

Structure from Motion from Two Views

1. fejezet

Algoritmus

A Structure from motion (SfM) folyamat segítségével 3D rekonstrukciót hajthatunk végre egy képpár segítségével.

1. Két kép közötti ritka ponthalmazok megfeleltetése (pontmegfeleltetés): az első kép sarokpontjainak azonosítása a *detectMinEigenFeatures* függvénnyel, majd azok követése a második képre a *vision.PointTracker* segítségével.
2. Az esszenciális mátrix becslése *estimateEssentialMatrix* használatával.
3. Kamera elmozdulásának kiszámítása *estrelpose* függvénnyel.
4. Két kép közötti sűrű ponthalmazok megfeleltetése (pontmegfeleltetés): több pont kinyeréséhez újra kell detektálni a pontokat a *detectMinEigenFeatures* függvény segítségével a *'MinQuality'* opciót használva. Ezt követi a sűrű ponthalmaz követése a második képre a *vision.PointTracker* használatával.
5. Az illeszkedő pontok 3D helyzeteinek meghatározása a *triangulate* segítségével (háromszögelés).

2. fejezet

Kód magyarázata

2.1. Képpár betöltése

1. *fullfile(string1, string2, ...)* = az argumentumként kapott stringekből összeállít egy elérési útvonalat, pl.:

```
path = fullfile('myfolder', 'mysubfolder')  
path = 'myfolder\mysubfolder\'
```

toolboxdir(toolbox) = visszaadja az argumentumként kapott toolbox abszolút elérési útvonalát.

2. *imageDatastore(path)* = létrehoz egy ImageDatastore objektumot a kapott elérési útvonallal meghatározott képekből. Az ImageDatastore objektum segítségével egy mappában található összes képet össze lehet gyűjteni egy változóba (de alapból nem lesz az összes kép egyszerre betöltve).
3. *readimage(datastore, n)* = betölti az n. képet a megadott datastore-ból.
4. *figure* = létrehoz egy új, üres ábra ablakot.
5. *imshowpair(image1, image2, 'montage')* = a meghatározott két képet egymás mellé helyezi a legutolsó ábrán.
6. *title('string')* = hozzáad egy címet a legutolsó ábrához.

2.2. A Camera Calibrator alkalmazás segítségével előre kiszámolt kamera paraméterek betöltése.

1. `load(file_name.mat)` = betölti egy korábban elmentett workspace adatait a jelenlegi workspace-be. A workspace egy ideiglenes tároló amely a MATLAB elindítása óta létrehozott változókat tárolja. Alapértelmezetten a MATLAB ablak jobb oldalán látható. A workspace-t el lehet menteni, így a benne tárolt változókat később vissza lehet tölteni a MATLAB-ba.

2.3. Lencse által okozott torzítás eltávolítása.

1. `undistortImage(image, intrinsics)` = a második argumentumként megadott kamera paramétereket felhasználva eltünteti a kamera lencséje által okozott torzítást a megadott képről.

A kamera kalibrációja során kapott kamera paramétereket és a torzítási együtthatókat felhasználva kiszámítjuk a bemeneti kép minden pixelének eredeti pozícióját. Az egyes pixelek pozícióját az alábbi torzítások módosítják:

- **Radiális torzítás** = kiváltó oka, hogy a lencse szélén áthaladó fény jobban törik, mint a lencse közepén környezetében áthaladó fény. Ez kiszámolható:

$$x_r = x(1 + k_1r^2 + k_2r^4)$$

$$y_r = y(1 + k_1r^2 + k_2r^4)$$

(Ahol x_r, y_r = radiális torzulásmentes koordináták; x, y = torzított koordináták; k_1, k_2 = radiális torzítási együtthatók; $r^2 = x^2 + y^2$)

- **Tangenciális fordítás** = előfordul, ha a kameraszensor és a lencse nem állnak tökéletesen párhuzamosan. Ez kiszámolható:

$$x_t = 2p_1xy + p_2(r^2 + 2x^2)$$

$$y_t = 2p_2xy + p_1(r^2 + 2y^2)$$

(Ahol x_t, y_t = tangenciális torzulásmentes koordináták; x, y = torzított koordináták; p_1, p_2 = tangenciális torzítási együtthatók; $r^2 = x^2 + y^2$)

A végső együttható értékének kiszámítása:

$$x_f = x_r + x_t$$

$$y_f = y_r + y_t$$

Az egyes pixelek korrigált helyének kiszámítása nem egész számú értékeket is előállít. Mivel a nem egész szám nem lehet pixel koordináta, ezért bilineáris interpolációt is végre kell hajtani. A bilineáris interpoláció során, a legközelebbi négy szomszédot felhasználva először lineáris interpolációt hajtunk végre az egyik irányba (pl. az x tengely mentén), majd pedig a másik irányba (az y tengely mentén):

$$out_P = I_1(1-\Delta X)(1-\Delta Y) + I_2(\Delta X)(1-\Delta Y) + I_3(1-\Delta X)(\Delta Y) + I_4(\Delta X)(\Delta Y)$$

(Ahol I_1, I_2, I_3, I_4 = a szomszédos négy koordináta intenzitása az eredeti, torzított képen; $\Delta X, \Delta Y$ = a nem egész értékű koordinátákkal rendelkező vizsgált pixel és a vizsgált pixelhez legközelebb eső, egész értékű koordinátákkal rendelkező szomszédai közötti távolság; out_P = végeredményként kapott pixel intenzitás)

A szomszédos pixelek efféle súlyozott átlagolásával, az interpoláció eredményeképp egy pixel intenzitás értéket kapunk, amely a legközelebbi egész érték koordinátával rendelkező pixel intenzitása lesz.

Az előállított, torzítatlan képen néhány pixel (leginkább a kép szélén) nem rendelkezik megfelelő pixel párral az eredeti, torzított képről (ezek azok a területek, ahol az eredeti képből nincs információ). Ezek a pixelek alapértelmezetten 0 értéket kapnak (feketék lesznek).

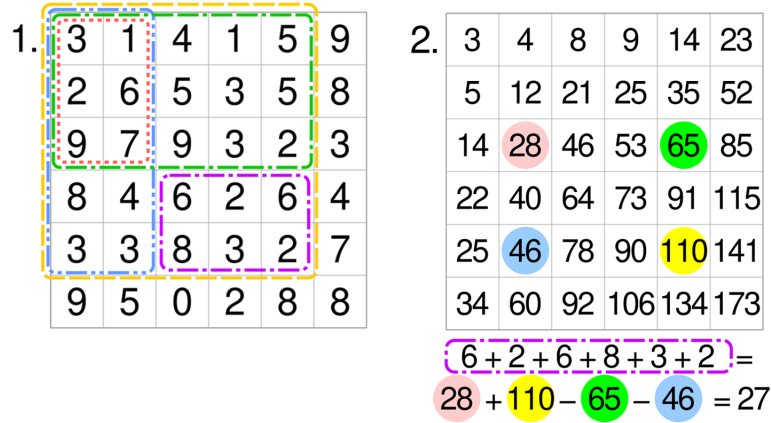
2.4. Pontmegfeleltetés a képek között.

1. *detectSURFFeatures(grayImage, "MetricThreshold", value)* = A Speeded Up Robust Features (SURF) a SIFT algoritmus továbbfejlesztése-

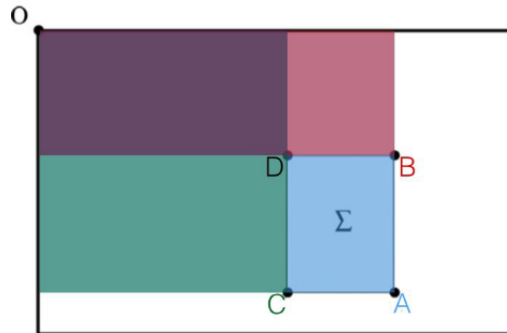
nek is tekinthető.

- (a) **Integrál kép létrehozása** = A számítások gyorsítása érdekében létrehozunk a bementi kép integrál képét. Az integrál kép egy olyan reprezentáció, amely minden pontban a bal felső saroktól addig a pontig lévő intenzitásértékek összegét tárolja:

$$I_{\Sigma}(x, y) = \sum_{i=0}^x \sum_{j=0}^y I(i, j)$$



Az integrál kép felhasználásával gyorsan ki lehet a pixelértékek összegét az adott képen vagy annak egy meghatározott részterületén. Pl.:



$$\Sigma = A - B - C + D$$

A kulcsponthoz megkeresésére a teljesítmény optimalizálása érde-

kében SURF algoritmus a Hess-mátrixot használja. A kulcspontok pozícióját és a skáláját is a Hess mátrix determinánsának közelítésével számoljuk. A Hess determináns approximálásához a SURF a további teljesítménynövelés eléréséhez ún. dobozszűrőt (*box filter*) használ. A doboz szűrő egy olyan szűrő, amely minden, egy előre meghatározott ablakon belüli képpontok értékét a környező pixelek átlagaival helyettesíti. Ennél a folyamatnál nagyon hasznos az integrált kép, ugyanis bármely képpontra meg tudjuk mondani az őt megelőző pixelek értékeinek összegét (gyorsítva a számolást). Ezen lépés után a Hess mátrix determinánsának approximáltja:

$$\det(\mathcal{H}_{approx}) = D_{xx}D_{yy} - (wD_{xy})^2$$

ahol:

- D_{xx}, D_{yy}, D_{xy} = másodrendű képpont deriváltak horizontális, vertikális és keresztirányban
- w = dobozszűrő ablakára vonatkozó súly (általban 0.9)

2. `imshow(image, InitialMagnification = value)` = a meghatározott kép megjelenítése *value*%-os megnagyításban.
3. `hold on` = a következőkben végrehajtott grafikus (pl.: grafikonok kirajzolása, stb.) parancsokat a legutolsó, aktuális ábrára fogja rárajzolni.
4. `plot` = létrehoz kétdimenziós grafikont.
5. `selectStrongest(featurePoints, N)` = visszaadja az N legerősebb (talán ez az intenzitások különbségének nagyságát jelenti) jellemzőpontot (nálunk sarokpontok lesznek) a megadott *featurePoints* változóból.
6. `tracker = vision.PointTracker(MaxBidirectionalError=value1, NumPyramidLevels=value2)` = létrehoz egy Kanade-Lucas-Tomasi (KLT) algoritmus szerint működő pontkövető objektumot.
A KLT algoritmus kifejezetten jól működik olyan objektumok követésére, amely nem változtat alakot, valamint egyedi és részletes textúrával rendelkezik. A KLT algoritmus a Lucas-Kanade (LK) optikai áramlás becslés algoritmuson alapul. Az optikai áramlás az objektumok egy látszólagos/vizuális elmozdulása (nem feltétlenül egyezik meg a valós

elmozdulással). Általános feltételezése, hogy egy objektum megfigyelt pixeleinek intenzitása elmozdulástól függetlenül állandó:

$$I(x, y, t) = I(x + u, y + v, t + 1)$$

Ahol:

- $I(x, y, t) = (x, y)$ pozíción lévő pixel intenzitása t időben
- u = elmozdulás az x tengelyen
- v = elmozdulás az y tengelyen

Ebből kifejezhető az optikai áramlás egyenlete Taylor sort alkalmazva:

$$I(x + u, y + v, t + 1) \approx I(x, y, t) + I_x u + I_y v + I_t$$

$$I(x + u, y + v, t + 1) - I(x, y, t) = I_x u + I_y v + I_t$$

$$0 \approx I_x u + I_y v + I_t$$

Ahol:

- $I_x u$ = az intenzitás változása az x tengelyen.
- $I_y v$ = az intenzitás változása az y tengelyen.
- I_t = az intenzitás idő szerinti változása.

Tehát, ha az u és v elmozdulások helyesen vannak meghatározva, akkor az intenzitások különbsége megközelítőleg 0. Azonban u és v ismeretlenek és meghatározásukat nehezíti a **nyílás/rekesz/apertúra probléma (aperture problem)**, azaz amikor a tényleges mozgást egy kicsi rekeszen keresztül figyelve próbáljuk meghatározni. Egy objektum tényleges elmozdulásának kiszámítása nehéz, ha csak egy kis területet látunk belőle a résen keresztül.

A Lucas-Kanade algoritmus feltételezi, hogy az optikai áramlás (u és v) konstans és a képen textúrázott tárgyak láthatók. A megfigyelt pixel kezdeti intenzitása a , valamint a rés mögött látható tárgy elmozdítása után észlelt pixel intenzitása b . Ezek különbsége ($b - a$), azaz az időbeli intenzitáskülönbség $I_t(x, y)$. A (x, y) pozíción megfigyelt pixel intenzitásváltozás $I_x(x, y)$ az x tengelyen, valamint $I_y(x, y)$ az y tengelyen.

Az x tengelyen történő u mértékű és az y tengelyen történő v mértékű elmozdulás esetén a pixel intenzitása:

$$I_x(x, y)u + I_y(x, y)v = -I_t(x, y)$$

Nem tudom miért negatív az I_t :(

Az algoritmus helyes működéséhez nem elegendő egyetlen pixelt nézni, így ezt ki kell terjeszteni egy pixelszomszédságra. Egy 3x3 szomszédság esetén az egyenlet:

$$I_x(x + \Delta x, y + \Delta y)u + I_y(x + \Delta x, y + \Delta y)v = -I_t(x + \Delta x, y + \Delta y)$$

ahol $\Delta x = -1, 0, 1$ (a három szomszédos pixel az x tengelyen), $\Delta y = -1, 0, 1$ (a három szomszédos pixel az y tengelyen) Tömör formában a képlet:

$$S \begin{bmatrix} u \\ v \end{bmatrix} = \vec{t}$$

Ahol:

- $S = 9 \times 2$ mátrix amely tartalmazza: $[I_x(x + \Delta x, y + \Delta y), I_y(x + \Delta x, y + \Delta y)]$.
- $\vec{t} =$ vektor amely tartalmazza: $-I_t(x + \Delta x, y + \Delta y)$.

Ezen két ismeretlenes egyenlet megoldása a legkisebb négyzetek módszerével történik, amihez megszorozzuk az egyenletet S^T (én sem értem miért):

$$S^T S \begin{bmatrix} u \\ v \end{bmatrix} = S^T \vec{t}$$

Ebből $\begin{bmatrix} u \\ v \end{bmatrix}$ kifejezve:

$$\begin{bmatrix} u \\ v \end{bmatrix} = (S^T S)^{-1} S^T \vec{t}$$

Röviden (ha jól fogtam fel) a Lucas-Kanade optikai áramlás becslés algoritmus a képkockák közötti mozgást próbálja nyomon követni. Ehhez két képkockán kiválaszt egy pixelszomszédságot és azok térbeli elmozdulását elemzi. Ezután kiszámolja a pixelértékek intenzitásváltozását, amelyből próbálja kiszámolni az objektum tényleges elmozdulását.

Gyorsan mozgó vagy nem ritkásan textúrázott objektumok esetén nem megbízható.

A tényleges pontmegfeleltetést a Kanade-Lucas-Tomasi (KLT) algoritmus végzi, amelyből a MATLAB `vision.PointTracker` függvény a pyramid KLT továbbfejlesztett verziót használja. Először minden kiválasztott jellemzőponthoz egy kis ablakot/apertúrát illeszt. A következő lépésben létrehoz egy kép piramist: a kiinduló képből több szintet hoz létre, ahol az alsó szinten található a teljes (eredeti) felbontású kép, a piramis teteje felé haladva a vizsgált kép egyre kisebb felbontású verziója fog szerepelni. Ezután kiszámolja a kiválasztott pixelek elmozdulás vektorait a piramis legfelső szintjétől kezdve, az előbb tárgyalt Lucas-Kanade optikai áramlás becslés algoritmusát felhasználva. A becsült mozgásokat használva finomítja a jellemzőpontok pozícióit a következő alacsonyabban elhelyezkedő, magasabb felbontású piramisszinten. Ezt addig folytatja, amíg el nem éri a legalsó szintet. A piramisos módszer előnye, hogy az alacsonyabb felbontású képeken végzett kezdeti nyomkövetés javítja a teljes nyomkövetési pontosságot és akár nagyobb távolságok követését is lehetővé teszi.

A `vision.PointTracker` függvényhívásnál használt két argumentum:

- *MaxBidirectionalError* = a funkció lényege, hogy az épp vizsgált pontot az első képről követi a második képre, majd ugyanazon pontot követi visszafelé a második képről az első képre. Ezután kiszámolja a vizsgált pont eredeti és a visszakövetés utáni, új pozíciója közötti különbséget (az első képen). Ha ez az érték nagyobb, mint az függvénynek átadott *MaxBidirectionalError* érték, akkor az adott pont érvénytelen. Hatékony módszer a nem megbízhatóan követhető pontok kiküszöbölésére. Az ajánlott érték 0 és 3 pixel között van.
- *NumPyramidLevels* = a képpiramis szintjeinek számát határozza meg. A nagyobb érték lehetővé teszi a nagyobb elmozdulások követését, viszont lassabb futási időt eredményez. Az ajánlott érték 1 és 4 között van.

7. *imagePoints1 = imagePoints1.Location* = a sarokpont objektumok pozícióinak kiolvasása.
8. *initialize(pointTracker, points, image1)* = inicializálja az első argumentumként kapott pontkövető objektumot, inicializálja a második argu-

mentumként kapott követendő pontokat, valamint beállítja az utolsó argumentumként kapott képet a kezdeti képkockának.

9. $[returnValue1, returnValue2] = step(tracker, image2)$ = a *step* függvény indítja el az első argumentumként kapott pontkövető objektumot. A függvény második argumentum a második képkocka, amelyen el kell végeznie a pontkövetést az objektumnak. Az első visszatérési érték a korábban meghatározott jellemzőpontok koordinátái a második képkockán. A második visszatérési érték egy logikai indexvektor, amely azt jelzi, hogy mely pontokat sikerült nyomon követni. A megbízhatóan követhető pontok *true*, míg ellenkező esetben *false* értéket kapnak.
10. $matchedPoints = imagePoints(validIdx, :)$ = a megbízhatóan követhető pontok lementése. Az *imagePoints* egy mátrix, melynek sorai tartalmazzák a jellemzőpontokat, két oszlopoi pedig a jellemzőpontok x és y koordinátáit. A *imagePoints(validIdx, :)* rész egy mátrix címzés, amely visszaadja mátrix azon sorait ahol $validIdx = 1$ vagy *true*, valamint a mátrix összes oszlopát (: jelentése összes index).
11. $showMatchedFeatures(image1, image2, matchedPoints1, matchedPoints2)$ = a pontmegfeleltetés eredményének kirajzolása. A két argumentumként kapott kép színekódolást kap, valamint a másik két, argumentumként kapott jellemzőpontok is kirajzolódnak és egy vonallal lesznek összekötve a hozzájuk tartozó párjukkal.
12. $[returnValue1, returnValue2] = estimateEssentialMatrix(matchedPoints1, matchedPoints2, intrinsics)$ = az esszenciális mátrix becslése egy képpár megfeleltetett pontjaiból.
Az esszenciális mátrix két kamera közötti mozgást kapcsolja össze, megfeleltetett pontpárok közötti geometriai kapcsolatok leírásával. Tehát a különböző szögből készült képekhez használt kamerák relatív helyzetét és orientációját segít meghatározni, **ha** ismertek a kamera paraméterek (amiket a kamera kalibrációjakor kapunk meg).
Az esszenciális mátrix egy 3x3 szerkezetű, homogén, 2-es rangú mátrix, amely az alábbi alapján írja le a két képkocka közötti kapcsolatot (*koplanaritási egyenlet*):

$$x_1^T E x_2 = 0$$

Az x_1 egy adott pont homogén koordinátája az első képen, míg az x_2 ugyanazon pont homogén koordinátája a második képen (mindkét

esetben a kamera koordináta rendszerében, nem pedig pixel koordináta rendszerben):

$$x_1 = \begin{bmatrix} x_1 \\ y_1 \\ 1 \end{bmatrix}, x_2 = \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix}$$

A két képpont kapcsolatát leíró egyenlőség szerint, ha a két képpont ugyanazon pontot írják le, akkor az esszenciális mátrix (E) segítségével kifejezhető a köztük lévő kapcsolat. A képlet szétbontva:

$$\begin{array}{ccccc} [x_1 & y_1 & 1] & \begin{bmatrix} E_{11} & E_{12} & E_{13} \\ E_{21} & E_{22} & E_{23} \\ E_{31} & E_{32} & E_{33} \end{bmatrix} & \begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} = 0 \\ \uparrow & & & \uparrow & \uparrow \\ ismert & & ismeretlen & & ismert \end{array}$$

Ha elvégezzük az ismert elemek szorzatát, majd szétválasztjuk az ismeretlen az ismert részekről:

$$A \begin{bmatrix} E_{11} \\ \vdots \\ E_{33} \end{bmatrix} = 0$$

Ezt szinguláris érték felbontással végezhetjük el, megkapva az esszenciális mátrixot. Ezt a módszert 8 pont algoritmusnak hívják, amely minimum 8 pontpárt és normalizált koordinátákat követel meg. Az esszenciális mátrix két alapvető elemből áll:

$$E = [t]_x R$$

Ahol:

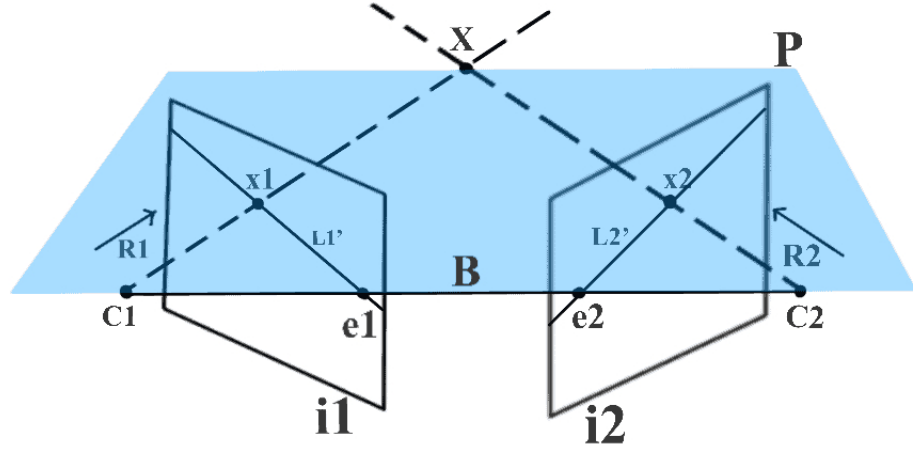
- R = egy ortogonális 3x3-as rotációs mátrix amely a kamerák közötti forgást írja le: $R = \begin{bmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{bmatrix}$
- $[t_x]$ = a t egy translációs vektor, amely a kamera két pozíciójának közötti eltolást írja le: $t = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$. $[t_x]$ = a translációs vektor ferde mátrixa (*skew-symmetric matrix*, én sem tudom mi az), amely

biztosítja, hogy a vektorok közötti keresztszorzat mátrixművelet-

tel leírható legyen: $[t]_x = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix}$

Ez a rész valószínűleg erősen hiányos, de nem teljesen értem, hogy hogyan történik az esszenciális mátrix becslése.

A második visszatérési érték, az *epipolarInliers* megadja, hogy a megfeleltetett pontok mennyire esnek közel az epipoláris síkhoz. (Ha jól értelmeztem) az epipoláris sík magyarázata:



Az X pontról készítünk egy-egy képet két különböző szemszögből (a kamera középpontja az első pozícióban C_1 , míg a második pozícióban lévő kamera középpontja C_2). Az első képen az X pont x_1 pontban, míg a második képen x_2 pontban lett megörökítve. Ha húzunk egy vonalat a C_1 középpontból a képen szereplő x_1 ponton túl, a vonal át fog haladni a lefényképezett X ponton, azaz az első képen x_1 pont a C_1 kameraközéppont és a lefényképezett X pont közé húzott vonalon helyezkedik el (ez a vonal az R_1). Ha ez a vonal látható lenne, akkor a második képen látható részlete lenne a második képhez tartozó epipoláris vonal (L_2). Ahogy az első képen az x_1 az R_1 vonalon helyezkedik el, úgy a második képen az x_2 pontnak is az L_2 epipoláris vonalon kell elhelyezkednie.

A MATLAB *estimateEssentialMatrix* függvény egy *MSAC* elnevezésű algoritmust használ, hogy eldöntse mely pontok outlierok (az epipoláris síktól messze találhatóak) és melyek az inlierek (az epipoláris síkhoz közel találhatóak). Mivel a megfeleltetett képpontoknak az epipoláris síkon kellene elhelyezkedniük, az outlierok rosszul lettek párosítva a pontmegfeleltetés során, így ezeket kiszűrjük.

13. *inlierPoints = matchedPoints(epipolarInliers, :)* = lementsük az epipoláris vonalhoz közel eső (tehát a valószínűleg helyesen párosított) pontokat.
14. *estrelpose(essentialMatrix, intrinsics, inlierPoints1, inlierPoints2)* = kiszámítja a második kamera relatív pozícióját az elsőhöz képest az essenziális mátrixból kinyert rotációs mátrix és translációs vektor alapján. Nem találtam meg, hogy pontosan hogyan teszi ezt :(
15. *camProjection = cameraProjection(intrinsics, tform)* = a visszatérési értéke egy kamera vetítési mátrix. Homogén koordinátával rendelkező, 3D pontok képre való vetítésére szolgál az argumentumként kapott *tform* transzformáció szerint, valamint a kamera belső paramétereit is figyelembe véve.

A vetítési mátrix egy 3x4 szerkezetű mátrix, amely leírja a kamera valóvilágbeli, 3D pontok leképezését 2D pontokra a képsíkra/elkészült képre:

$$x = PX$$

Ahol:

- $x = 2D \text{ képpont} = \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$
- $P = \text{kamera mátrix} = K \begin{bmatrix} R & t \end{bmatrix}$, ahol K = a kamera belső paramétere, R = rotációs mátrix, t = translációs mátrix.
- $X = 3D \text{ pont a valóvilágban} = \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$

Az alkalmazott merev transzformáció (*rigidtransform3d*) egy olyan transzformáció, amelybe beletartozik a forgatás és eltolás, de nem változtatja

meg az objektum méretét és formáját. Ha nem adunk meg semmilyen argumentumot, akkor a *rigidtform3d* egy identitás transzformációt hajt végre, amely semmilyen módon nem változtatja meg az objektumot a leképezés során (forgatást és eltolást is beleértve). Mivel az első kamera a referenciapontunk (tehát az első kamera az origón található), ezért csak egységtranszformációt hajtunk végre. A kamera mátrix kiszámítása a meghatározott transzformációval:

$$P = K [rigidtform3d.R | rigidtform3d.t]$$

A mátrix tartalmazza a képre vetített 3D pontokat homogén koordinátákban.

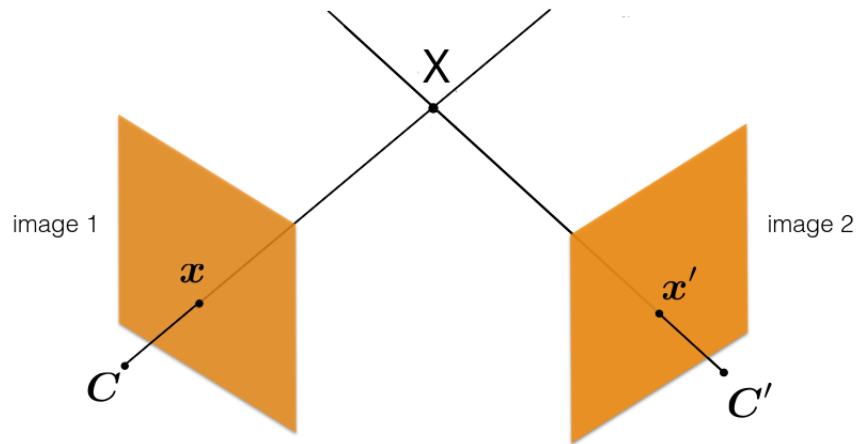
A második kamera esetében az első kamera szerinti relatív elhelyezkedést és orientációt (*relPose*) átadjuk a *pose2extr* függvénynek, amely visszaadja a kamera külső paramétereit (extrinsic mátrix). A kamera külső paramétereit azt mutatják meg, hogyan kell a kamera koordináta-rendszeréből a világ koordináta-rendszerébe végrehajtani a transzformációt. Ez azt jelenti, hogy ha van egy pont a kamera koordináta-rendszerében, akkor a *pose2extr* által visszaadott extrinsic mátrix segítségével átalakíthatjuk ezt a pontot a világ koordináta-rendszerébe. Innentől fogva a *cameraProjection* függvény a belső és külső paramétereket felhasználva leírja a második kamerához tartozó kamera vetítési mátrixot. (Nem tudom mennyire helyes ez a bekezdés).

16. *triangulate(matchedPoints1, matchedPoints2, camProjection1, camProjection2)* = Az argumentumként kapott pontpárok és kamera vetítési mátrixok segítségével háromszögelést végez. Visszatérési értéke a megfeleltetett pontpárok világkoordináta-rendszerbeli koordinátái. Az előző pontokban említettük, hogy a kamera mátrix megadja a 3D pont képsíkra történő 2D leírását:

$$x = PX$$

Mivel ismert az x és a P elméletben ebből az X könnyen meghatározható:

Triangulation



- C = az első kamera középpontja.
- C' = a második kamera középpontja.
- x = az első kamera X objektumról készített leképezése az első képsíkra.
- x' = a második kamera X objektumról készített leképezése a második képsíkra.
- X = az eredeti 3D objektum.

Elméletben tehát ha húzunk egy vetítősugarat az első kamera középpontjából kezdve, át az x ponton, valamint húzunk még egy vetítősugarat a második kamera középpontjától kezdve, át az x' ponton, a két egyenes metszéspontja meg kell határoznia X pontot. A probléma, hogy ehhez tökéletes mérések kellenek, a valóságban viszont a különböző zajok és mérési pontatlanságok miatt a két vetítősugár nem fogja metszeni egymást. Ennek következtében a valóságban csak közelítő eredményt keresünk. A háromszögelés lineáris megoldása:

Szétbontjuk az alábbi egyenletet:

$$x = PX$$

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p_{11} & p_{12} & p_{13} & p_{14} \\ p_{21} & p_{22} & p_{23} & p_{24} \\ p_{31} & p_{32} & p_{33} & p_{34} \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Vesszük a P mátrix sorait és transzponáljuk:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} \text{---} & p_1^T & \text{---} \\ \text{---} & p_2^T & \text{---} \\ \text{---} & p_3^T & \text{---} \end{bmatrix} \begin{bmatrix} | \\ X \\ | \end{bmatrix}$$

A mátrix egyes sorait szorozzuk az X vektorral:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} p_1^T X \\ p_2^T X \\ p_3^T X \end{bmatrix}$$

Vesszük az egyenlőség két oldalának keresztszorzatát:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \times \begin{bmatrix} p_1^T X \\ p_2^T X \\ p_3^T X \end{bmatrix} = 0$$

(Két párhuzamos vektor keresztszorzata 0. Keresztszorzatot az alábbi minta szerint végzünk:

$$a \times b = \begin{bmatrix} a_2 b_3 - a_3 b_2 \\ a_3 b_1 - a_1 b_3 \\ a_1 b_2 - a_2 b_1 \end{bmatrix}$$

)

$$\begin{bmatrix} yp_3^T X - p_2^T X \\ p_1^T X - xp_3^T X \\ xp_2^T X - yp_1^T X \end{bmatrix} = 0$$

A harmadik sor az első két sor kombinációja (az első sor x -szer, és a második sor y -szer), az a sor jelenleg nem jelentős, így azt elhagyjuk.

Az eddigi eljárást megcsináljuk mindkét kamera esetében:

$$\begin{bmatrix} yp_3^T X - p_2^T X \\ p_1^T X - xp_3^T X \end{bmatrix} = 0, \begin{bmatrix} y' p_3'^T X - p_2'^T X \\ p_1'^T X - x' p_3'^T X \end{bmatrix} = 0$$

A két mátrixot összeillesztjük és kiemeljük az X -et:

$$\begin{bmatrix} yp_3^T - p_2^T \\ p_1^T - xp_3^T \\ y'p_3'^T - p_2'^T \\ p_1'^T - x'p_3'^T \end{bmatrix} X = 0$$

A mátrixot egy A -val jelöljük:

$$AX = 0$$

Ez az egyenlőség pedig szinguláris értékek felbontásával (SVD) oldható, megkapva a pont közelítő, 3D koordinátáját.

17. $numPixels = size(var, dim1) * size(var, dim2)$ = a $size$ függvény visszatérési értéke a var változó mérete a meghatározott $dim1$ dimenzióban (1 = sorok száma, 2 = oszlopok száma). A mi példánkban az egyenlettel lekérjük a kép magasságát és szélességét, ezek szorzatával pedig kiszámítsuk a kép pixeleinek számát.
18. $allColors = reshape(I1, [numPixels, 3])$ = a megadott képet átalakítjuk egy tömbbé, amelynek minden sora a kép egy pixele, az oszlopai pedig az adott sorban található pixel RGB értékét adja meg.
19. $colorIdx = sub2ind([size(I1, 1), size(I1, 2)], round(matchedPoints1(:,2)), round(matchedPoints1(:,1)))$ = A megfeleltetett pontok indexeinek kiolvasása. Az indexeket subscript (sor és oszlop indexek) típusból lineáris indexszé (az index egy darab szám, amely az oszlopok mentén haladva a legelső elemtől kezdődik) alakítjuk.
20. $color = allColors(colorIdx, :)$ = kiolvassuk a megfeleltetett pontok színeit az indexvektor segítségével.
21. $pointCloud(xyzPoints, colors)$ = meghatározott színekkel létrehozza a pontfelhő objektumot.
22.

```
figure
plotCamera(Size=cameraSize, Color="r", Label="1", Opacity=0);
hold on
grid on
plotCamera(AbsolutePose=relPose, Size=cameraSize, Color="b", Label="2",
```

$Opacity=0$) = kirajzoljuk az első kamerát az origóba piros színnel, majd pedig a második kamerát kék színnel az első kamera pozíciójától a kiszámolt relatív pozícióba.

3. fejezet

Forrás

- <https://www.mathworks.com/help/vision/ug/structure-from-motion-from-two-views.html>
- https://www.mathworks.com/help/vision/ref/undistortimage.html?s_tid=doc_ta
- <https://www.mathworks.com/help/visionhdl/ug/image-undistort.html>
- https://e-learning.ujs.sk/pluginfile.php/23441/mod_resource/content/1/01-ProjektivKamera.pdf
- <https://www.mathworks.com/help/vision/ref/detectmineigenfeatures.html>
- <https://aishack.in/tutorials/features/>
- <https://aishack.in/tutorials/harris-corner-detector/>
- <https://aishack.in/tutorials/shitomasi-corner-detector/>
- https://docs.opencv.org/3.4/dc/d0d/tutorial_py_features_harris.html
- <https://www.mathworks.com/help/vision/ref/vision.pointtracker-system-object.html>
- <https://lorenzopeppoloni.com/lkttracker/>

- <https://www.baeldung.com/cs/optical-flow-lucas-kanade-method>
- <https://www.inf.u-szeged.hu/~kato/teaching/IpariKepfeldolgozas/08-Motion.pdf>
- http://www.inf.fu-berlin.de/inst/ag-ki/rojas_home/documents/tutorials/Lucas-Kanade2.pdf
- https://link.springer.com/chapter/10.1007/978-3-319-29451-3_29
- <https://www.mathworks.com/help/matlab/ref/step.html>
- <https://www.mathworks.com/help/vision/ref/showmatchedfeatures.html>
- <https://www.baeldung.com/cs/fundamental-matrix-vs-essential-matrix>
- https://e-learning.ujs.sk/pluginfile.php/23450/mod_resource/content/1/05-SztereoKamera.pdf
- <https://learnopencv.com/introduction-to-epipolar-geometry-and-stereo-vision/>
- <https://www.mathworks.com/help/vision/ref/estrelpose.html>
- <https://www.mathworks.com/help/vision/ref/cameraprojection.html>
- <https://ksimek.github.io/2012/08/14/decompose/>
- https://www.cs.cmu.edu/~16385/s17/Slides/11.1_Camera_matrix.pdf
- <https://www.mathworks.com/help/images/ref/rigidtform3d.html>
- <https://www.storyofmathematics.com/rigid-transformation/>
- <https://staff.fnwi.uva.nl/r.vandenboomgaard/ComputerVision/LectureNotes/CV/StereoVision/triangulation.html>
- https://e-learning.ujs.sk/pluginfile.php/23450/mod_resource/content/1/05-SztereoKamera.pdf **SURF**

- https://en.wikipedia.org/wiki/Summed-area_table
- https://www.cs.umd.edu/class/spring2020/cmsc426-0201/files/14_SURF.pdf
- https://link.springer.com/chapter/10.1007/11744023_32
- https://docs.opencv.org/3.4/df/dd2/tutorial_py_surf_intro.html
- <https://medium.com/@deepanshut041/introduction-to-surf-speeded-up-robust-fea>