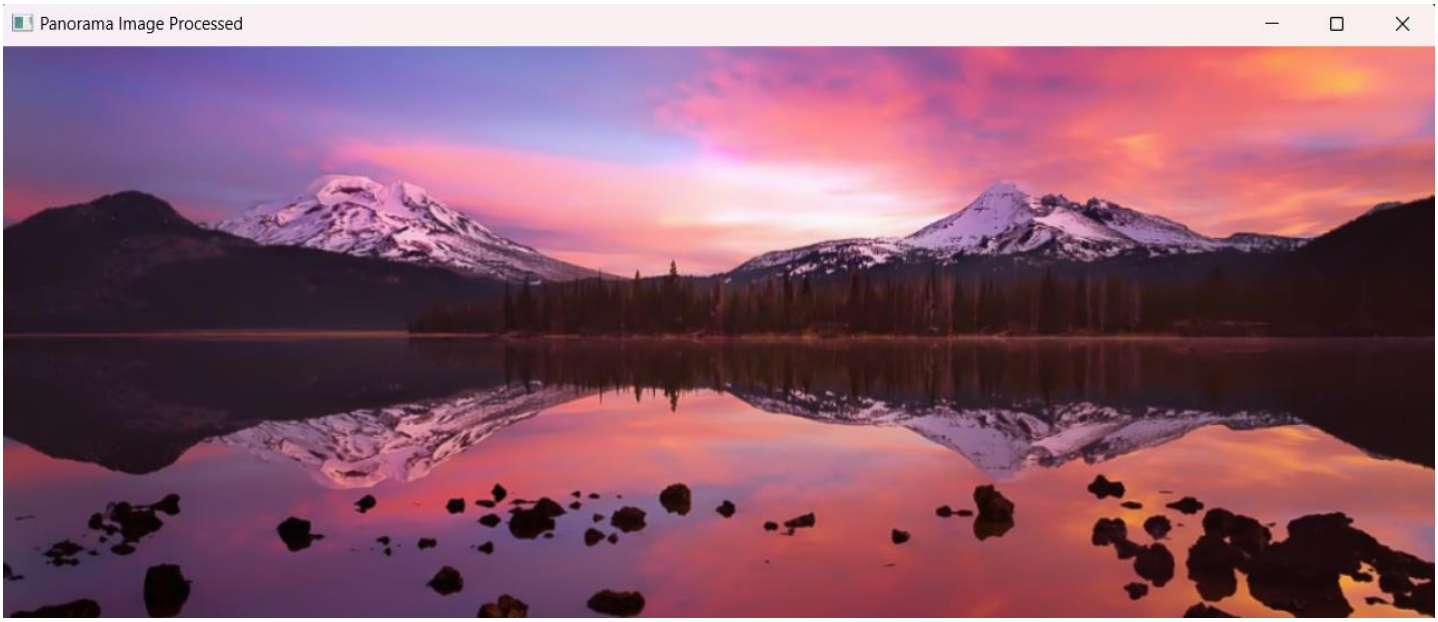


Слика 4 Совпаднати клучни точки помегу Слика 1 и Слика 2



Слика 5 Совпаднати клучни точки помеѓу Слика 2 и Слика 3

Панорамска слика



5. Заклучок

Во оваа семинарска работа беше разгледан и применет процесот на создавање панорамски слики од повеќе слики. Преку употреба на алгоритмот SIFT за детекција на карактеристични точки и нивно споредување со BFMatcher, овозможено е стабилно и прецизно порамнување на сликите.

Дополнително, значаен придонес има и пост-обработката на финалната панорамска слика. Со отстранување на несаканите празни делови, кои често се појавуваат по спојувањето на сликите, се обезбедува поуредена и визуелно поатрактивна финална слика.

Преку визуелните резултати може да се види дека успешно се искористени алгоритмите за детекција и спојување на сликите, што доведе до креирање на висококвалитетни панорамски слики. Конзистентноста и прецизноста на порамнувањето, како и ефективната пост-обработка, овозможуваат финална слика да биде од висок квалитет.

Овие резултати потврдуваат дека применетите техники се ефикасни и можат да се користат во различни сценарија каде е потребна автоматизирана обработка на повеќе слики. Ваквиот пристап ја демонстрира моќта на модерните алгоритми во полето на компјутерската визија и нивната практична примена.

6. Референци

[1] https://www.geeksforgeeks.org/image-stitching-with-opencv/
[2] https://pylessons.com/OpenCV-image-stiching
[3] https://www.opencvhelp.org/tutorials/image-processing/color-spaces-opencv/
[4] https://pyimagesearch.com/2018/12/17/image-stitching-with-opencv-and-python/
[5] https://www.educative.io/answers/what-is-sift
[6] https://docs.opencv.org/3.4/dc/dc3/tutorial_py_matcher.html