Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Irányítástechnika és Informatika Tanszék

3D Képfeldolgozás

Mérési Útmutató

Rendszer- és Alkalmazástechnika Laboratórium 1.

Szemenyei Márton, Kertész Zsolt

Tartalomjegyzék

1 3D számítógépes látás

A számítógépes látás alapvető célja, hogy egy kamera képe(i) alapján a valós világról automatikusan információkat tudjunk nyerni egy számítógépes algoritmus segítségével. A legtöbb gyakorlatban használt képalkotó rendszer azonban a valós, háromdimenziós világról egy kétdimenziós vetületet készít, amely során számos információ teljes mértékben elveszik vagy torzul. Egy tipikus képen nem maradnak meg az egyes képpontok kamerától mért távolságai, így ezeket csak becsülni lehet. Ezen felül belátható, hogy a vetítés művelete miatt számos, számunkra fontos geometriai jellemző is torzul. Ezen felül a térben egyenlő méretű objektumok a képen eltérő méretűek lesznek, ha a kamerától vett távolságuk más.

Ez a probléma különösen jelentős különböző autonóm robotokban vagy járművekben használt látórendszerek esetén, ugyanis ezeknek az eszközöknek pontos geometriai ismeretekre van szükségük az őket körülvevő térről. Ennek a problémának az egyik megoldása a gyakorlatban elterjedt egyidejű lokalizációs és térképkészítés (Simultaneous Localization and Mapping - SLAM) algoritmusa, amely a jelen mérés tárgyát képezi.

1.1. SLAM

SLAM algoritmusok esetén a környezetről inkrementális módon készítünk 3D rekonstrukciót, vagyis minden újonnan beérkező kép esetén hozzáadunk a már meglévő rekonstrukcióhoz, miközben a kamera új pozícióját becsüljük. Ebben az esetben a becslést általában az előző képhez és állapothoz képest végezzük. A valósidejűségi követelmény miatt a párosításokat általában optikai áramlás, vagy hasonlóan gyors módszer segítségével végezzük. Egy tipikus SLAM algoritmus lépései a következők:

1. Jellemző detektálás
2. Az előző kép jellemzőivel párosítás
3. Kamera póz becslése
4. 3D pont koordináták becslése

5. Csoport igazítás (Bundle Adjustment)

A SLAM típusú algoritmusok egyik alapvető problémája a csúszás (drift) jelensége. Ez a jelenség abból adódik, hogy a kamera új pozícióját mindig az előző pozícióhoz képest becsüljük, így a kamera abszolút pozíciója relatív elmozdulások összességéből adódik. Mivel végtelen pontosan becsülni lehetetlenség, ezért az újabb és újabb becslések hibája egyre inkább halmozódik, és a kezdetekben kicsi hiba egyre nagyobb lesz, vagyis a becsült pozíció fokozatosan „elcsúszik" az igazitól.

Ennek a jelenségnek az orvoslására számos módszer létezik, melyek közül az első, hogy a relatív elmozdulást nem csak az előző, hanem korábbi képekhez képest is megbecsüljük, így mérsékelve a csúszás mértékét. Ezt természetesen nem lehet a végletekig művelni, hiszen a mozgó kamera egy idő után új területeket lát majd, és nem lesz átfedés a túl régi képkockákkal, így közös jellemzőket sem fogunk találni. Egy másik megoldás a hurokzár detektálás, melynek során érzékeljük, ha a kamera visszaért egy korábban meglátogatott pozíció közelébe, és ebben az esetben a korábbi pozícióban készített képkockához képest becslünk pozíciót. Általánosságban is lehetséges az a megoldás, hogy az előző képkockák mellett a már elkészült térképrészlethez képest is becsüljük a lokalizációt.

A SLAM algoritmusok implementációját rendkívüli módon egyszerűsíthetjük, ha a bemenetként kapott képhez mélységi információnk is tartozik. Ebben az esetben RGB-D SLAM megoldásról beszélhetünk, amely a hagyományos módszerrel szemben metrikus térképet eredményez. Ekkor ugyanis a kamera elmozdulását nem 2D, hanem 3D pontpárokból végezhetjük el, amely egzakt módon és egyértelműen megoldható.

1.1.1. Képjellemzők

A képek közötti párosítás problémáját igyekszik kezelni a skálainvariáns képjellemző transzformáció, vagyis a SIFT-algoritmus (Scale Invariant Feature Transform), amely a perspektív torzítás kivételével minden transzformációra invariáns, így robusztus lokális régióleírót produkál, amelyet széles körben alkalmaznak. Alapelve, hogy sarokszerű pontokat keres a képen, azonban ezekhez nem egyetlen mértéket, hanem a sarokpontok lokális környezetét invariáns módon leíró kódot készít, amelynek segítségével más képeken megtalált jellemzőkkel összevethető, párosítható.

Ezt a párosítást általában a jellemző vektorok közötti távolság segítségével végezzük. Olyan algoritmusok, amelyek valós számokból álló vektorokat eredményeznek (pl.: SIFT, SURF, KAZE) célszerű a négyzetes távolságot alkalmazni, míg bináris

leíró vektorok (ORB, AKAZE) esetén a Hamming-távolság a megfelelő választás. Pusztán a legközelebbi szomszéd megkeresése azonban nem elegendő jó párosítások létrehozására, ugyanis legközelebbi szomszédja azoknak a jellemzőknek is lesz, amelyeknek valamilyen oknál fogva nem található párja a másik képen.

Erre a problémára két lehetséges megoldás is létezik, melyeket gyakorta használnak együtt. Az első az ún. arányteszt: ekkor az adott jellemzőnek nem csak a legközelebbi szomszédját keressük meg a másik képen detektált jellemzők közül, hanem a második legközelebbit is. Ezt követően, ha azt találjuk, hogy a legközelebbi szomszéd leírójától vett távolság lényegesen kisebb, mint a második (vagyis a kettő aránya jelentősen kisebb 1-nél), akkor jó minőségű, egyértelmű párosítást kaptunk, ellenkező esetben a párt elvetjük.

A második lehetséges megoldás a szimmetriateszt: Ekkor, ha azt találjuk, hogy az egyik képen talált x jellemző legközelebbi párja a másik képen található jellemzők közül y, megkeressük y legközelebbi szomszédját az első képen található jellemzők közül. Ha ez az x jellemző, akkor a párosítás jó, mivel x legjobb párja az y, és ez fordítva is igaz, vagyis a pár szimmetrikus. Ellenkező esetben a párt elvetjük.

1.1.2. Geometriai Becslések

Pinhole kamera

A számítógépes látásban gyakran használt a pinhole kameramodell. A pinhole kamera egyszerűen elképzelhető úgy, mint egy doboz, aminek az egyik oldalán vagy egy kis lyuk, amin keresztül fény képes beáramlani. A lyukon beérkező fény hatására a doboz ellenkező oldalán egy fordított állású kép keletkezik. Valódi kamerák esetén itt helyezkedik el a fényszenzor. A valódi kamerák további jelentős különbsége, hogy egyetlen kis lyuk helyett egy lencsét alkalmaznak, ami a párhuzamos fénysugarakat egy helyre fókuszálja, így képes a pinhole-t helyettesíteni. A lencse alkalmazásának előnye, hogy lényegesen több fényt ereszt be, mint a pinhole, azonban – ahogy azt az előző kötetben is tárgyaltuk – geometriai torzítást okoz a képen. A pinhole kameramodell az alábbi egyenletek segítségével írható le:

$$u = f_x \frac{x}{z} + p_x; \quad v = f_y \frac{y}{z} + p_y; \tag{1.1}$$

Ahol u és v a pixelek koordinátái, x, y, és z az objektum térbeli koordinátái, f a kamera fókusztávolsága, p pedig a principális pont. Amennyiben f, p és z ismertek, akkor az egyenlet könnyedén megfordítható, és a 3D rekonstrukció minden további nehézség nélkül elvégezhető.

A RANSAC algoritmus

Amennyiben legalább két 3D pont pár rendelkezésünkre áll, a köztük lévő Euklideszi transzformáció az SVD-felbontás segítségével meghatározható. A téves párosítások torzító hatásának elkerülése végett azonban célszerű a robusztus RANSAC algoritmus használata. A RANSAC alapvetően egy univerzális paraméterbecslési eljárás, amelyet a pontfelhők szegmentálásán kívül még számos helyen használnak: többek között kamerakalibrációra is.

Az algoritmus alapelve rendkívül egyszerű: egy ponthalmazból véletlenszerűen pontokat kiválasztva egy jelöltet készít a megoldásra, majd ezt újabb véletlen választásokkal ismételve nagy számú véletlen jelöltet állít elő. Ezt követően megvizsgálja, hogy az egyes jelöltekre a teljes ponthalmazból hány pont illeszkedik rá. Az egy adott jelöltre illeszkedő pontokat inlier-nek hívjuk. A RANSAC algoritmus minden jelöltre összeszámolja az inliereket, és eredményként visszaadja a legtöbb inlierrel rendelkező jelöltet. A képjellemzők párosításáról, valamint a SLAM és RANSAC algoritmusokról bővebben a Számítógépes Látórendszerek c. tárgy jegyzetében [1] olvashat.

2 A mérés környezete

A mérés során a *PyCharm* elnevezésű IDE áll rendelkezésre, amely rendkívül sokoldalú szolgáltatásokkal könnyíti meg a szoftverfejlesztést, például konfigurálható automatikus formázási lehetőségek állnak rendelkezésünkre. További részletekért érdemes lehet a JetBrains ide vonatkozó weboldalát [2] felkeresni. Függvények, objektumok esetében a ***Ctrl+P*** billentyűkombináció pop-up segítségként szolgálva mutatja nekünk a paramétereket. A mérés során használt programnyelv a Python 3-as verziója lesz.

A Python programnyelvhez számos hasznos függvénykönyvtár tartozik, melyek a mérési feladatok megvalósítását nagymértékben megkönnyítik. A Python nyelv egyik rendkívül kényelmes funkciója a beépített package manager, amelynek segítségével az egyes könyvtárak automatikusan telepíthetők, telepítsük után pedig minden további beállítás nélkül használhatók. A Pythonhoz két ilyen package manager is tartozik, az egyik a Pip, amely a legtöbb telepíthető Python verzió mellé automatikusan települ, a másik pedig az Anaconda [3], ami a könyvtárkezelési funkciókon túl virtuális környezeteket is képes kezelni.

A Python egyik legfontosabb függvénykönyvtára a Numpy, amely tömbök kezelésére, illetve számtalan numerikus algoritmus használatára ad lehetőséget. A Numpy funkcionalitását kiegészíti a MatplotLib, melynek segítségével különböző ábrákat készíthetünk a tömbjeinkről. Egy harmadik rendkívül hasznos könyvtárcsalád a scikit, ami számos tudományos számításhoz szükséges alkönyvtárt foglal össze. A sckit-image képek kezelésére, a scikit-learn gépi tanulás algoritmusok használatára, míg a scikit-fuzzy fuzzy logika használatára ad lehetőséget. Ezek a könyvtárak tulajdonképpen együttesen kiadják a Matlab funkcionalitásának jelentős részét.

2.1. Haladó Python

A korábbi mérés(ek) során megismerkedtünk a Python nyelvvel, a jelen mérés során azonban a nyelv bonyolultabb funkcióira is szükségünk lesz. Az alábbiakban tekintsük át legfontosabb adattípusokat és nyelvi elemeket.

A Python egy fontos típusa a *None*, amely nagyjából a null pointerrel ekvivalens objektum. Gyakori, hogy egyes függvények sikertelen futás esetén ezt adják visszatérési

értéknek, így célszerű tudni, hogy hogyan lehetséges egy objektum esetén a *None* státuszt ellenőrizni. Fontos tudni, hogy a hagyományos == operátor alkalmazása ekkor kerülendő, ugyanis bizonyos osztályok ezt az operátort felüldefiniálhatják, és ekkor nem kívánt működést kaphatunk.

```
class Foo:
    def __eq__(self,other):
    return True

foo=Foo()
print(foo==None) # True
print(foo is None) # False
```

A Python legfontosabb konténer osztályai a list és a tuple. Ezek közt a különbség, hogy a tuple immutable, vagyis nem módosítható. Létrehozásuk és az elemeik elérése az alábbi módon történhet:

```
myList = [”wow”,”you”,”can”,”mix”,2,”or”,”more”,”different”,”types”]
myTuple = (”wow”,”you”,”can”,”mix”,2,”or”,”more”,”different”,”types”)
myList[4] = ”two” # 2 becomes ’two’
myTuple[4] = ”two” # Error
```

Érdemes tudni, hogy habár (1D) listák esetében a + operátor értelmezett, ez a listákat összefűzi, nem pedig összeadja.

```
myList = [1,2]+[3,4]
    >>> [1,2,3,4] # Not [4,5]
```

Listákhoz új elemet az alábbi módon lehet hozzáadni:

```
myList = [1,2,3,4]
myList.append(5) # [1,2,3,4,5]
```

Listák, vagy Tuple-ök elemein az alábbi módokon lehet végigiterálni:

```
for elem in myList:
    print(elem)

for i, elem in enumerate(myList):
    print(i,elem) # i is the index
```

Adott esetben egyszerre több (egyenlő elemszámú) listán is végigiterálhatunk egyidejűleg:

```
if len( list1 ) == len(list2):
    for (elem1,elem2) in zip( list1 , list2 ):
        print(elem1,elem2)
```

Érdemes megjegyezni, hogy az 'in' kulcsszó feltételeknél is használható annak eldöntésére, hogy egy adott érték szerepel-e a listában:

```
myList = [1,2,3,4]
if 1 in myList:
    print("1 is in the list")
if 5 not in myList:
    print("5 is not in the list")
```

A Python programnyelv támogat bizonyos funkcionális programozási elemeket is, melyek közül az egyik leghasznosabb az úgynevezett list comprehension. Ezt akkor használhatjuk, ha szeretnénk egy lista minden elemén elvégezni egy műveletet és ezek eredményeiből egy újabb listát készíteni. Az alábbi példa egy lista minden elemét négyzetre emeli:

```
myList = [1,2,3,4]
mySqrList = [elem**2 for elem in myList] # [1,4,9,16]
```

A list comprehension a korábban említett for ciklus változatokkal (zip, enumerate) is használható, az új lista elemének kiszámolásához pedig bármilyen érvényes Python kód írható. Ezen felül a kifejezés feltétellel is kiegészíthető:

```
myList = [9,-16,25,-4]
mySqrtList = [sqrt(elem) for elem in myList if elem >= 0] # [3,5]
```

2.2. Haladó NumPy

Fontos megjegyezni, hogy a Python list és tuple osztályok különböző típusú objektumokat tartalmazhatnak, vagyis egy lista egyik eleme lehet egy másik lista. Ez lehetőséget nyújt arra, hogy a lista osztályt N-dimenziós tömbként használjuk, azonban egy listákat tartalmazó listában az egyes részlisták különböző hosszúak és típusúak lehetnek, ami hagyományos tömböknél nem fordulhat elő. Éppen ezért méret- és típus konzisztens tömbök tárolására a NumPy array osztályt célszerű használni.

A NumPy tömbök között definiált a $+$, $-$, $*$ és a $/$ művelete, azonban ezeket elemenként végzi, így a tömbök méretének kompatibilisnek kell lennie. A NumPy broadcasting szabályai alapján két tömb az alábbi feltételek teljesítése esetén kompatibilis méretű:

1. A két tömb minden mérete megegyezik
2. Amelyik dimenzió mentén nem egyeznek meg a méretek, ott az egyik tömb mérete 1

A tömbök méretét az alábbi módon ellenőrizhetjük:

```
arr1.shape # [2,4,6]
arr2.shape # [1,4,1]  - Compatible
arr3.shape # [3,4,6]  - Incompatible
```

Bizonyos esetekben előfordulhat, hogy új elemet szeretnénk a tömbhöz adni, vagy két tömböt össze szeretnénk fűzni, amelyeket az alábbi módon tehetünk meg:

```
extArr = np.append(arr,elem,dim)
catArr = np.concatenate((arr1,arr2 ,..., arrn),dim)
```

A *dim* változó azt adja meg, hogy melyik dimenzió mentén történjen a kiegészítés/összefűzés. Ha None értéket adunk meg (nem adjuk meg a paramétert), akkor a NumPy megpróbálja magától kitalálni.

Gyakran előfordul, hogy egy általunk használt tömbben van egy extra dimenzió, aminek az értéke egy. Például, ha van egy 2D mátrixokból álló tömbünk, amelyiknek kivesszük egy elemét. Ez az extra dimenzió sok fejfájást tud okozni, mert ha konkatenálni, vagy mátrixokat szorozni szeretnénk, a NumPy nem megfelelő méret hibákat fog dobni. Ennek megoldására a squeeze parancs való.

```
mtxArr.shape # [N,3,3] - N 3x3 matrices
myMtx = mtxArr[i]
myMtx.shape # [1,3,3]
myMtx = mtxArr[i].squeeze()
myMtx.shape # [3,3]
```

Előfordulhat az is, hogy a tömbjeink megfelelő méretűek, de valamilyen oknál fogva a dimenziók sorrendje eltér. Például képek esetén elterjedt konvenció a $H \times W \times Ch$ sorrend használata, a Deep Learning könyvtárak viszont a $Ch \times H \times W$ sorrendet preferálják. Erre megoldás a NumPy transpose függvény.

```
myArr.shape # [N,3]
np.transpose(myArr).shape # [3,N]

# Több dimenzióban
myImg.shape # [1080,1920,3]
np.transpose(myImg,(2,0,1)).shape # [3,1080,1920]
```

Ezen felül érdemes még említeni, hogy a NumPy képes különböző lineáris algebrai műveletek elvégzésére, mint például a mátrixszorzás (amennyiben a dimenziók megfelelőek), valamint az invertálás:

```
myMtx = np.array  ([[1,2,3],[4,5,6],[7,8,9]])
myInv = np.linalg.inv(myMtx)
```

```
myData = np.random.randn(50,3) # 50 3D coordinates (Normal distribution)
myTrData = np.matMul(myMtx,myData) # myTrData = myMtx*myData
```

Érdemes ezen felül azt is tudni, hogy a NumPy támogat magas szintű indexelést. Ez azt jelenti, hogy egy tömbbe nem csak egyetlen, hanem egyszerre több (akár egy egész tömbnyi) indexszel is indexelhetünk.

```
myData = np.random.rand(50,3) # 50 3D coordinates (Uniform distribution)

# Get 1.,5.,13., and 44. rows
myRowIndices = np.array([1,5,13,44])
myRows = myData[myRowIndices]

# Get 1. and 3. columns
myColumnSelectors = np.array([True,False,True])
myColumns = myData[:,myColumnSelectors]

# Index using a condition
myPositiveData = myData[myData>0]
```

3 Mérési feladatok

A feladatok megoldása során célszerű arra figyelni, hogy az alábbi mintakódokban bizonyos függvények listákat várnak/adnak vissza, míg más metódusok 1-1 objektummal működnek. A változók elnevezése (egyes-többes szám) ezt egyértelműen jelzi.

FONTOS: Az alábbi feladatokat célszerű list comprehension segítségével megoldani, ugyanis ez a hagyományos kibontott for ciklusoknál 30%-kal gyorsabb.

3.1. Feladat: Geometriai segédfüggvények megvalósítása

A feladathoz a *Geometry.py* fájlban kell dolgozni.

3.1.1. Feladat: Írjon eljárást, amely a kameramátrix ismeretében meghatározza egy $[u, v, d]$ képkoordináta-mélység pár 3D pozícióját!

A feladathoz a **pts23D** függvényt kell megvalósítani. A függvény bemenetei:

- *center*: egy tuple, amely az adott pont képkoordinátáit tartalmazza
- *depth*: egy szám, amely az adott pont mélységét tartalmazza milliméterben
- *A*: egy NumPy tömb, amely a kameramátrix értékeit tartalmazza

3.1.2. Feladat: Írjon eljárást, a bemenetére kapott 3D pontokat egy szintén bementként kapott projektív transzformációs mátrix segítségével transzformál!

A feladathoz a **transformPoints** függvényt kell megvalósítani. A függvény bemenetei:

- *pts*: egy $N \times 3$ NumPy tömb, melynek minden sora egy 3D pont
- *mtx*: Egy 3×4 mátrix, amely egy euklideszi transzformáció

Kimenetként szintén $N \times 3$ méretű tömböt várunk.

3.2. Feladat: Végezze el két egymást követő képen a képjellemzők detektálását és robusztus párosítását!

3.2.1. Képjellemzők detektálása

A feladathoz a *SLAM.py* fájlban definiált **SLAM** osztály konstruktorát, valamint az **addFrame** függvényét kell módosítani.

AKAZE képjellemző detektor létrehozása (konstruktorba)

```
self.feat = cv2.AKAZE_create(cv2.AKAZE_DESCRIPTOR_KAZE,threshold=0.0005)
```

Kép szürkárnyalatosítása

```
img_gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
```

Kulcspontok detektálása. A kimenet egy lista, melynek minden eleme egy kulcspont objektum.

```
keypoints = self.feat.detect(img_gray)
```

FONTOS! A keypoints lista létrehozása után csak azokat a kulcspontokat szabad megtartani, amelyekhez 0-nál nagyobb depth érték tartozik. List comprehension segítségével lehetséges ugyanannak a listának a felülírása.

Depth érték meghatározása egy kulcsponthoz. Ez azért szükséges, mert a detektált kulcspontok pozíciója szubpixeles pontossággal lett meghatározva (értsd: a koordináták lebegőpontos számok), így nem tudunk a depth képtömbbe egyszerűen beleindexelni. Ezért az alábbi függvény segítségével bilineáris interpolációt hajtunk végre.

```
kpDepth = getSubpix(depth,keypoint)
```

Leírók számítása. A descriptors lista minden eleme a vele megegyező indexű kulcsponthoz tartozó leírót tartalmazza.

```
keypoints, descriptors = self.feat.compute(img_gray,keypoints)
```

3D koordináták meghatározása. A *kpDepth* meghatározása itt a fentihez hasonló módon történik.

```
pt3D = pt23D(keypoint.pt,kpDepth,self.A)
```

Feature létrehozása egy kulcsponthoz

```
Feature(pt3D,descriptor)
```

A fenti sorok egymásba ágyazásával az összes kulcsponthoz tartozó Feature objektum egyszerűen előállítható. (Tipp: a zip segítségével a keypoints és a descriptors listák egyszerre iterálhatók.)

3.2.2. Képjellemzők párosítása

A feladathoz a *Feature.py* fájlban definiált **match** függvényét kell megvalósítani. Ebben a korábban ismertetett arány-alapú párosítást kell megvalósítani.

Matcher létrehozása

```
matcher = cv2.FlannBasedMatcher()
```

kNN párosítás meghívása (2 legközelebbi szomszédot keresünk).

```
matches = matcher.knnMatch(dSrc,dDst,k=2)
```

A kapott *matches* lista minden eleme további két elemre bontható. Ezek közül az első a legjobb, a másik pedig a második legjobb párosítás. Egy párosítás esetén a jellemző leíró vektorok négyzetes távolságát a *distance* tagváltozó tartalmazza.

Ezt követően alkalmazzuk az aránypár alapú szűrést. Egy párosítást akkor érdemes megtartani, ha a legközelebbi és a második legközelebbi szomszédok távolságának aránya egy bizonyos küszöbértéknél kisebb. A két párosítás közül a kimeneti listában már csak a legjobb maradjon benne. A mérésben próbálják ki a 0.1 an 0.7 és a 0.9 küszöbértékeket.

Ezt követően a megvalósított függvényt a *SLAM.py* fájlban definiált **SLAM** osztály **addFrame** függvényének megfelelő helyén meg kell hívni.

Előző képjellemzőkkel történő párosítás

```
prevMatch = match(self.prevFeat,features)
```

3.3. Feladat: Implementálja a RANSAC algoritmust, és határozza meg a két képkocka közti merev transzformáció mátrixát!

A feladathoz a *RANSAC.py* fájlban definiált **RANSAC** osztály **___call___** függvényét kell megvalósítani.

A feladat során három fontos változónk lesz: A $numCandidates$ az összesen előállítandó megoldás jelöltek száma, a $numMatches$ a képjellemző párok száma, a $self.N$ pedig azt adja meg, hogy egyetlen jelöltet hány darab pont pár segítségével határozzunk meg.

$numCandidates \times self.N$ darab $[0 - numMatches)$ közti random integer előállítása. Ezeket használhatjuk a képjellemző koordinátákat tartalmazó tömbökbe történő indexelésre.

```
indices = np.random.randint(0,numMatches,(numCandidates,self.N))
```

Egy darab transzformáció jelölt készítése. Az $srcCoordsSelection$ és $dstCoordsSelection$ bemeneti változókat a teljes listákból ($srcCoords$ és $dstCoords$) az előbb előállított random indexek segítségével kell kiválasztani.

```
candidate = self.generateCandidate(srcCoordsSelection,dstCoordsSelection)
```

A kapott *candidate* jelölt lehet **None** bizonyos esetekben, így a jelölteket tartalmazó listából ezeket utólag törölni kell.

Jelölt kiértékelése és inlierek meghatározása. A kimenet egy lista, melynek minden eleme 0 vagy 1 az adott pontpár inlier státuszának függvényében. FONTOS: Míg az egyes jelöltek becsléséhez az előbb csak $self.N$ darab véletlen pontpárt használtunk, a kiértékelésnél azt kell megnézni, hogy az összes pontpárból hányra illik rá az adott jelölt. Az összes jelölthöz tartozó inliers tömböket az $arrayOfInliers$ listába tároljuk el.

```
inliers = self.evalCandidate(candidate,srcCoords,dstCoords)
```

Listában lévő számok összege (bináris lista esetén az egyesek száma)

```
score = sum(inliers)
```

Tömb maximum pozíciójának megkeresése

```
best_i = np.argmax(scores)
```

A legjobb jelölthöz tartozó inlierek. A bool konverzió azért szükséges, mert a NumPy az int tömbbel történő indexelést másképp értelmezi, mint a bool tömb esetén.

```
inliers  = np.array(arrayOfInliers[best_i],dtype='bool')
```

Ezt követően a jelölt állításához használt módszerrel finomítsuk a legjobb jelöltet csak az inlier pontok használatával. FONTOS: Ez megint lehet **None**, ebben az esetben használjuk a finomítás nélküli legjobb jelölt értéket.

Érdemes észrevenni, hogy itt már csak a legtöbb inlierrel rendelkező jelöltet generáljuk újra, nem pedig az összes jelöltet. Éppen ezért ebben a lépésben egyetlen mátrixot hozunk létre, nem pedig egy listát. Ezt azért tesszük meg, mert a jelöltet eredetileg néhány pontpár alapján becsültük, inlierből viszont tipikusan 1-2 nagyságrenddel több van, így a kapott becslés lényegesen pontosabb.

```
mtx = self.generateCandidate(InlierSrcCoords,InlierDstCoords)
```

Ezt követően a megvalósított függvényt a *SLAM.py* fájlban definiált **SLAM** osztály **addFrame** függvényének megfelelő helyén meg kell hívni.

Relatív transzformáció meghatározása

```
trPrev,matchPrev,featPrev = self.RANSAC(self.prevFeat,features,prevMatch)
```

Aboszolút transzformáció meghatározása

```
self .transform = np.matmul(self.transform,trPrev)
```

4 Ellenőrző kérdések

1. Hogyan lehet képeken komplex képjellemzőket detektálni? Hogyan érdemes ezek párosítását elvégezni?
2. Mik a SLAM algoritmus lépései?
3. Mi az a drift és hogyan kerülhető el?
4. Írja fel a Pinhole kamera vetítésének egyenletét!
5. Mi az a RANSAC algoritmus? Milyen lépései vannak?
6. List comprehension írása. Az összes alábbi feladatot egyetlen sorban kell megvalósítani.
   (a) Adott két lista a és b (egyenlő hosszúak). Szorozza össze ezek minden második elemét és tegye ezeket a kimeneti listába.
   (b) Adott két lista a és b (akármilyen hosszúak). A kimeneti lista tartalmazza a $myFunc(a_i)$ értékeket ($a_i \in a$), de csak akkor, ha a_i nincs benne a b listában.
   (c) Adott két lista a és b (egyenlő hosszúak). A kimeneti lista egyes elemeinek értéke legyen $myFunc1(a_i, c_i)$, ahol c_i a $myFunc2(b_i)$ függvény segítségével állítható elő ($a_i \in a \;\; b_i \in b$).
   (d) Adott egy lista a. A kimeneti listába gyűjtse ki a azon értékeit, melyek nem $None$-ok.
7. Numpy tömbök manipulációja. A feladatokat for ciklus/list comprehension használata nélkül kell megoldani.
   (a) Adott egy NumPy tömb X, amely $N \times 3$ méretű, és minden sora egy 3D pont koordinátája. Alakítsa ezeket homogén koordinátákká!
   (b) Adott két NumPy tömb A, amely 4×4 méretű és X, amely $N \times 4$ méretű. Szorozza ezeket össze úgy, hogy kimenetként újra egy $N \times 4$ tömböt kapjunk (a dimenziók sorrendjére tessék figyelni)!

(c) Adott egy NumPy tömb X, amely $N \times 4$ méretű, és minden sora egy 3D pont homogén koordinátás alakja. Konvertálja ezeket vissza euklideszi koordinátákká!

(d) Adott egy NumPy tömb X, amely $N \times 3$ méretű. Generáljon 10 darab random integert a $[0 - N)$ tartományban, és válassza ki a tömbből a random indexeknek megfelelő sorokat.

(e) Adott egy NumPy tömb X, amely $N \times 3$ méretű. Válassza ki azokat a sorokat, ahol az első elem értéke pozitív.

(f) Adott egy ismeretlen méretű NumPy tömb X. Generáljon annyi egyenletes eloszlású véletlen számot, ahány elem van X nulladik dimenziója mentén. Válassza ki ennek segítségével a tömb elemeinek véletlen 50%-át!

Az utolsó két kérdés helyes megoldásához a Google Colab [4] online Python notebook környezetben lehet kísérletezgetni. A beugróban szereplő feladatok a fenti példáktól valamelyest különbözni fognak, a szintaktikai hibákkal szemben persze elnézőek vagyunk!

Bibliography

[1] M. Szemenyei, *Számítógépes Látórendszerek*, M. Szemenyei, Ed. BME, 2019. [Online]. Available: http://deeplearning.iit.bme.hu/jegyzetFull.pdf.

[2] *PyCharm Quick Start Guide*. [Online]. Available: https://www.jetbrains.com/help/pycharm/quick-start-guide.html.

[3] *Anaconda*. [Online]. Available: https://www.anaconda.com/.

[4] *Google Colaboratory*. [Online]. Available: https://colab.research.google.com/.