BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM
GÉPÉSZMÉRNÖKI KAR
MECHATRONIKA, OPTIKA ÉS GÉPÉSZETI INFORMATIKA TANSZÉK

SZEPESSY TAMÁS

SZAKDOLGOZAT

Optikai alapú drónirányítás

Konzulens:
Molnár József
mesteroktató

Témavezető:
Dr. Tamás Péter
c. egyetemi tanár

Budapest, 2019

ABSTRACT

The goal of this thesis was to establish an automatic control system for a UAV, that navigates and collects data only by using the built-in camera. Our program is written in Python, using the OpenCV package for basic computer vision algorithms and ArUco markers for the measurements.

At start, we take a glance at some possibilities on controlling a drone with optical sensors, such as Lidar or Time-of-Flight cameras. On a much more detailed note, we summarize other studies on depth-sensing and mapping with a digital camera: stereovision, neural networks and probability models. As there seems to be a need for robust position tracking of UAVs, AR markers have been chosen for further work.

We chose a DJI Tello quadcopter with Wi-Fi connectivity and a 720p camera. Separate UDP sockets are being used to command the drone, get the video stream or read sensor state values. It can be easily controlled from a computer through various SDK commands, from which the RC control is being adapted for this project.

After the first successful tests, the mathematic model behind the system must be built. With the help of coordinate transformations, one can pass on camera positions between a markers world coordinates and the chosen global coordinate system. Every marker's coordinate system must have an origin point in global coordinates and a rotation, which takes the base to the global bases. The transformations are being done with matrix operations.

With the created program, the quadcopter can navigate itself through a path set by ArUco markers. The control automatically passes on the targeted marker, to which the program's numeric PID controller must position itself. Throughout the automatic navigation, the system collects and stores position data and writes a video of the flight.

The created system has been tested for its accuracy with a motion capture system provided by the faculty. During the 5 test flights, the drone's navigation did not encounter any problem, the system successfully navigated in the environment and measured the UAV's position. The stored data of the two systems have then been compared. Although the AR measurement system has a significant bit of noise, a Kalman filter can smoothen out the movement points in post-processing. The filtered dataset is then similar in its characteristics to the motion capture's reference measurements, with only few centimetres difference.

As conclusion, we can say, that the created system achieved its goal. We adapted an autonomous navigation system for UAVs indoors. Post processed data sets are well filtered enough, to be count as adequate measurements. In the future the program can be advanced to present real-time flight data. We can conclude, that a such system can be used for industrial drones also, where the AR markers' positions are formerly known.

Keywords: *UAV, autonomous navigation, AR marker, positioning, measurements*

TARTALOMJEGYZÉK

JELÖLÉSEK JEGYZÉKE

A táblázatban a többször előforduló jelölések magyar és *angol* nyelvű elnevezése, valamint a fizikai mennyiségek esetén annak mértékegysége található. Az egyes menynyiségek jelölése – ahol lehetséges – megegyezik hazai és a nemzetközi szakirodalomban elfogadott jelölésekkel. A ritkán alkalmazott jelölések magyarázata első előfordulási helyüknél található.

Latin betűk

Jelölés	Megnevezés, megjegyzés, érték	Mértékegység
g	gravitációs gyorsulás (9,81)	$\mathrm{m/s^2}$
$\mathbf{M}$	félkövér nagybetű mátrixot jelöl	SI
$\mathbf{v}$	félkövér kisbetű vektort jelöl	SI

Indexek, kitevők

Jelölés	Megnevezés, értelmezés
0	globális koordináta-rendszer vektorát jelöli
i	általános futóindex (egész szám)
m	m. elemek indexe egy tömbben
n	n. elemek indexe egy tömbben
o	origóba mutató vektort jelöl

1. BEVEZETÉS

1.1. Célkitűzések

A pilóta nélküli repülő járművek (UAV – Unmanned Aerial Vehicle), vagy köznapi elnevezéssel élve drónok, egyre szélesebb körben elterjednek, melynek egyik oka a miniatürizáció és az ezzel létrejövő új kategória: MAV (Micro Aerial Vehicle). Beltéri használatra is alkalmas quadcopterek a kis méretükből adódóan könnyen navigálnak a különböző berendezési tárgyak között. Széleskörű felhasználási területeken - pl. vizuális szemrevételezés, megfigyelés vagy akár filmforgatás -, a kis méret jobb mozgékonyságot és kisebb, elérhetetlenebb helyekre történő berepülést és feltérképezést tesz lehetővé.

A kis méret ára azonban, hogy a készülék kevesebb szenzorral rendelkezik, mivel a tömeg is kritikus faktor. Az autonóm működés során elengedhetetlen akadályelkerüléshez használt sokféle szenzor kiváltására alkalmas lehet egyetlen beépített kamera, megfelelő programmal. Manapság még a kompakt eszközök fedélzeti computerei nem rendelkeznek ekkora számítási kapacitással, ezért egy külső számítógépen tudjuk elvégezni a képfeldolgozást, majd visszaküldeni a vezérlési adatokat.

Jelen szakdolgozat célja, hogy a választott drón önállóan képes legyen egy repülési térképet alkotni a kamerakép alapján, illetve elnavigálni az akadályok között, így kialakítva egy beltéri navigációs rendszert, melyben a drón tájékozódni is képes.

1.2. Áttekintés

A szakdolgozat a feladatkiírás pontjai alapján épül fel. A szakirodalmi áttekintésben kitérve az optikai szenzorok és külön a kamerás képfeldolgozás elvén működő drónvezérlésekre. Ez után a programkörnyezet és a drón kiválasztásának indoklása következik, részletezve a választott drón vezérlési és adatkiolvasási módjait.

A dolgozat fő része a képfeldolgozó és vezérlőprogram működési elvének részletezése, az alkalmazott matematikai módszerek, a kialakított algoritmusok és a vezérlő program struktúrájának magyarázata. A működési teszteket végül egy kalibrált rendszerben is elvégezve, ezekkel a referenciapontokkal az elkészült program méréseit összevetve, a tanulmány egy végkövetkeztetést von és mérlegeli az esetleges továbbfejlesztési lehetőségeket.

2. SZAKIRODALMI ÁTTEKINTÉS/ELŐZMÉNYEK

Az alkalmazott optikai helymeghatározási és navigálási módszer kiválasztása előtt meg kellett vizsgálnunk a különböző lehetőségeket. Olyan módszereket tekintettünk át, amelyek alkalmasak lehetnek egy drón navigációja esetén. Ezek alkalmazhatóságát vizsgáljuk a mikrodrónok esetében.

2.1. Optikai elven működő szenzorok

Különféle szenzorok alkalmasak akadályelkerülésre, ilyenek például a LIDAR [1, 2] (Light Detection and Ranging) lézerfény elvű távérzékelő szenzorok, melyek nagypontosságú, gyors leképezést tesznek lehetővé a környezetről. Bármilyen visszaverő felület távolságáról adatokat kaphatunk vele. A lézerfény-elvű mérőeszközök gyors működésre képesek: a kibocsátott lézerimpulzusok visszaverődésének gyors és szisztematikus egymás után történő figyelésével, a kibocsátott és visszaérkező lézerfény fáziskülönbségének mérésével a tárgy geometriájáról is adatokat nyerhetünk. Ilyen lézerszkennelési eljárást nagyobb drónokban gyakran alkalmaznak. Ezek már a technológia miatt drágább készülékek, inkább az iparban alkalmazzák őket. Az akadályelkerülés és navigáció mellett, a LIDAR szenzorokkal szerelt gépekkel pontos térbeli mérések is végezhetők, nagy kiterjedésű tárgyakról is. A szenzor kiterjedése és a szükséges áramkörök nem teszik lehetővé a mi problémánk során történő alkalmazást, továbbá használata inkább kültéri körülmények között ajánlott.

Olcsóbb megoldást jelenthetnek az ultrahangos és infrafény [3] elvén működő szenzorok, melyek a kibocsátott jelek visszaverődési idejét figyelik. Előbbi esetében a működés lassabb, mivel nem fény, hanem hangterjedés elvén működik és csak egyetlen mélységadatot ad vissza, a tárgy mélységtérképét nem. Ultrahangos szenzorokat költséghatékonyságuk miatt előszeretettel alkalmaznak akadályelkerülésre, azonban ezekből mindenképp többet és több irányban kell felhelyezni a járműre.

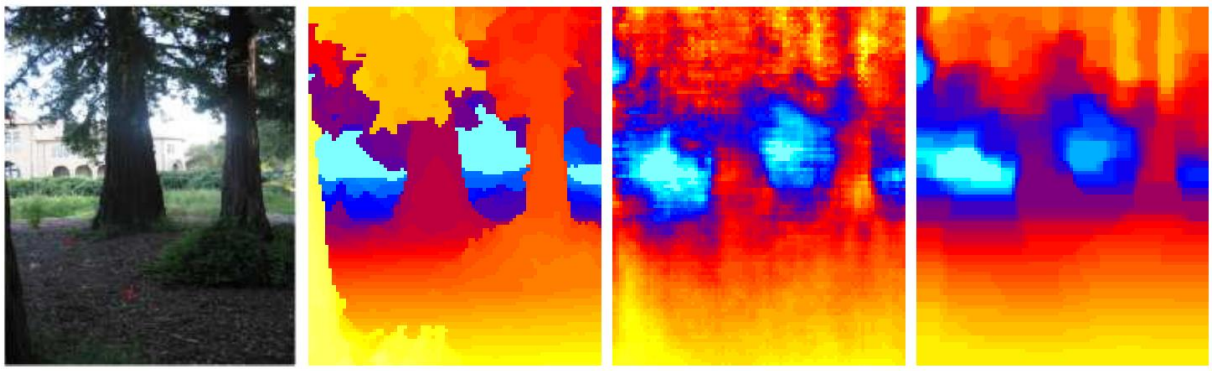
Szintén visszaverődés elvére épülnek a ToF (Time-of-Flight) kamerák [4], melyek pixelenként képesek mélységértékeket rendelni a felvételekhez a kibocsátott infravörös hullámhosszú segédfény pixelenként mért fáziseltéréséből. Ezek a szenzorok megfelelő környezeti körülmények között hasonlóan pontosan és gyorsan képesek működni, mint a LIDAR szenzorok. [5] Míg a LIDAR csak térbeli adatokat ad vissza, egy ToF kamera képes a felületek textúráit is eltárolni (akár egy hétköznapi digitális kamera), mely a későbbi adatfeldolgozást könnyítheti meg. A mélységkamerák beltéren is alkalmazhatók, térbeli felbontásuk a pixeleik számától függ. A szenzor mérete kisebb, mint lézeres elvű társáé, vezérlő áramköre is egyszerűbb, azonban a bevezetésben már említett tömegcsökkentési okból jelen esetben csak a beépített kamera képének feldolgozásával szeretnénk dolgozni.

2.2. Akadályelkerülés és tájékozódás kamerakép alapján

2.2.1. Feltanított neurális hálóval

A megvizsgált kamerakép alapú módszerek között szerepelt a gépi tanulás és neurális hálók alkalmazása [6], melyet az idő- és hardverigényes tanítási folyamatok miatt elvetettünk. A módszer lényege, hogy egy feltanított neurális háló mindössze egyetlen képből a tárgyak elhelyezkedése alapján képes hozzávetőleges mélységértékeket rendelni az egyes pixelekhez. Tanulómintákat egy egymás mellett elhelyezett LIDAR és digitális kamera párosából kaphatunk. Ezen tanulópárokból megfelelő mennyiségűt betáplálva a neurális háló kapcsolatot képes teremteni a képek pixelei és a LIDAR-ból (vagy sztereokamerából) nyert mélységértékek között. A háló tanítása történhet felügyelt, LIDAR-elvű és felügyeletlen, sztereopárokon alapuló tanulással is, egy valószínűségi elv alapján. A felügyelt tanulás sokszor túl szigorú, a felügyeletlen pedig pontatlan eredményeket ad, ezért ajánlott fél-felügyelt tanítást alkalmazni [7] vagy a háló által adott eredményeket működés közben összevetni egy kalibrált mélységérzékelő rendszerrel. [8]

Így az emberi látáshoz hasonlóan, tárgyak (pixelhalmazok) környezetéből és elhelyezkedéséből képes mélységértékeket visszaadni, majd felépíteni egy az *1. ábra* képein láthatóhoz hasonló mélységtérképet. A pontossága egyelőre kísérleti jellegű, egy ilyen rendszer még sok hibát vét, viszont így is próbálták már alkalmazni a drónirányítás területén. [9] Az 1. ábra mélységtérképein látható a különböző módszerek közti különbség és a módszer pontossága.



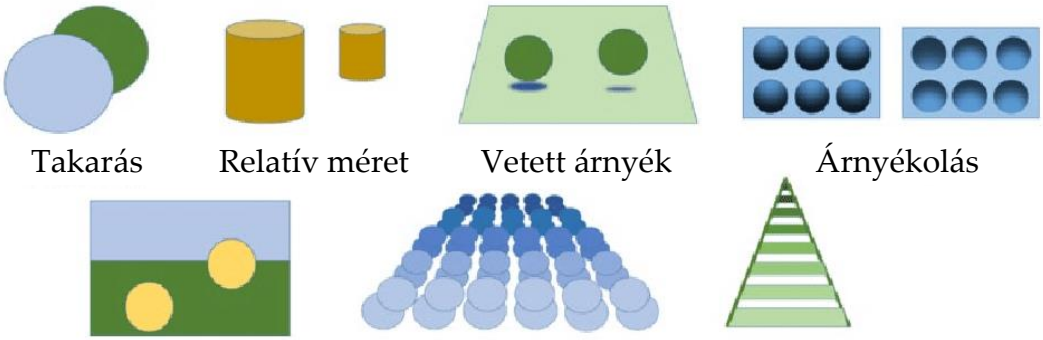
1. ábra: A betáplált kép (balra), az igazi mélységtérkép a szenzorok alapján (bal közép), végül az ún. Gauss-modell (jobb közép) és az ún. Laplace-modell (jobbra) által felépített mélységtérkép [6, 1.Im]

Egy ilyen neurális háló több tényezőt vesz figyelembe. [10] A *2. ábra* elemeinek magyarázata, vagyis a figyelt tényezők a következők:

- **takarás (occlusion):** Egy tárgynak nem látszik az egész kiterjedése, mert egy másik tárgy kitakarja, a kitakart tárgy távolabb van.
- **relatív méret (relative size):** Két tárgy egymáshoz viszonyított mérete eltér, a kisebbik tárgy távolabb helyezkedik el.

- **vetett árnyék (cast shadow):** A tárgy által vetett árnyék viszonyítása a tárgyhoz, ha látható árnyék.
- **árnyékolás (shading):** A tárgy térbeli kiterjedésére lehet következtetni a szélein vetülő árnyékokból. A sötétebb szél görbületre utal, hátrébb van.
- **távolság a látóhatártól (distance to horizon):** A látóhatártól távolabb elhelyezkedő tárgyak közelebb vannak a kamerához.
- **textúra gradiens (texture gradient):** Egy tárgy textúrájának távolabb elhelyezkedő részletei elvesznek. Ez a kamera (ember esetében a szemünk) felbontóképességéből adódik, a részletesebb textúrával rendelkező tárgy valószínűleg közelebb is van.
- **lineáris perspektíva (linear perspective):** Az emberi perspektíva érzékeléshez hasonlóan távolodva egyre kisebbnek érzékelhetők a tárgyak, illetve kiterjedésük is szűkül. A keskenyebb geometria távolabbi elhelyezkedésre utal.

A neurális háló tanítása azonban a nagymennyiségű képadat miatt igen időigényes, nagy teljesítményű grafikus kártyával is hetekbe telne egyetlen gépen feltanítani. A megfelelő mennyiségű tanulóminta beszerzése szintén erőforrásigényes. Probléma, ha már feltanított hálót használunk, a valós időbeni működés kialakítása - ami a drónirányítás során elengedhetetlen -, hogy a kalibrálással gyorsítva a rendszert ne vesszen el túl sok adat, mely ütközéshez vezethet. Lassú repülés során a módszer még képes az akadályokat elkerülni, azonban ezeket nem pontos körvonalú akadályként, csak relatív helyzetét tekintve közelebb elhelyezkedő ponthalmazként kezeli.



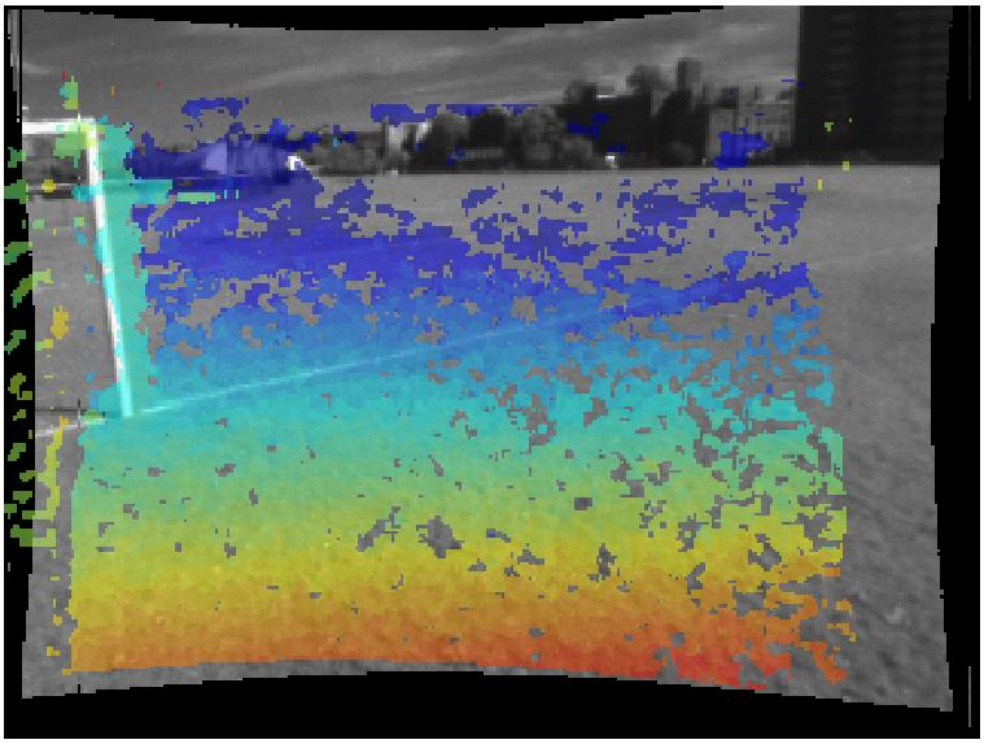
2. ábra: A monokuláris mélységérzékelés által figyelembe vett tényezők [10, 2.Im]

2.2.2. Sztereolátás elvén

Az egyik legrégebb óta alkalmazott kamerás mélységérzékelési technika a sztereopárok elvén alapul, az emberi szem leképezéséhez hasonlóan. [11] Két, egymás mellett rögzített kamerát egyszerre kalibrálhatunk, így kiszámolhatók a kamerák transzformációs mátrixai és a két külön képen, epipoláris egyenesek segítségével azonosított, megegyező képpontokból már képesek vagyunk egy mélységtérképet felépíteni. Az epipoláris egyenes szerepe a képek közötti pontpárok megtalálásának egyszerűsítése

és gyorsítása. A leképezési mátrixok alapján kiszámolhatjuk, hogy például a bal oldali kép egyik képpontja a vetítések után a jobb oldali képen melyik egyenesen fog elhelyezkedni. Így nem kell az egész kép területét átvizsgálnunk, csak egy egyenes mentén a pixeleket. A pixelek hasonlóságának vizsgálatához alkalmazhatunk például Fast Hessian (FH) operátort, mely a Hessian-mátrix determinánsán alapul.

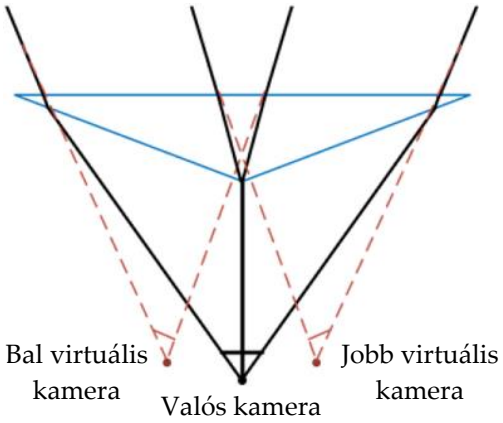
A sztereolátás alkalmazásának előnye, hogy sok szabadon felhasználható megoldás elérhető rá, így az OpenCV programkönyvtárban is. Drónirányítás esetében is alkalmazták már, azonban a megbízhatósága nagy sebesség mellett igen alacsony, ahogy az a *3. ábra* mélységtérképén is látszik. Tökéletes kalibráció és szilárd kamerarögzítés szükséges a mélységtérkép megalkotásához, különben hibás adatokat szolgáltat a rendszer. A pixelek környezetét is felhasználó FH-operátor miatt csak textúrával rendelkező felületeket képes összevetni a technológia. Jelen szakdolgozat során használt drón esetében azonban csak egyetlen kamera áll rendelkezésre, másik felhelyezésére és az eszközhöz csatolására pedig nincs lehetőségünk.



3. ábra: Drónra szerelt sztereopárból alkotott mélységtérkép (vörös - közel / kék - távol) [11, 3.Im]

A korábbiak után további megoldás lehetett volna egykamerás, de prizmával megoldott sztereolátás, mely egyetlen kamera optikáját a prizmával szétosztva valósítja meg a két nézőpontot. [12] Így a rögzített sztereokamerák elvén, diszparitás térkép felépítésével lehetne mélységértékeket kapnunk egyetlen, de két virtuális kamerára osztott kameraképpel, ahogy az a *4. ábra* rajzán is látszik. A prizma geometriai optikai fénytörésének egyenletei segítségével felírható egy transzformációs mátrix a jobb és bal oldali virtuális kamerákra. Ezek után megkapjuk a két virtuális kamera helyzetét, melyből már számolható kalibrált kamerapárhoz hasonlóan kapott sztereopárból, epipoláris geometriák alapján a mélységtérkép.

A mi alkalmazásunkban ezen módszer esetén gondot jelent a prizma szilárd rögzítése a drónon, ugyanis a legkisebb elmozdulás is teljesen elrontja a kalibrálást és használhatatlanná teszi a sztereopárokat. Továbbá így csökken a vízszintes látótér, a kapott kép függőleges kiterjedése hosszabb lesz, mint a vízszintes, mely navigálás során lényeges a drónunk számára.



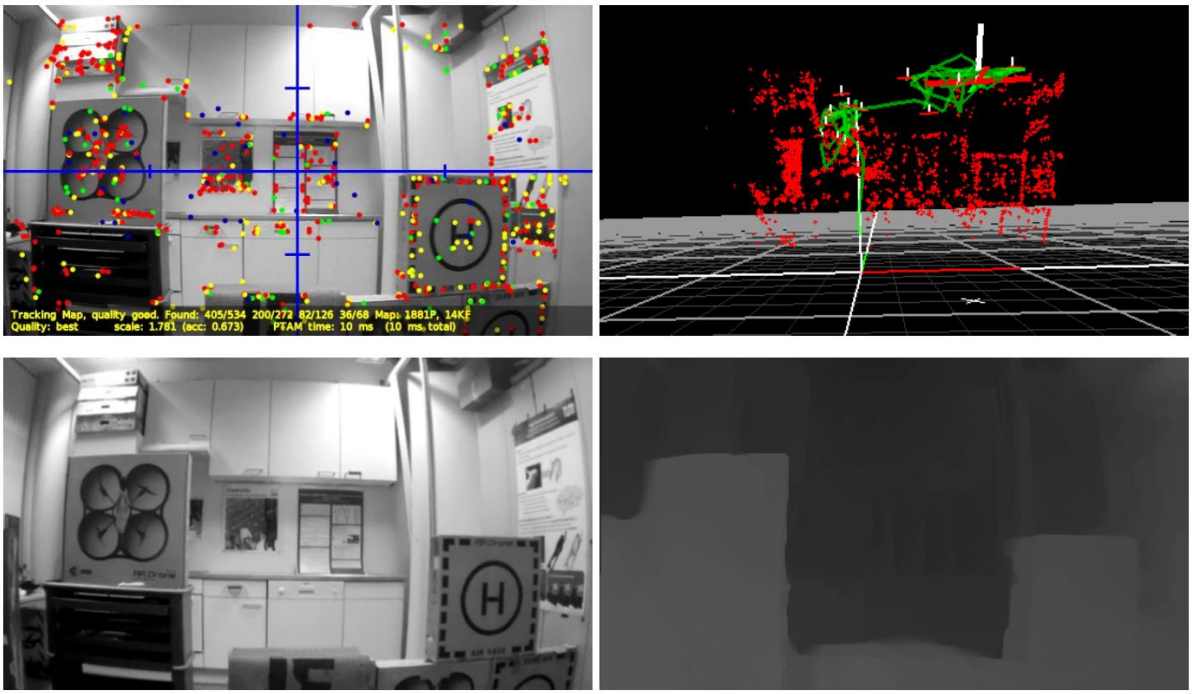
4. ábra: A prizma által kettéosztott kép miatt kialakuló két virtuális kamerakép [4.Im]

2.2.3. Mozgó kamerakép alapján

Sokáig fontolgatott megoldás volt a drón mozgásából számított kamerahelyzetek és az adott időpillanatban készült képekre illesztett epipoláris geometriák alapján alkotott mélységkép. [13] A módszer lényege, hogy ha nincs két kameránk, a drón mozgatásával több szemszögből is lefotózhatunk egy adott tárgyat. A sztereolátástól eltérően itt nem kettő, hanem több tíz képet készítünk és ezeken vetjük össze a képpontokat egymással és a kép készültekori kamerahelyzettel. A képeken fontos a kulcspontok kijelölése - hasonlóan sztereopárok esetén az FH-operátorokhoz-, melyek egy másik képen is biztosan ugyanazt a pontot jelentik. Így a több nézőpont és a kamera helyzete alapján kaphatunk egy mélységképet. (*5. ábra*)

Ami az *Alvarez et al.* féle tanulmány [13] alapján meglepő volt, hogy a drón lebegés közbeni pozícionálási hibái elég elmozdulást biztosítanak egy mélységtérkép megalkotásához. 30 kép már elegendő ahhoz, hogy a szükséges adatokat kinyerhessük belőle. Amennyiben a gyűjtött képek feldolgozása grafikus kártyán történik, a folyamat lényegesen gyorsabb, 30 FPS mellett 1 másodperc a képgyűjtés ideje, míg a feldolgozás 0,5 másodpercet vett igénybe. Összesen 1,5 másodperc alatt számol ki a rendszer egy 64 rétegű mélységtérképet. Ez az idő egyelőre a drónt szakaszonkénti haladásra készteti, ami azonban egy beltéri, alacsony sebességű alkalmazás során még elfogadható.

A rendszer pontosságáról végzett tesztek alapján itt is elmondható, hogy csak jól textúrázott, elkülöníthető felületek esetén ad megfelelő mélységértékeket. Előre beállított környezet esetén (amilyet az 5. ábra is mutat) nagy biztonsággal (90-100%) volt képes a drón az akadályok között elnavigálni. Egy ajtón való átrepülés, a nagyobb, simább felületek miatt gondot okozott: csak 50%-os hatásfokkal teljesítette az akadályt.



5. ábra: Felső sor: A mozgás során követett képpontok és azokból alkotott mélységtérkép
Alsó sor: Az eredeti kamerakép és a mozgás alapján nyert mélységtérkép [13, 5.Im]

Mindehhez azonban szükség van a kamera helyzetének és orientációjának ismeretére, melyek meghatározására tett próbálkozásokat a *3.4 fejezet* taglalja valamivel részletesebben. Jelen esetben a gyorsulásszenzorok zaja és az elintegrálódó hibák miatt elvetettük ezt az ötletet is.

2.2.4. Helyzetmeghatározás markerek segítségével

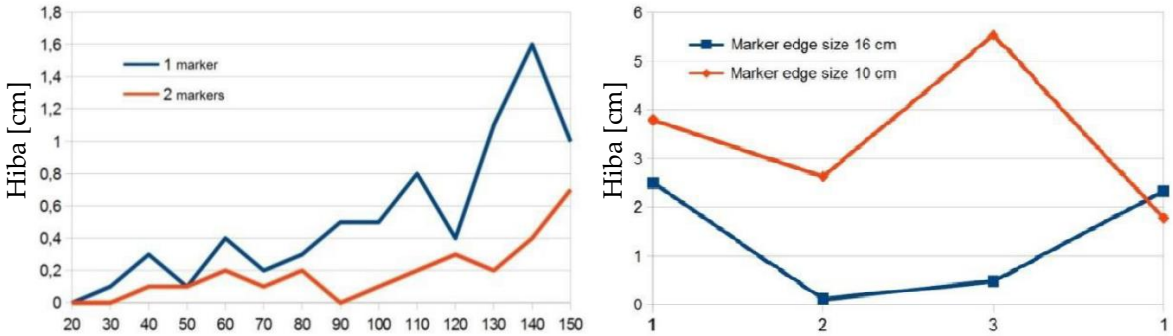
Az *2.2.3 fejezetben* taglaltak alapján fogalmazódott meg a célkitűzés egy „low-cost” (kis költségű) eszközökkel történő, kamerakép alapú helymeghatározásra. Ez alapján a drón pozíciója követhető lenne, ahogy a szögelfordulás és a kamerahelyzet is, mivel ezek a gyorsulásszenzor adataiból csak igen pontatlanul kaphatók meg. Így esett a választás végül az OpenCV nyílt forráskódú programozási eszköztár kiterjesztett valóságot alkalmazó ArUco markereire. Ezeket is alkalmazták már drónirányítás során [14, 15], segítségükkel pozíciót és szögelfordulást nyerhetünk a kamera terében, továbbá a markerek sorszámával vezérlési adatokat is közölhetünk a drónnal, hogy az akár autonóm közlekedésre is képes legyen egy megfelelően felmarkerezett térben.

Egy hallgatótársammal korábbi egyetemi projektünk során már alkalmaztunk ArUco markereket távolságérzékelésre. [16] Akkor sikeresen megállapítottuk, hogy az OpenCV ArUco könyvtárában foglalt függvények változatos körülmények között alkalmazhatók markerfelismerésre, akár homályos, gyenge fényviszonyok mellett is. A mért markerhelyzetek pontosságáról azt állapítottuk meg, hogy kis oldalméretű (0,03 m) markerek esetén is széles tartományban vagyunk képesek érzékelni a markereket átlagban ±0,05 méter pontossággal.

A projektünk során megállapítottuk továbbá, hogy a használt kamera specifikációi is meghatározók lehetnek a markerekkel történő helyzetmeghatározás során. Így a nagylátószögű kamerák a nagyobb lencsetorzítás miatt pontatlanabb méréseket eredményeznek, ahogy az autofókuszos kamerák a folyamatos fókusz-utánállítás során bizonyos mértékben elronthatják a kamera kalibrációs mátrixát. Továbbá a nagyobb kamerafelbontás az eddigiek betartása esetén értelemszerűen kisebb hibákat produkál, mivel a nagyobb pixelszám ekkor kisebb mérési egységeket jelent.

Autonóm robotok pozíciójának meghatározására és navigációjára gyakran alkalmaznak markereket. A rögzített jelzőkhöz képesti elmozdulások Descartes-térben végzett geometriai transzformációk segítségével meghatározhatók, illetve a markerek elfordulásából akár még a kamera forgása is számítható. Mivel az ArUco markerek felismerése megfelelően gyorsan történik, a rendszer valós idejű működésre képes.

Korábbi tanulmányok során [17] egy, autonóm közlekedésre képes robot markerek segítségével kapott kamerahelyzetét vizsgálták. A robot képes volt navigálni a markerek által határolt térben és átszámolni pozícióját a marker teréből a globális térbe. A módszer pontossága függött a látott markerek számától és a marker oldalhosszától. Több marker esetén a kiolvasott pozíciók átlagolásával pontosabb adatok nyerhetők, ahogy a nagyobb markerek alkalmazása esetén is jobb mérések végezhetők.



6. ábra: A markerekből nyert adatok pontossága. a) A pontosság a kamera marker(ek)től vett távolságának függvényében 1 és 2 marker használata esetén. b) A robot egyes ismert pozícióiban a mérési eltérések különböző markeroldalhosszak (10 cm és 16 cm) esetén. [17, 6.Im]

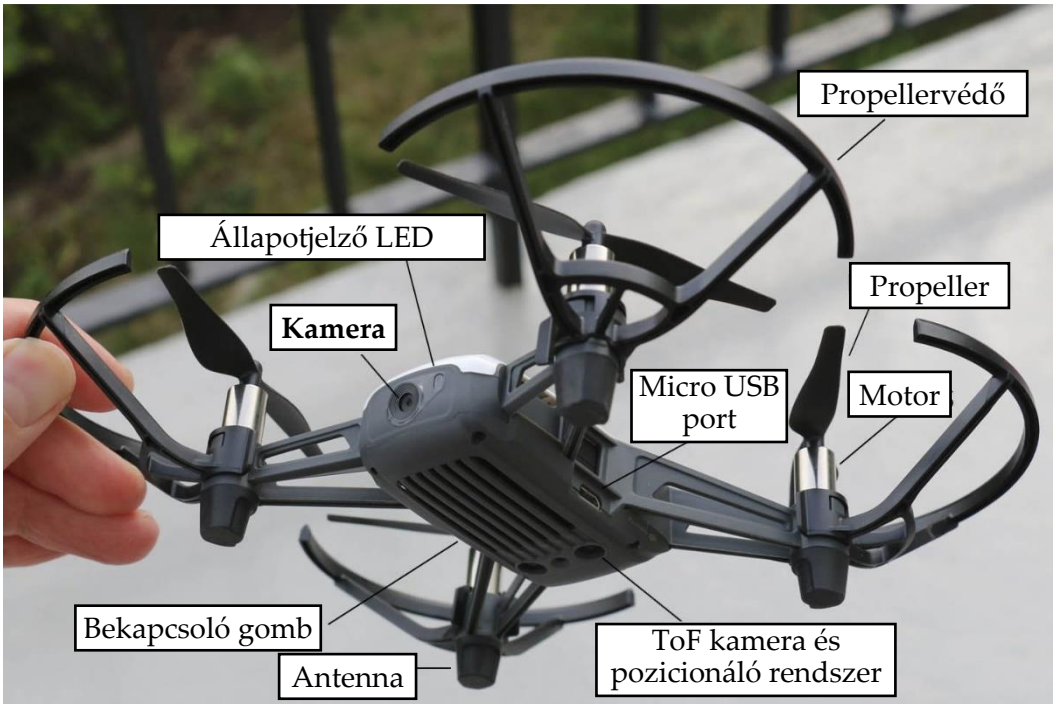
Az említett eredmények figyelembevétele szükséges egy saját rendszer megalkotásához, melyben egy drón önmaga képes navigálni a markerek között, miközben adatokat gyűjt a mozgásáról. A rendszertervezés során vegyük elfogadottnak a *Babinec et al.* féle tanulmányban [17] foglaltakat, a markerek tulajdonságainak kiválasztását is erre alapozva!

3. KÍSÉRLETI KIALAKÍTÁS

3.1. Drón választása

A lehetséges programozható drónok funkcióinak megvizsgálása után a választott eszköz a Ryze cég által fejlesztett, de DJI által támogatott *Tello* quadcopter. Ez Intel processzorral és egy beépített 5 MP felbontású kamerával rendelkezik, wifin, UDP kapcsolaton keresztül képes kommunikálni, tiszta repülési ideje kb. 13 perc és szélmentes környezetben képes pontos pozicionálásra, stabil, egyhelyben történő lebegésre. Az eszköz propellervédőkkel és behelyezett akkumulátorral mindössze 89 grammot nyom, ebből is látszik, hogy a tömeg kritikus tényező.

A pozíciószabályozás és magasságtartás több szenzorral történik. Egy beépített barométer magasságadatokat szolgáltat a drónnak, ahogy az eszköz aljába szerelt infraszenzor és ToF kamera is, ami szintén a síkbeli pozíció tartását segíti. (*7. ábra*) Nélkülözhetetlen továbbá egy IMU szenzor, mely háromirányú gyorsulás és szögelfordulási adatokat szolgáltat.



7. ábra: A DJI Tello drón képe, jelölve az egyes alkatrészeit [7.Im]

A drónhoz elérhető egy *Python* programozási nyelven megírt SDK, mely az UDP kommunikáció részleteit tartalmazza. A drón saját wifi hálózatára rácsatlakozva vezérelhetjük azt különböző parancsokkal, és ki is olvashatunk belőle adatokat. AZ UDP kapcsolat maximális csomagmérete kisebb, mint a kamerakép átvitele, ezért azt több csomagban sugározza a drón.

3.2. Programnyelv és eszközkészlet

A drónhoz szolgáltatott SDK-ból adódóan a Python programnyelv használata szükséges, azon túl pedig *Damiá Fuentes* szabadon felhasználható GitHub könyvtára [16] volt a kiindulási alap. A képfeldolgozás alapja a nyílt forráskódú *OpenCV* (Open Computer Vision) modulcsomag 4.1.0-ás verziója. Továbbá a felhasználói felület a *pyGame*, a mérési adatok kirajzolása a *Matplotlib* csomag alkalmazásával készült.

A magasabb szintű programnyelv miatt, a működés sajnos lassabb lesz, mintha C++-ban dolgoznánk. Ez kritikus tényező akadályelkerülésnél, azonban most a tesztjeinkhez előre beállított, statikus környezetben dolgozunk, így dinamikusan előkerülő akadályok kerülésére nem kell számítani.

3.3. A Tello drón irányításának módja

A gyártó által szolgáltatott dokumentációban részletezve van, miként lehetséges a drónt egy Python program segítségével irányítani. [19] A drón saját wifi hálózatára csatlakozva egy UDP kapcsolatot kell megnyitni az irányításhoz a 192.168.10.1 címen és 8889 porton keresztül. Az irányítás megkezdése előtt mindig egy *„command"* utasítást kell elküldeni a drónnak, ami ezzel kapcsol olyan állapotba, melyben várja a parancsokat.

A drónból továbbá ki lehet olvasni a repülésvezérlő állapotát, melyeket az eszköz firmware-je is használ, a 0.0.0.0 című szervert megnyitva a 8890-es porton. Ez a már említett GitHub könyvtárban nem szerepelt, ezért kiegészítettük vele a programot. Új UDP socket kapcsolatot nyitottunk a már említett szerver és port adatokkal, mely külön szálon fut, hasonlóan a drón vezérlési válaszainak háttérbeni fogadásával.

1. kódrészlet: Az állapotkiolvasó szál megvalósítása (*tello.py/class Tello/def __init__()*)

```
# A drón vezérlő utasításokra adott válaszait figyeli
thread = threading.Thread(target=self.run_udp_receiver, args=())
thread.daemon = True
thread.start()

# A drón állapotait kiolvasó szerver szála
thread_st = threading.Thread(target=self.get_read_state, args=())
thread_st.daemon = True
thread_st.start()
```

Az *1. kódrészlet* mutatja a Python *Threading* könyvtárában foglalt szálkezelést, melyet egy adott függvényhez lehet hozzárendelni, az fog a főprogramtól külön, önálló szálon futni. Az állapotolvasó szál a Tello osztály *get_read_state()* függvényére mutat, azt indítja el külön szálon, így a háttérben fut. A `daemon = True` beállítást érdemes bekapcsolni, ekkor a szál daemonként fog elindulni, azaz ha a nála magasabb szintű program kilép, a daemon szálak is kilépnek. Ha ezt nem kapcsoljuk be, akkor külön kell lekezelni a szálakból történő egyenkénti kilépést.

Az állapotokat a drón a már említett szerveren sugározza, ezeknek olvasása nem fogja lassítani a drónból történő kamerakép-kiolvasást, mivel az állapotok stringjét a drón a *command* módba lépés után akkor is sugározza, ha mi nem olvassuk azt. A hivatalos dokumentációban [19] szerepel, hogy milyen formátumban küldi ki a drón a szenzorainak állapotát. Egyetlen, összefüggő, pontosvesszővel tagolt szöveges ASCII állományt sugároz a vezetéknélküli UDP kapcsolaton keresztül a fogadó szerver felé, melynek formátuma a következő:

```
"pitch:%d;roll:%d;yaw:%d;vgx:%d;vgy:%d;vgz:%d;templ:%d;temph:%d;tof:%d;h:%d;
bat:%d;baro:%.2f;time:%d;agx:%.2f;agy:%.2f;agz:%.2f;\r\n"
```

Ahol az egyes állapotértékek (a koordináták értelmezése a *8. ábra* alapján értendő):

- *pitch, roll, yaw:* a drón x, y és z tengely körüli (intrinsic) elfordulása egész szögben mérve, óramutató járásával megegyezően pozitív, balkéz-szabály szerint
- *vgx, vgy, vgz:* a drón x, y, z irányokban beállított sebessége cm/s-ban (Nem a ténylegesen mérhető sebességértékek, hanem a vezérlés során a drón sebesség változójára kapcsolt érték.)
- *templ:* a legalacsonyabb hőmérséklet C°-ban
- *temph:* a legmagasabb hőmérséklet C°-ban
- *tof:* a drón alján elhelyezkedő Time-of-Flight kamera magasságértéke cm-ben
- *h:* magasság cm-ben
- *bat:* a drón fennmaradó akkukapacitása egész százalékban
- *baro:* magasságérték a beépített barométer alapján cm-ben
- *time:* motorok bekapcsolva töltött ideje (repülési idő) másodpercben
- *agx, agy, agz:* a drón x, y, z irányú gyorsulása a gyorsulásszenzorból 0.001g-ben értelmezve, ahol g a nehézségi gyorsulás

Ezt a pontosvesszők mentén tömbbé szétválaszthatjuk a Python *string.split(";")* függvényével, majd a tömb elemeiből kettőspont utáni értékeket megfelelő formátumú numerikus típussá konvertáljuk. Csak a szükséges értékeket konvertáljuk számmá, majd az értékeket egy listába töltjük, melyet átadunk egy sornak. A Python szálkezelésében csak a *Queue* típus képes biztonságos, szálak közti kommunikációra, ha ezt nem alkalmazzuk, a program szálai közötti adatcserére, a program kifagyhat. Az állapotkiolvasó függvényt mutatja a *2. kódrészlet*. Az Euler-szögek esetén a jobbkéz-szabályba való áttérés miatt kell negatív előjel.

További lehetőségünk adódik az állapotok kiolvasására a vezérlés során is, ekkor egy kérdőszót elküldve a drónnak parancsként, melyre az UDP-kapcsolaton keresztül reagál. Ezek a kérdőszavak ugyanolyan szenzorállapotokat adnak meg, mint az állapotkiolvasás (pl.: *"acceleration?"*, *"speed?"*, *"battery?"*, *"tof?"* …). Ennek a funkciónak a használatát is megfontoltuk, azonban a tesztek alapján ez a módszer kisebb biztonsággal ad megfelelő eredményt. Ez a kommunikáció, az állapotkiolvasással ellentétben lassítja és terheli a drón további funkcióit, mivel az ezekre történő reagáláshoz külön processzoridőt kell biztosítania. Így, mivel szálkezeléssel sem kapunk biztos eredményeket, ez a funkció a mi céljainkra nem használható.

2. kódrészlet: Az drón állapotait kiolvasó függvény (*tello.py/class Tello/def get_read_state()*)

```
def get_read_state(self):
    while True:
        time.sleep(1/25)
        try:
            state_temp, _ = self.stateSocket.recvfrom(1024)
            self.state = state_temp.decode('ASCII').split(";")
            pitch = -int(self.state[0][self.state[0].index(":")+1:])
            roll = -int(self.state[1][self.state[1].index(":")+1:])
            yaw = -int(self.state[2][self.state[2].index(":")+1:])
            tof = int(self.state[10][self.state[8].index(":")+1:])
            bat = int(self.state[10][self.state[10].index(":")+1:])
            self.data_queue.put([pitch, roll, yaw, tof, bat])
        except Exception as e:
            self.LOGGER.error(e)
            break
```

A kamerakép, ami a feladat során elengedhetetlen, szintén UDP-n keresztül kerül sugárzásra, megnyitva a 0.0.0.0 című szervert az 11111 porton keresztül. Mivel a kép kódolt adatainak mérete nagyobb, mint egy UDP-átvitel maximális payload mérete, a képadatokat 1460 bájtos blokkokra osztva sugározza a drón, melyekből az utolsó blokk mérete kisebb mint 1460 bájt. [20] A sugárzott videót a fogadás után az OpenCV csomag segítségével jeleníthetjük meg. A hivatalos DJI-SDK-dokumentációban [20] szereplő *ffmpeg* és *libh264decoder* kodekcsomagok jelen esetben így nem szükségesek, *Microsoft Windows* operációs rendszer alatt tesztelve a programot.

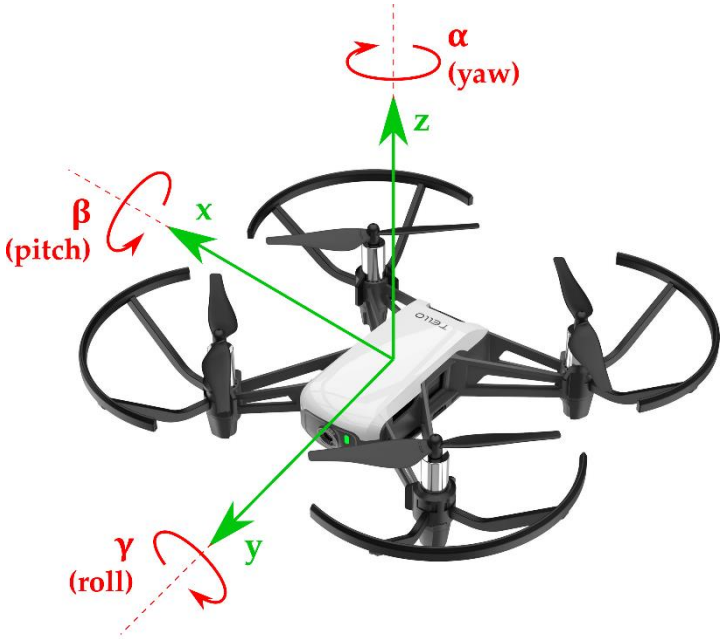
Az SDK vezérlő utasításai három csoportra oszthatók:

1. irányító utasítások
    a. *„ok”*-t ad vissza, ha teljesült
    b. *„error”*-t vagy hibaüzenetet, ha nem
2. kiolvasó utasítások
    a. visszaadja a kért paraméter értékét
3. paraméterek beállításával történő irányítás
    a. *„ok”*-t ad vissza, ha teljesült
    b. *„error”*-t vagy hibaüzenetet, ha nem

A drónnak a *„takeoff”* utasítást elküldve felszáll, *„land”*-ra leszáll, *„streamon”* és *„streamoff”* utasításokkal a videó sugárzása kapcsolható. A drónt magát lehet irányonként és egyszerre, egy *„go x y z speed”* utasítással vezérelni. Ez esetben a drón adott helyzetéhez képest mozdul el a megadott sebességgel a három koordinátával megadott irányba. Ennek a vezérlésnek a pontossága a drón méreteivel egyezik meg, így 20 centiméternél kisebb elmozdulásra nem képes.

A firmware továbbá lehetőséget nyújt ívben történő irányításra is: a drón aktuális pozíciójához képest még két pontot megadva x, y, z koordinátákkal, leírja az általuk meghatározott ívet, amennyiben az ív sugara 0,5 és 10 méter között van.

A drón koordinátáinak értelmezéséhez készült a 8. ábra. A továbbiakban nem fogunk már erre az ábrára visszautalni, amennyiben a drón irányait említjük, az eszerint definiált irányokat értjük.



8. ábra: A drón SDK-val történő vezérlése során meghatározott irányok

Jelen feladat során a drónnal végzett tesztek alapján a legmegfelelőbb az ún. RC irányítás, mely a vezérlési módok 3. csoportjába tartozik. A drón *x*, *y*, *z* és *yaw* irányban értendő sebességeit lehet beállítani az *„rc x y z yaw"* paranccsal -100 és 100 közötti integer értékeket megadva. (A *yaw* sebesség itt szintén balsodrású koordináta-rendszer szerint értendő.) Így akár egy görbét is leírathatunk a parancsok adott frekvenciával történő küldésével, amennyiben egy paraméteres görbe sebességértékeit ugyanilyen frekvenciával mintavételezve adjuk be vezérlésként. Ezzel a vezérlési móddal kisebb elmozdulásokat is elérhetünk, mint a már említett *„go x y z speed"* utasításokkal.

Az utasítások 2. csoportjában szerepelnek a szenzoradatokat lekérdező utasítások. Így például az aktuálisan beállított sebesség, akkumulátor-töltöttség, repülési idő, magasság, gyorsulás és szögelfordulás adatok is kiolvashatók a drón fedélzeti kontrolleréből. Ezeket a korábban már említett megbízhatatlan kiolvashatóság miatt nem alkalmazzuk.

3.4. A drón pozíciómeghatározása

A gyorsulásadatokat lehetett volna helyzetmeghatározáshoz használni, azonban a kiolvasás az *„acceleration?"* parancs elküldése után igen bizonytalanul ad választ. A tesztek alatt beállított 0,1 másodperces időközökkel elküldött kérésre a kapcsolat *timeout* idejének változtatása után is volt, hogy csak 0,5 másodperc után adott értelmezhető választ, de sokszor csak timeout (időtúllépési) hibát dobott. Így a gyorsulásszenzor eleve zajos adatai még egy jelentős mintavételezési zajjal is terhelődtek.

A másik lehetőség, ha külön szálon futó szervert nyitunk a korábban már említett állapotadatok fogadására. Ez esetben egyetlen stringben kapunk adatokat, köztük a gyorsulásszenzor és a giroszkóp adatait is. Ezen adatokat először egy kiegészítő szűrőn [22] vagy Kálmán-szűrőn [23] lenne szükséges átfuttatni, hogy a szenzorzajt eltüntessük. A gyorsulásértékek kétszeri integrálásával kaphatjuk meg a drón elmozdulását, azonban az integrálás művelete a zajokat felerősíti, azok integrálási hibával terheltek és biztos helyzetpontok hiányában ezek korrekciójára sincs lehetőség. Ezt a lehetőséget az integrálással halmozódó zajok miatt elvetettük.

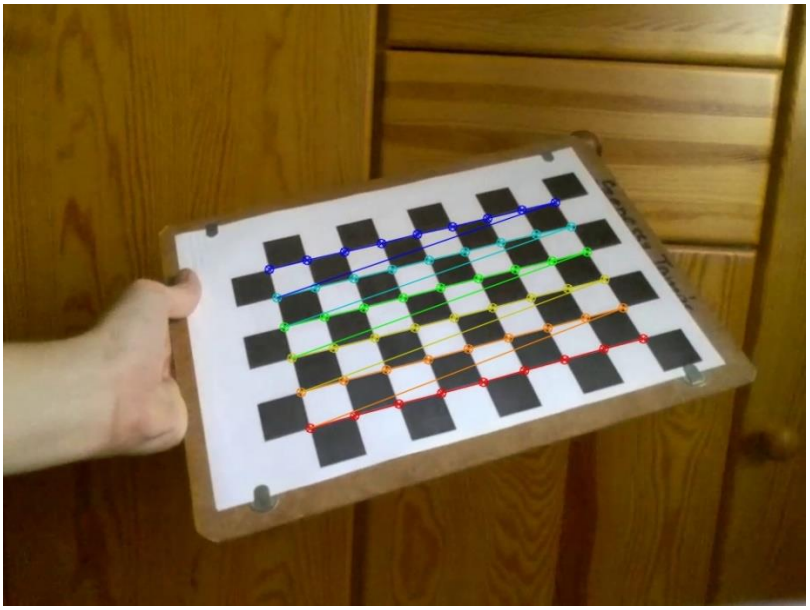
Ezért esett a választás az ArUco markerekre, melyekhez képest mérni lehet a kamera relatív elmozdulását. Ezekkel gondot jelenthet, hogy csak akkor nyerünk helyzetadatokat, amikor a kameraképben felismerhetően megjelenik egy marker. Ez a probléma azonban a teresztterep megfelelő felmarkerezésével megoldható. Megfelelő szabályok alapján - melyeket a *3.4.2 fejezetben* fogalmaztunk meg - kell felhelyezni a markereket és így a kamerahelyzet ezek alapján meghatározható.

3.4.1. Az ArUco markerek alkalmazása

Az ArUco markerek generálásához szükséges függvények megtalálhatók az *OpenCV-Contrib*, kiegészítő modulokat tartalmazó csomagban. A helyzetmeghatározásukhoz szükség van a kamera kalibrációjára, mely szintén elvégezhető az OpenCV-ben megtalálható kalibrációs algoritmussal, amely *Z. Zhang* [24] tanulmányán alapszik, megbecsülve a kamera paramétereit.

A kalibráláshoz szükség van egy sakktáblára, mely négyzeteinek oldalhosszait, sor és oszlopszámát kell megadni. Ezek alapján az algoritmus elkészíti négyzetek belső sarokpontjainak elméleti térképét, majd a kamerával készített minták alapján kiszámolja a kamera belső és külső tulajdonságait, transzformációs mátrixokban. A kameramátrix tartalmazza a kamera öt belső (*intrinsic*) tulajdonságát, köztük a fókusztávolságot, a képszenzor oldalarányát és a pinhole kameramodell főpontját. A nemlineáris belső tulajdonságokhoz tartozó lencsetorzítás disztorziós mátrixát is megkapjuk, ami alapján a kép szélein fellépő hordó- és párnatorzítás megszüntethető. A kamera külső (*extrinsic*) tulajdonságait szintén egy transzformációs mátrix írja le, megvalósítva a valós 3D tér koordinátáiból a kamera 3D koordinátáiba történő átvitelt. A mátrix homogén transzformációt alkalmaz: a **T** vektorral történő eltolásra a kameraszenzor középpontjába (origó) és az **R** forgatási mátrixszal történő forgatásra.

A helyes kalibráláshoz elméletben már két minta is elég lenne, azonban a pontosabb adatok elérése érdekében 20 adatból dolgozunk. A kalibrálási folyamat során készült a *9. ábra* fotója. Az így kapott mátrixokat egy *camcalib.npz* nevű fájlban tároljuk és onnan olvassuk be, hogy ne kelljen minden használat előtt kalibrálni a drón beépített kameráját.



9. ábra: Fotó a kamera kalibrálásának folyamatáról a sakktáblára vetített objektumpontokkal

Az egyik leggyakoribb hibaforrás a kamerával végzett mérések során a hordótorzítás, mely a kalibrálás során kapott disztorziós mátrix segítségével megszüntethető az OpenCV beépített *cv2.undistort()* függvényével. Ennek hatására csökken a halszem-effektus és bizonyos mértékben csökken a látómező, azonban így a későbbi, ArUco markerekkel végzett helyzetmeghatározás során a kép szélein pontosabb értékeket kaphatunk.

3.4.2. A MARKERFELHELYEZÉS SZABÁLYAI

A szabályokat következőképpen határozzuk meg a markerfelhelyezéshez:

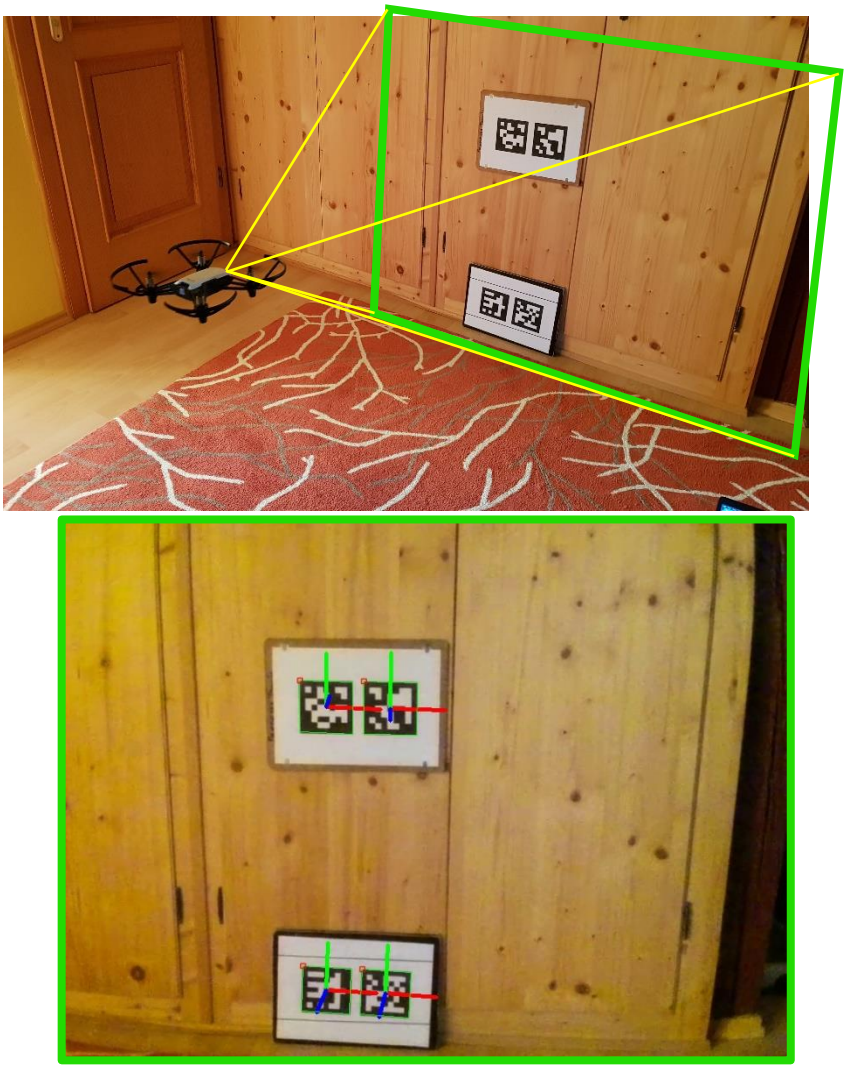
1. A program elindítása után a drón rögtön keressen markert! Amint talál egyet, az első helyzetadatainak egyike, legyen az elmozdulásokhoz választott origó! Azért nem feltétlenül az első, mert a kép legszélein található markerek a disztorziós korrekció ellenére sem biztos, hogy a legpontosabb értéket adják.
2. A drón előre kijelölt pályájának vonalán mindig legyen legalább egy marker a képben!
3. Mielőtt egy marker eltűnne a belátott képből, kell pár képkockányi idő, amíg a következő markerrel együtt látszik. Így a két marker helyzetéből adódó különbséggel a koordináta-rendszerek transzformációja elvégezhető és a drón helyzetmeghatározása megszakítás nélkül megoldható.
4. Az előre kijelölt pálya vonalán az egyes ID-vel jelölt markereknek előre meghatározott jelentésük kell legyen, hogy a drón hozzávetőlegesen tudja, merre keresse a következő markert.

Korábbi tanulmányokban is alkalmaztak már ArUco markereket drón vezérlésre. [25] Ebben a markereket a földre fektették és a drón inerciális szenzorainak értékeivel összevetve használták fel a kinyert információkat.

Fontos továbbá, hogy a drón által látott első marker orientációját viszonyítani tudjuk a drón kamerájának orientációjához. Ehhez feltételezzük, hogy a kamera a drón vízszintes, nyugalmi repülési helyzetében helyezkedik el, közel vízszintes helyzetben fedezi fel az 1. markert. A drón irányváltás és haladás során tér ki a vízszintes helyzetből, de ez a szögeltérés most elhanyagolható, mivel csak kis sebességek mellett szeretnénk egyelőre alkalmazni a programot. A kamera elhelyezéséből adódóan adódik egy 10° és 11° közötti depressziós szög, melynek pontos értéke nehezen megmérhető a drón házának kialakítása miatt, ezért vegyük 10,5°-nak!

3.5. Első tesztek a markerekkel történő helyzetmeghatározásra

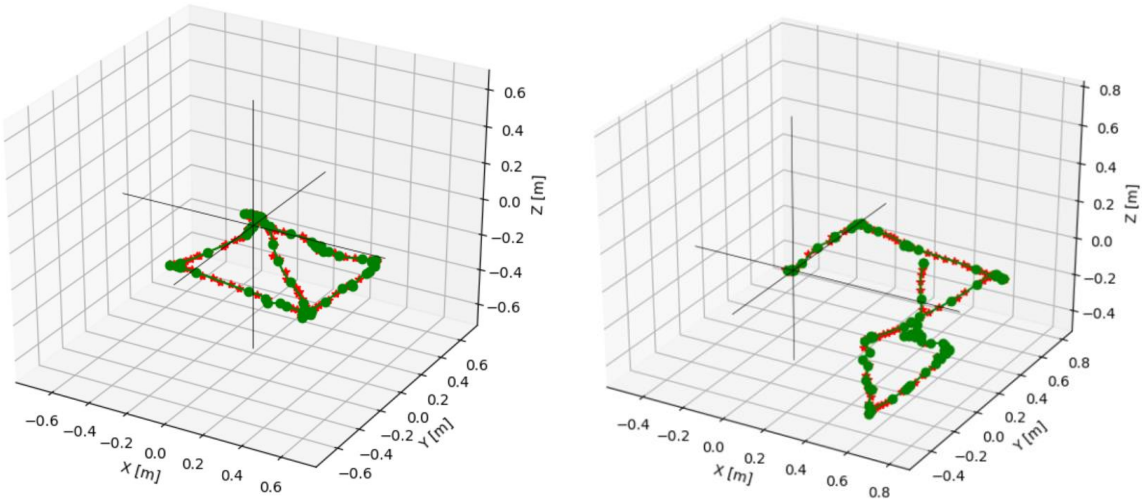
A teszteléshez négy 7x7-es ArUco markert használtunk, 0,0957 m oldalhosszúsággal, a DICT_7x7_100 kódú könyvtárból. 4x4, 5x5 és 6x6 területű markerek is rendelkezésre állnak, azonban a 7x7-es könyvtár nagyobb felbontása pontosabb pozícionálást tesz lehetővé. Akár 1024 különböző marker is elérhető minden könyvtárban, ami bőven elég lehet a drón irányításához szükséges információk közlésére.



10. ábra: A kísérleti markerelhelyezés, bejelölve a drón látóterét, illetve a drón által látott kép

A program markerek első felbukkanását veszi alapul, ezeket a pontokat eltárolja origóként, melyekhez képest aztán a markerek elmozdulását viszonyítjuk. Azért ajánlott egynél több marker alkalmazása, mert a rendszer több értékből képes átlagolni, így pontosabb eredményeket kaphatunk átlagolással, egyszerűen leszűrve az esetleges kiugró értékeket. Továbbá, ha több marker szerepel a drón látómezejében, akkor kevésbé baj, ha egyet közülük elveszt, hisz a többiből továbbra is képes értékeket kiolvasni.

Az első kísérlet során a markereket a drón kamerájára merőlegesen, függőlegesen egy falnak támasztva helyeztük el, a *10. ábra* fotóján látható módon. A teszt célja az volt, hogy mennyire képes a valóságot közelítő adatokat eltárolni a program.



11. ábra: Két, a drón által bejárt útvonal kirajzolva Descartes-koordináta-rendszerben

A *11. ábra* bal oldali ponthalmazán látszik, hogy a mérési ponthalmaz sikeresen lekövette, amint a drónnal egy négyzetet írunk le és még egyik átlóját is behúzzuk, a jobb oldalon pedig az látszik, hogy a magasságváltoztatást is kiválóan érzékelte a markerek segítségével a rendszer. A fekete tengelyek metszéspontja a drón elmozdulásainak origója. Az ábrán pirossal jelölt pontok a valós értékek, a zöld görbe pedig a Python programnyelvben elérhető *Scipy* csomag *interpolate* alkönyvtárában található *splprep()* függvény által a ponthalmazra illesztett B-spline görbe. A függvény finomítási paraméterét a dokumentáció alapján [26] pár kísérlet után s=0,1 értékűnek választottuk, mely már kellően közelíti a ponthalmazt a zajos pontok kiszűrésével. Végül a végleges programverzióban a B-spline interpolációs módszer helyett egy Kálmán-szűrőt alkalmazunk az adatpontok szűrésére, gondolva a későbbi real-time megvalósításra, illetve a Kálmán-szűrő adaptívabb viselkedésére.

Egyetlen korrekciót kellett alkalmazni a kísérleti ponthalmazon: az x tengely körül 10,5°-kal negatív irányba elforgatni a pontokat, különben konstans magasság esetén is emelkedtek volna a pontok Z-koordinátái a markerekhez közeledve. Ez a kamera vízszintestől eltérő szögéből adódik, a markerekhez közeledve magasabb pozíciót érzékelt. (Ez a későbbiekben a markerek szögelfordulására végzett korrekció miatt felesleges lesz.)

A drón ezen elrendezésben csak akkor ad valós elmozdulásértékeket, ha a kamera síkja (a depressziós szögtől eltekintve) párhuzamos az elhelyezett markerek síkjával. Amennyiben a helyzete ettől eltér, a drón már helytelen értékeket ad. Az elmozdulásértékeket a rotációs szögek figyelembevételével kell megbecsülnünk, különben nem tudjuk az ArUco markerek transzlációs vektorainak értékeit átkonvertálni az elsőnek látott marker koordináta-rendszerébe, amit globális koordináta-rendszerként szeretnénk használni. Az OpenCV csomag *cv2.aruco.estimatePoseSingleMarkers()* függvénye által adott vektorértékek a kamera koordináta-rendszerében adottak, melyeket majd át kell konvertálni a marker koordináta-rendszerébe.

3.5.1. A TESZTEK ÉRTÉKELÉSE ÉS A RENDSZERREL KAPCSOLATOS VÁRAKOZÁSOK

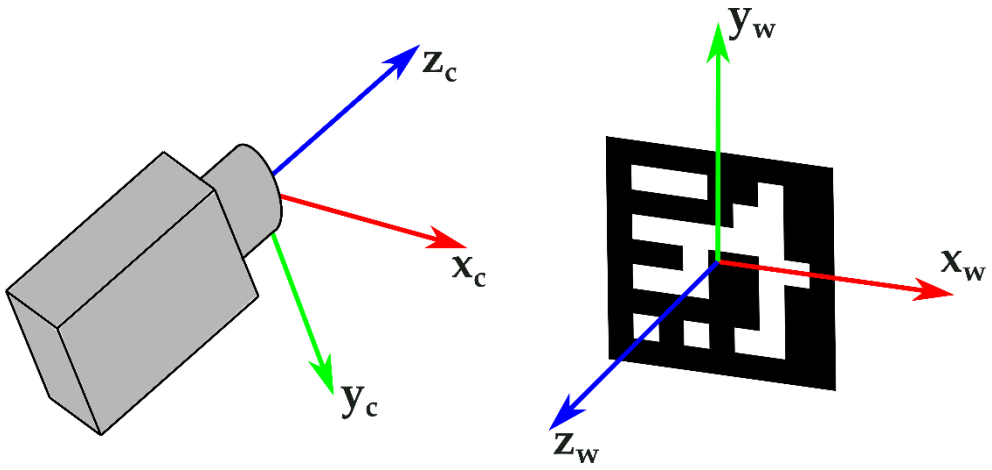
Az első tesztek alapján érdemes tovább vizsgálni ezt a pozíciómeghatározási lehetőséget, mivel viszonylag kevés transzformációval, valós időben képesek vagyunk vele a drón helyzetét kiszámítani. Sajnos, az *Nvidia* cég által biztosított *CUDA* programcsomag grafikus gyorsítását nem tudjuk alkalmazni, ugyanis az *OpenCV-Python* csomag függvényei még nincsenek mind hozzáadva a *cv2.cuda* könyvtárhoz Python programnyelven. A képfeldolgozás így lassabb lesz, de így is nagyobb csúszást fog okozni a működésben a drón által sugárzott kamerakép 1,5…2 másodperces késése.

A módszer mérési hibája előzetes várakozásaink alapján, növelt markermérettel és megfelelő számú marker felhelyezésével 0,05 m köré fog esni. A markerkoordináta-rendszerek közti átszámítások miatt azonban számíthatunk továbbá egy drifthibára, mely a globális koordinátát jelentő markertől távolodva növekszik, a koordináta-rendszerek között meghatározott transzformációk pontatlansága miatt.

3.6. A helyzetmeghatározás elve, finomítása és robusztussá tétele

A kamera programját nem a drón segítségével teszteljük, mivel a sok teszt közben az akkumulátorok töltése és az újbóli próbálkozások túl sok időbe teltek volna. A képfeldolgozó program írása során egy Logitech C250 típusú 640x480 pixel felbontású USB-s webkamerát alkalmazunk. A felbontása ugyan eltér a drón 960x720 pixeles felbontásától, de a program kialakítása alkalmas rá, hogy más kalibrálási adatok megadásával a dróntól független kamerával is futtatható legyen.

A program írása során az OpenCV ArUco csomagjában szereplő függvényeket alkalmaztuk, így a kamerahelyzet meghatározásához a *cv2.aruco.estimatePoseSingleMarkers()* függvényt. Ez a már vázolt kalibrálási eljárás után képes egy *Perspective-n-Point* (PnP) transzformáció segítségével minden markerhez egy transzlációs és egy rotációs vektort számolni a kamera koordináta-rendszerében, melynek konvenciói a *12. ábra* képén láthatók. A transzformációt a *cv2.aruco.solvePnP()* függvény végzi a látott markerek pixelben értendő sarokpontjai, a kameramátrix és a torzítási mátrix alapján, pinhole kameramodellt alkalmazva. [27]



12. ábra: Az OpenCV által értelmezett kamera- és marker-koordináta-rendszer

3.6.1. Perspektív vetítés, PnP transzformáció

A PnP transzformáció egy perspektív vetítés, amely a valós, 3D-s tér és a 2D-s képsík között teremt kapcsolatot a kalibrált kamera mátrixainak segítségével. Amennyiben ismert az objektumpontok egymáshoz viszonyított helyzete - jelen esetben a markerek sarokpontjai a megadott oldalhosszak távolságában helyezkednek el egymástól -, akkor a képsíkból képesek vagyunk a valós térbe átszámítani a markerek helyzetét.

Adottak egy 3D-s térben a már említett markersarokpontok, mint objektumpontok, és a 2D-s képsíkon a sarokpontok pixelben mért helyzete. Ezeket áttranszformálhatjuk a kamera koordinátarendszerébe, megkapva a marker kamerához viszonyított 6 szabadsági fokú (6 DoF) helyzetét egy transzlációs- és egy rotációs vektor segítségével. A *cv2.solvePnP()* függvénynek ezek alapján betáplálhatjuk a markerek oldalhosszát az objektumpontokhoz, a sarokpontjaikat pixelben mérve, mint vetítési pontokat, továbbá szükség van a kameramátrixra és a torzítási mátrixra, melyeket a kalibráció során kaptunk. Így egy marker sarokpontjai, a saját koordináta-rendszerében, ahol $mLen$ a marker oldalhossza méterben értelmezve:

$$\mathbf{x_{w_1}} = \begin{bmatrix} -mLen/2 \\ mLen/2 \\ 0 \end{bmatrix} \quad \mathbf{x_{w_2}} = \begin{bmatrix} mLen/2 \\ mLen/2 \\ 0 \end{bmatrix} \quad \mathbf{x_{w_3}} = \begin{bmatrix} mLen/2 \\ -mLen/2 \\ 0 \end{bmatrix} \quad \mathbf{x_{w_4}} = \begin{bmatrix} -mLen/2 \\ -mLen/2 \\ 0 \end{bmatrix}$$

Az objektumpontokat, mint világkoordináták pontjait ($\mathbf{x_w}$) levetíthetjük a képsíkra ($[u, v]$) egy perspektív vetítési mátrix ($\mathbf{\Pi}$) és a kamera intrinsic mátrixa ($\mathbf{A}$) segítségével, ahogy a *(3.1.) egyenlet* mutatja. Ezeken felül még szükség van egy ${}^{\mathbf{c}}\mathbf{M_w}$ mátrixra a marker világ koordináta-rendszeréből a kamera koordináta-rendszerébe történő transzformációhoz. Mivel a leképezés minden más paramétere ismert, az ismeretlen transzformációs mátrix a mátrixegyenlet átrendezésével megkapható. Az ${}^{\mathbf{c}}\mathbf{M_w}$ mátrix egy homogén transzformációt valósít meg a koordináta-rendszerek között egy forgatás és eltolás segítségével, a *(3.2.) egyenlet* szerint. A mátrixot átalakítva kapjuk a transzlációs- és rotációs vektorértékeket minden markerhez. [28]

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \mathbf{A\Pi^c M_w} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{3.1.}$$

$$\mathbf{A^{-1}\Pi^{-1}} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_w \\ y_w \\ z_w \\ 1 \end{bmatrix} \tag{3.2.}$$

Megjegyzés: A forgatási mátrix forgásvektorrá alakítása a *3.6.3 fejezetben* részletesen is ki van fejtve, az OpenCV által használt konvenciók szerint.

3.6.2. A MÉRÉSI KONVENCIÓK

A terv, hogy a markerek helyzetét és orientációját az elsőként látott, origónak vett markerhez kell eltárolni, így átszámíthatók a későbbi helyzetek is a globális koordináta-rendszerünkbe. A rendszer kialakítása során tételezzük fel, hogy az első markerhez minimális szöghelyzettel áll be a drón! Így annak elfordulását is ahhoz mérjük. A drón állapotkiolvasása során, lehetőségünk van a drón szöghelyzetének kiolvasására, ahogy azt már említettük. Mivel ez egy jól szűrt, robusztus szöghelyzetet ad a drón orientációjáról az Euler-szögek konvenciója szerint, ezért alkalmazzuk inkább ezt. Kísérletet tettünk a markerekből történő szöghelyzetkiolvasásra is, azonban a két módszer közül előbbi bizonyult megfelelőbbnek.

A drón Euler-szögek szerinti orientációjának feldolgozására és megjelenítésére forgatási mátrix számolható a *(3.3.) képlet* alapján. Ezzel megszorozva a drónnak megfelelő koordináta bázist, annak elforgatását kapjuk. A Tello drón esetében az alkalmazott szögkonvenciók megértését a 8. ábra segíti. Ezek balkéz-szabály szerint adottak, a drón elfordulása óramutató járásával megegyezően ad pozitív értéket.

$$\mathbf{R} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\gamma & -\sin\gamma \\ 0 & \sin\gamma & \cos\gamma \end{bmatrix} \begin{bmatrix} \cos\beta & 0 & \sin\beta \\ 0 & 1 & 0 \\ -\sin\beta & 0 & \cos\beta \end{bmatrix} \begin{bmatrix} \cos\alpha & -\sin\alpha & 0 \\ \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{3.3.}$$

ahol:

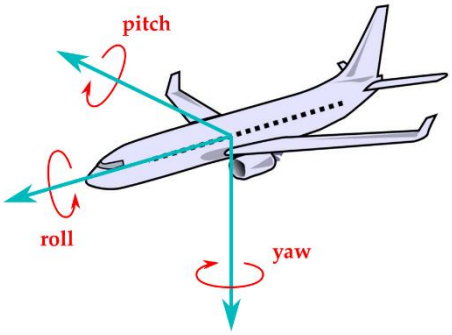
R: a szögekből kapott forgatási mátrix

α: a *z* tengely körül (yaw – függőleges tengely körüli fordulás)

β: az *x* tengely körül (pitch – transzverzális tengely körüli „buktatás")

γ: az *y* tengely körül (roll – hosszanti tengely körüli „döntés")

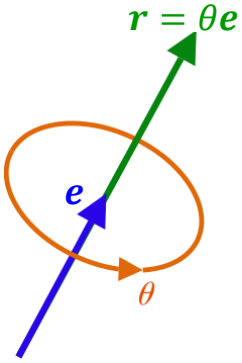
Mivel az Euler-szögek konvenciójára nincs egyetemes megállapodás, mindig definiálnunk kell az alkalmazott konvenciót. Jelen esetben a már említett rendben, a drón SDK-ból kapott negatív értékek szerint tároljuk el a szögeket. Ezek megegyeznek a DIN 9300-2 repülési szabvány (*13. ábra*) szerinti légi járműveknél használt szögkonvencióval. A választott szögkonvenció pontosabb meghatározási csoportja: *Tait-Bryan-szögek z-y'-x"* intrinsic konvenciója, más néven nautikus- vagy *Cardan-szögek*. [29]



13. ábra: Egy légi jármű főtengelyei DIN 9300 szabvány alapján [13.Im]

3.6.3. AZ OPENCV ÁLTAL HASZNÁLT KONVENCIÓK

Az *estimatePoseSingleMarkers()* függvény által adott értékek közül a transzlációs vektor értékei méterben értelmezettek, a forgásvektor pedig egy ún. tengely-szög (axis-angle) vagy Rodrigues szerinti reprezentációt használ, ami a rotációs mátrix legkompaktabb tárolási formája. Az OpenCV *Rodrigues()* függvénye képes sorvektorként megadott rotációs vektorokból a *(3.4.) egyenlet* szerint rotációs mátrixot számolni, illetve ennek inverzének meghatározására is alkalmas, ha a bemenet egy forgatási mátrix. [28]



14. ábra: A Rodrigues-féle tengely-szög forgásvektor értelmezése [14.Im]

A forgásvektor legyen a következő (a *14. ábra* szerint):

$$\mathbf{r} = \theta\mathbf{e}$$

így: $\theta = norm(\mathbf{r})$ és $\mathbf{e} = \mathbf{r}/\theta$

$$\mathbf{R} = cos\theta\mathbf{I} + (1 - cos\theta)\mathbf{e}\mathbf{e}^{\mathbf{T}} + sin\theta \begin{bmatrix} 0 & -e_z & e_y \\ e_z & 0 & -e_x \\ -e_y & e_x & 0 \end{bmatrix} \tag{3.4.}$$

ahol:

R: a Rodrigues-féle forgásmátrix

r: a Rodrigues-féle forgásvektor

e: a forgástengely egységnyi irányvektora

θ: a forgásvektor normája (hossza), a tengely körüli elfordulás szöge

3.6.4. A KAMERAHELYZET ÁTSZÁMÍTÁSA GLOBÁLIS KOORDINÁTA-RENDSZERBE

Felmerült még a homogén transzformációk alkalmazása is, ugyanis a koordináta-rendszerek átszámítása, egy forgatás és egy eltolás egyszerre valósítható meg. Így egyetlen mátrixszorzás segítségével egymásba vihetők lennének a koordináta-rendszerek, ha a kamerahelyzetet egy quaternióvá alakítjuk. Ezt a transzformációt mutatja a *(3.5.) egyenlet*. Az *OpenCV* által használt vektorkonvenciók mellett azonban átláthatóbbnak bizonyult a markerek koordináta-rendszereinek külön történő elforgatása és eltolása, mely a *(3.6.) egyenleten* látható.

A transzformáció az *n.* marker koordináta-rendszerének egy pontjára mutató $\mathbf{t_{nn}}$ vektor és a globális koordináta-rendszer között teremt kapcsolatot. Ez levetíti a pontot a $\mathbf{dR_n}$ forgatási mátrix segítségével a 0. koordináta-rendszerbe, illetve eltolja azt az *n.* koordináta-rendszer origójába, melyet a $\mathbf{t_{on}}$ vektor ad (globális koordináta-rendszerben értelmezve).

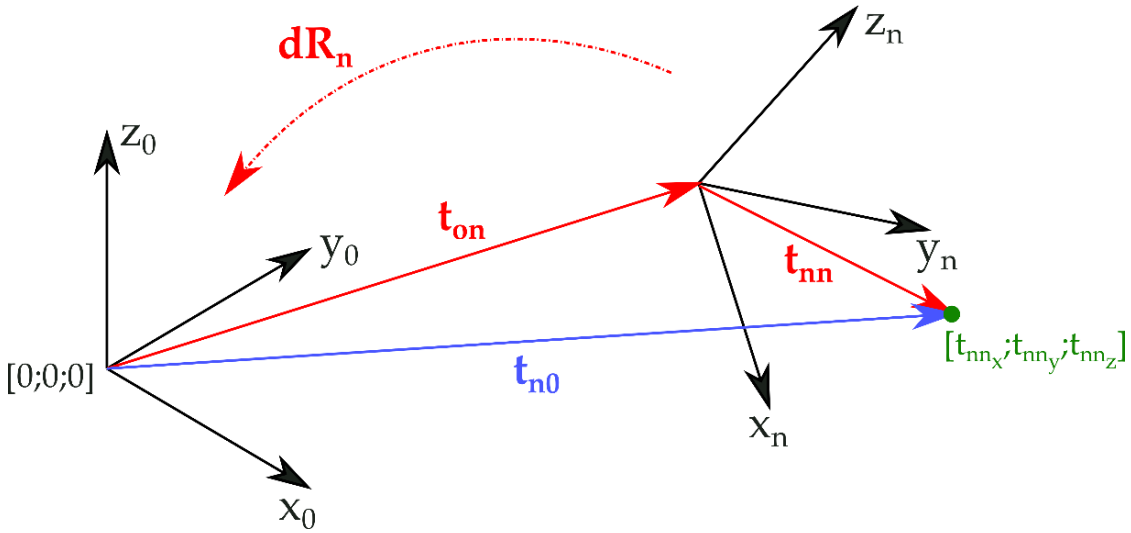
Homogén transzformációval:

$$\mathbf{t_{n0}} = \begin{bmatrix} \mathbf{dR_n} & \mathbf{t_{on}} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{t_{nn}} \\ 1 \end{bmatrix} = \begin{bmatrix} dR_{n_{11}} & dR_{n_{12}} & dR_{n_{13}} & t_{on_x} \\ dR_{n_{21}} & dR_{n_{22}} & dR_{n_{23}} & t_{on_y} \\ dR_{n_{31}} & dR_{n_{32}} & dR_{n_{33}} & t_{on_z} \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} t_{nn_x} \\ t_{nn_y} \\ t_{nn_z} \\ 1 \end{bmatrix} \tag{3.5.}$$

Eltolással és forgatással (ezt alkalmazzuk):

$$\mathbf{t_{nm}} = \mathbf{t_{on}} + \mathbf{dR_n t_{nn}} \tag{3.6.}$$

Így minden időpillanatban kiszámolható a látott marker koordináta-rendszerében értelmezett kamerahelyzet, majd onnan átszámítható a globális koordináta-rendszerbe a *15. ábra* illusztrációja szerint. Ezekhez szükség van a koordináta-rendszerek közötti eltolások és forgatások meghatározására. Az elképzelés szerint a program ezt a drón kameraképéből két látott marker esetén, adott számú mintából automatikus átlagolással számolja és tárolja el.



15. ábra: A globális koordináta-rendszerbe történő transzformáció illusztrációja

3.6.5. A MARKEREK ORIGÓJÁNAK ÉS FORGATÁSÁNAK SZÁMÍTÁSA

Minden markerhez szükséges eltárolni egy eltolást, mely az origójának helyzetét adja a globális koordináta-rendszerben, illetve egy forgásmátrixot, mely a marker koordináta-rendszerét elforgatja (levetíti) a globális, 0. koordináta-rendszerbe. Ezek az *n.* marker esetén már említett $\mathbf{t_{on}}$ transzlációs vektor és a $\mathbf{dR_n}$ rotációs mátrix.

Meghatározásuk a következőképpen történik, ha adott:

- → $\mathbf{t_{om}}$: az *m.* krsz. origójába mutató globális vektor
- → $\mathbf{dR_m}$: az *m.* krsz.-t a globálisba forgató mátrix
- → $\mathbf{t_{cam_m}}$ és $\mathbf{t_{cam_n}}$: az *m.* és az *n.* marker vektora a kamera krsz.-ében
- → $\mathbf{R_{cam_m}}$ és $\mathbf{R_{cam_n}}$: az *m.* és az *n.* marker forgásmátrixa a kamerához képest (ezeket a *(3.4.) transzformáció* alapján beépített függvénnyel számolhatók)

Ismert, hogy a forgatási mátrix Descartes-koordináták között ortogonális transzformációt valósít meg, azaz a forgásmátrix inverze megegyezik a transzponáltjával ($\mathbf{R^{-1}} = \mathbf{R^{T}}$). Így általánosan a következő transzformáció igaz a kamera helyzetének a marker koordináta-rendszerébe történő átszámítására:

$$\mathbf{t_{marker}} = -\mathbf{R}_{\mathbf{cam}}^{\mathrm{T}} \cdot \mathbf{t_{cam}} \tag{3.7.}$$

A markerek közötti különbség a kamera koordináta-rendszerében:

$$\mathbf{dt_{cam}} = \mathbf{t_{cam_m}} - \mathbf{t_{cam_n}} \tag{3.8.}$$

Ezek alapján az *n.* marker origója számolható a *(3.9.) egyenlet* alapján az *m.* marker ismert adatai segítségével. A markerek transzformációi a globális origótól indulva halmozódnak így.

$$\mathbf{t_{on}} = \mathbf{t_{om}} + \mathbf{dR_m}\left(-\mathbf{R}_{\mathbf{cam_m}}^{\mathrm{T}} \cdot \mathbf{dt_{cam}}\right) \tag{3.9.}$$

Mivel a két marker egymáshoz képesti helyzete rögzített, ezért közöttük felírható egy konstans forgatási transzformáció, mely az *n.* marker koordináta-rendszeréből az *m.* markerébe forgat. Ez bizonyítható a forgatás sajátosságai és a mátrixegyenletek alapján, azonban erre jelen esetben nem térünk ki. Ezek alapján a forgásmátrix általánosan a következőképpen kapható:

$$\mathbf{dR} = \mathbf{R}_{\mathbf{cam_m}}^{\mathrm{T}} \cdot \mathbf{R_{cam_n}} \tag{3.10.}$$

A globális koordináta-rendszerbe forgató mátrix az *n.* markerre:

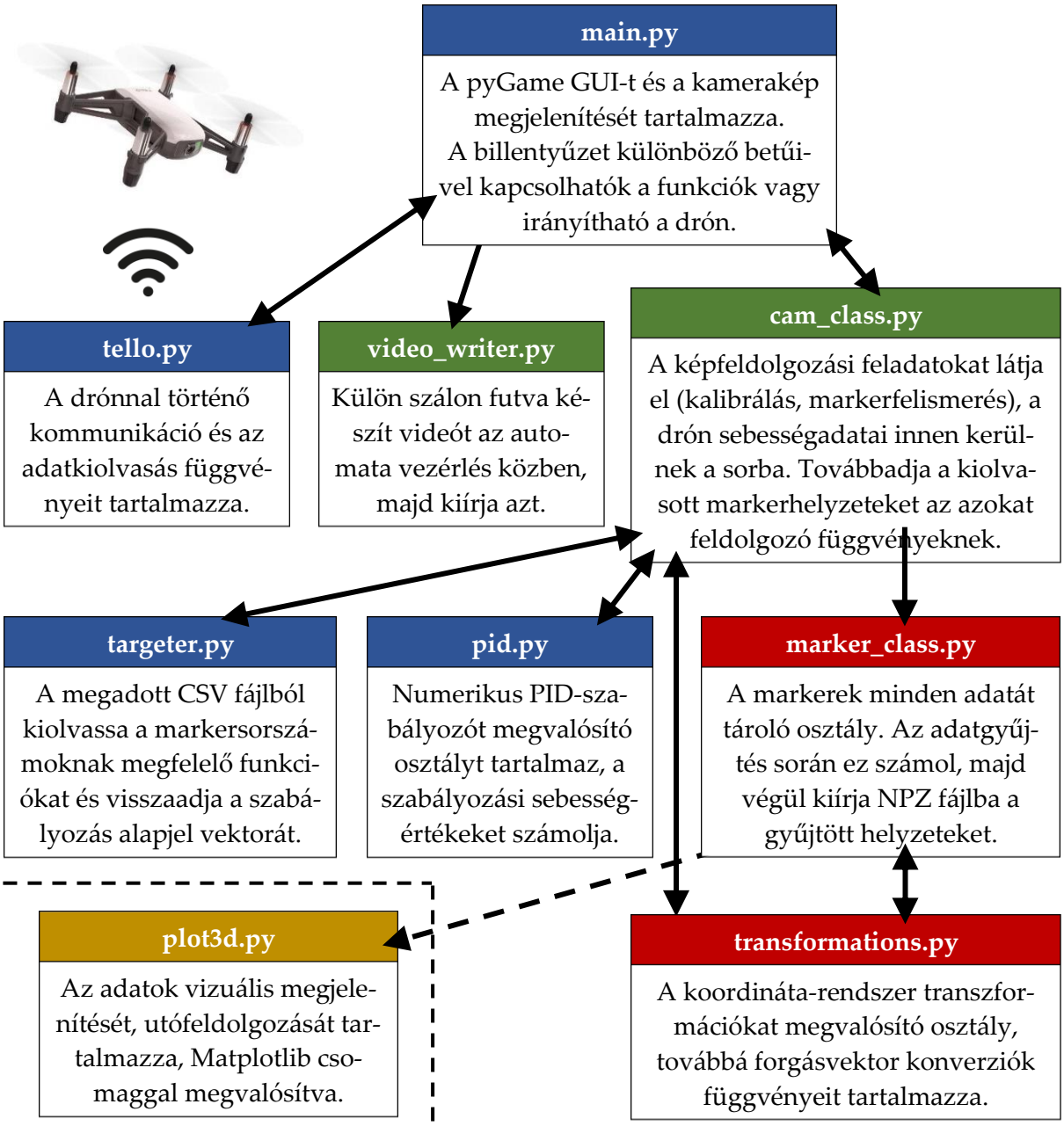
$$\mathbf{dR_n} = \mathbf{dR_m}\left(\mathbf{R}_{\mathbf{cam_m}}^{\mathrm{T}} \cdot \mathbf{R_{cam_n}}\right) \tag{3.11.}$$

Ezek az értékek a tényleges programban adott számú mintából átlagolással számítódnak a lehetséges kiugró értékek leszűrése érdekében. Így a kamera leképezési hibái kompenzálhatók. Az így kapott $\mathbf{t_{on}}$ vektort és a $\mathbf{dR_n}$ mátrixot eltároljuk az *n.* marker adatai közé, később ezek segítségével számítható át a kamerahelyzet a globális koordináta-rendszerbe. A módszer hátránya, hogy adott ideig mindkét marker, melyek között transzformációt számolunk, megfelelően kell látszódjon a kameraképen.

4. A DRÓN PROGJAMJÁNAK MEGVALÓSÍTÁSA

4.1. A programok struktúrájának áttekintése

A drón irányítását, illetve az adatgyűjtést összesen 8 program végzi. Az adatfeldolgozás külön scriptben végződik, a kamerakép késése miatt real-time adatmegjelenítésre egyébként sincs lehetőség. A programok struktúrájának értelmezéséhez készült a *16. ábra*, melyen látszik, hogy melyik Python script melyik másikkal áll kapcsolatban és milyen feladatot lát el. Minden script fájl a csatolt DVD *1. mellékletében* megtalálható.



16. ábra: A drónt vezérlő programok struktúrája

4.2. A drón irányítását végző programok

A drónnal való kommunikációt a *tello.py* program végzi, ennek a kódnak nagy része a *Damià Fuentes* által megírt [18] drónvezérlő programot tartalmazza, ahogy a *main.py* billentyűzetről történő vezérlése és pyGame kezelőfelülete is át lett véve. A 2. kódrészletben már említett állapotkiolvasó thread új funkció, mely az adatgyűjtés során fontos számunkra.

A pyGame GUI hatékonyan képes lekezelni a billentyűleütéseket a Python programnyelvben, ezért esett a döntés a megtartására. Így a *main.py* programban történő vezérlés során megmaradt a billentyűzettel történő irányítás lehetősége is, a veszélyes helyzetekben történő beavatkozás esetére. Megfelelő billentyűk különböző eventeket kapcsolnak, melyekkel a program egyes funkciói elindíthatók, így például a navigáció, kalibrálás stb. A Python nyelvben az eseményosztály jelentősége és előnye az egyszerű változóként tárolt igazságértékekhez képest, hogy szálak közti kommunikációra is képes. Így egymástól külön futó szálakban is figyelhetjük ugyanannak az eseménynek a bekövetkezését, nem kell függvényargumentumként vagy sorban átadni, inicializálás során, csak egyszer szükséges megadni.

A navigáció típusa között az „O" billentyű megnyomásával válthatunk, ez kapcsolja be a markerkereső és vezérlő programot. A sebességértékeket a *cam_class.py* Camera osztályának *navigateToMarker()* függvénye számolja, és helyezi el egy sor típusú tárolóban. (Erről részletesen a *4.3 fejezet* értekezik.) Ennek alkalmazása során a sor telítődésére kell ügyelni, ugyanis ha a sorba gyorsabban töltünk elemeket, mint ahogy azokat kiolvassuk, akkor a kiolvasás során egy növekvő késés jön létre. Ezért ajánlott a Queue típus *clear()* függvényét alkalmazni egy-egy kiolvasás után, így a sor rendszeresen kiürül, nem kell tartanunk a telítődésétől. Hasonlóképpen a „K" billentyű lenyomására újrakalibrálhatjuk a drón kameráját, az „M" billentyű lenyomásával a program kimenti a kameraképet egy JPG fájlba, az „X" billentyűvel pedig a videóírást zárhatjuk le. (Ezzel felülírva az automatikus lezárást.)

Az állapotadatok sorából itt olvasódnak ki ciklusonként a drón által sugárzott adatok, illetve a kamerakép is. Nevéből is adódóan ez a fő programciklus, innen adjuk át a képfeldolgozó programnak a kameraképet, illetve küldjük azt a videóíró programszál sorába.

4.2.1. DRÓN POZÍCIÓSZABÁLYOZÁSA

A drón vezérlés szerinti helyzetébe történő beállásához egy numerikus PID-szabályozó alkalmazása szükséges, párhuzamos P, I, D elrendezésben, az időlépést konstansnak feltételezve. Ennek behangolása tapasztalati úton történt, a drónnal történő kísérletezéssel. A szabályozás a kamera koordináta-rendszere szerint történik, így ugyanis egyszerűbb lekezelni a különböző orientációjú markerekhez történő beállást.

Nehézséget jelent a szabályozásban, hogy a kamerakép 1,5…2 másodperces késleltetéssel érkezik meg a laptop szerverére. A kísérleti behangolás során igyekezni kell ezt is figyelembe venni. A tapasztalatok szerint továbbá, a függőleges mozgás irányában (z koordináta), a szabályozó erősítését növelni kell, a hatékonyabb beállás érdekében. Ez a fel-le történő mozgáshoz szükséges nagyobb motornyomatékból adódhat. A *yaw* irányú szögsebesség szabályozására is külön szabályozó beállítása történt meg. Amennyiben a drón teljes 2 másodpercre elveszti a markert, keresni fogja az utoljára látott x irány szerint elforogva jobbra vagy balra.

A beálláshoz szükséges célértékeket markertől függően változtatjuk, ez a szabályozó referenciajele, hibaszámítást ehhez végez. A szabályozó kimenetét még lineárisan erősíteni kell a drón sebességértékével és egy konstans erősítéssel. Ezt az erősítést szintén kísérleti úton határozhatjuk meg. A drón automatikus vezérlése során használt alapsebesség értéke 0,15 m/s, azaz a drón vezérlésében 15 cm/s. Gyorsabb alapsebesség beállítására a kamerakép késése miatt nincs lehetőség.

Amennyiben a drón beállt a kívánt helyzetbe, léphet a következő markerre, annak kívánt helyzete kerül a szabályozó alapjelére és ehhez igyekszik beállni. A beállást a drón kívánt helyzetéhez egy bizonyos hibahatárral figyeljük. Ha ez a feltétel teljesül, az épp látott, új markerek közül a közelebbihez próbál beállni.

A 3. kódrészletben látható a szabályozó megvalósítása. A PID osztály egy példányában tárolt változók a következők: K_P, K_I, K_D, korábbi hiba és hibaintegrál. A szabályozó a drón mindhárom irányában a *(4.1.) egyenlet* szerint valósítható meg, ahol e_n az alapjelhez képesti hiba az n. időpillanatban, K_P az arányos tag, K_I az integráló tag és K_D a deriváló tag erősítése. Az így kapott v_{n0} értéket még erősítenünk kell, hogy a megfelelő $\mathbb{Z} \in [-100;\ 100]$ tartományba essen a drón RC vezérlése számára. Ha a betáplált sebesség ezen az intervallumon kívül esik, a drón automatikusan maximálisnak veszi.

$$K_P \cdot e_n + K_I \cdot \sum_{i=0}^{n} e_i + K_D \cdot (e_n - e_{n-1}) = v_{n0} \tag{4.1.}$$

3. kódrészlet: A numerikus PID-et megvalósító függvény (*pid.py/Class PID/def control()*)

```
def control(self, error):
    self.error_int += error
    if self.error_prev is None:
        self.error_prev = error
    error_deriv = error - self.error_prev
    self.error_prev = error
    return self.kp*error + self.ki*self.error_int + self.kd*error_deriv
```

A választott szabályozótagok értékeit és az utólagos erősítést (A) az egyes irányokban a *1. táblázat* tartalmazza. A sebességadatok az irányokat tartalmazó Queue-ba töltődnek és a drón *main.py* programjában akkor olvassuk ki, ha szerepel a sorban érték. Az így kapott értékeket még egész számmá kell konvertálni a Python *int()* típuskonverziójával, ugyanis a drón vezérlése csak egész számokat képes értelmezni. Így már elküldhető a *tello.send_rc_control()* függvénnyel a drónra a 4 sebességérték.

1. táblázat: A PID-szabályozók tagjainak megválasztott értékei irányonként (S az alapsebesség)

sebességirány	K_P	K_I	K_D	A
x	0.3	0.00001	0.0001	$10S$
y	0.3	0.00001	0.0001	$10S$
z	0.8	0.00001	0.0001	$10S$
yaw	0.1	0.00001	0.001	$S/2$

Az egyes markerek sorszámaihoz rendelt célhelyzet-típusok kiolvasása egy CSV-fájlból történik, hogy a későbbi módosítás és markerbővítés egyszerűen, a programban való módosítás nélkül történhessen. 10 markertípust hoztunk létre, mindegyikhez más alapjelvektor tartozik, ezek között vált a CSV-fájl és sorszám alapján a *targeter.py* script. Így a célmarker sorszámának megváltoztatása után a célvektort is módosítva az eszköznek új helyzetbe kell beszabályoznia magát.

A meghatározott alap markertípusok a következők:

- **origó** (*"Origin"*): Mindig az 1. sorszámú marker, vele szembe kell beálljon a drón, hogy a koordináta-rendszereik közti szöghiba a lehető legkisebb legyen.
- **jobb oldalról** (*"Right sideways"*) **és bal oldalról** (*"Left sideways*") **követendő egyenes navigáció**: A további markerek belátása érdekében egy fal mellett adott szögben navigáló drónhelyzet.
- **jobbra és balra fordító markerek** (*"Right/Left rotate corner 1/2/3"*): Egyetlen sarokba több is kellhet, a lépcsőzetesség az egyszerre látott markerek miatt és a falba ütközés elkerülése végett lényeges.
- **végmarker** (*"End"*): Jelenleg az 50. sorszámú marker, de ez bővíthető. Vele szemben beállva a drón lezárja az adatgyűjtést és a videóírást.

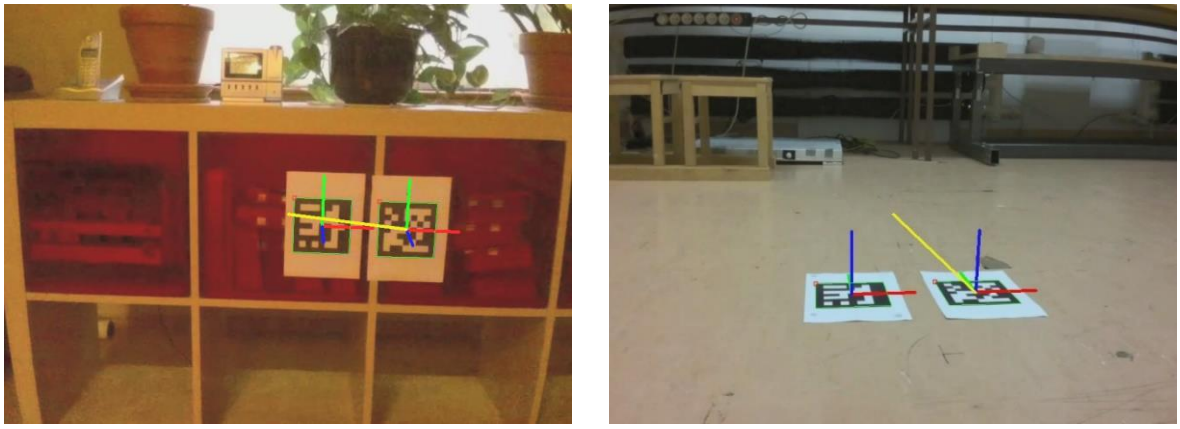
4.3. A képfeldolgozást végző programok

A drón irányításához szorosan kötődik, azonban inkább képi adatfeldolgozást végez a *cam_class.py* program, melynek fő függvénye az ArUco markerfelismerésért felel. További képfeldolgozást végző program az OpenCV *cv2.videoWriter()* függvényét használó, külön szálban futó *video_writer.py*, mely a drón kameraképéből automatikusan eltárol egy AVI formátumú videót. Erre részletesen jelen munkában nem térünk ki.

Az ArUco markerek felismerése az OpenCV beépített *cv2.aruco.detectMarkers()* függvénnyel történik. Ez egy szürkeárnyalatos képen alkalmazott adaptív szegmentálás után megtalálja a betöltött markercsomagban szereplő kódokat, és visszaadja azok pixelben mért sarokpontjait és a sorszámukat. A már korábban említett PnP transzformáció segítségével számolható a markerek valós, háromdimenziós helyzete a kameramátrixok alapján. A kapott helyzetek marker-sorszámukkal megfelelően küldhetők tovább feldolgozásra (*4.4 fejezet*), illetve alkalmazhatók a drón automatikus navigációjára.

4.3.1. A DRÓN NAVIGÁCIÓJA A MARKEREK SEGÍTSÉGÉVEL

Mivel később a kameraképen is meg akarjuk jeleníteni a markert, melyhez képest navigál a drón, ezért a navigációt végző függvény nem került a kameráétól külön osztályba. A célmarker jelzésére szolgál a kameraképen berajzolt sárga egyenes a kép középpontja és a marker pixelben mért középpontja között. (*17. ábra*) Ennek vektoriális jelentése nincs, csupán szemléltető célt szolgál a kezelő számára. Jelen fejezet vezérlési programleírásaiban is ilyen képkockák szerepelnek, a drón által automatikusan rögzített videóból kimentve őket.



a) b)

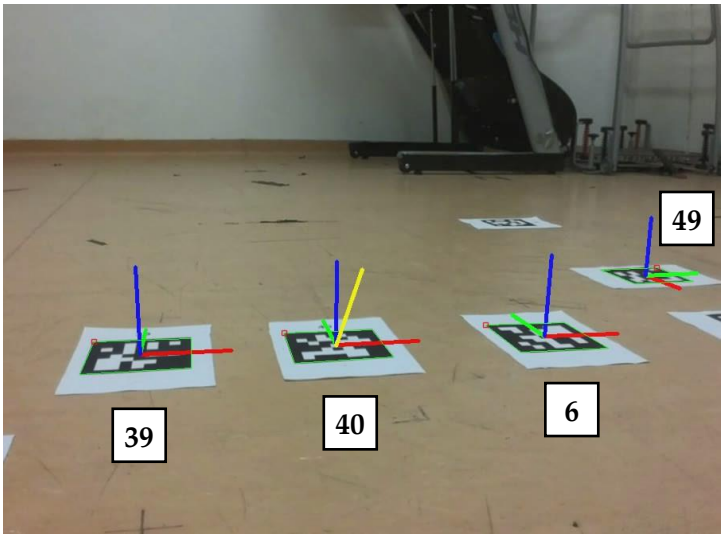
17. ábra: A drónnavigáció indítása utáni pillanat, az 1. markerhez (balra) való beállás után. Az a) esetben függőleges, a b) esetben pedig vízszintes markerorigóval indul a vezérlés. Az épp folyamatban lévő, 2. markerhez való vezérlést a sárga szakasz szemlélteti.

A navigáció elindításának feltétele, hogy a drón a megkövetelt hibavektorral beálljon az 1. sorszámú markerhez, melyet origónak választunk. Mivel a drón kamerájának és nem a markernek a koordináta-rendszerében figyeljük a helyzetet, a drón így kétféle orientációjú marker esetén tud helyes koordináta-rendszer origót elhelyezni: vízszintesen és függőlegesen, helyesen (nem ferdén) felhelyezett markert képes origóként értelmezni. (17. ábra a) és b)) A többi marker orientációja nem fontos a transzformációk miatt. A beállás feltétele, hogy a drón bizonyos hibahatáron belül beálljon a célmarkerhez. A megengedett hibavektor értékeit mutatja a *2. táblázat*. A beállás további feltétele, hogy 1 másodpercig a drón hibahatáron belüli helyzetben legyen. A hiba értéke pozitív és negatív is lehet, ezért abszolútértékre vizsgálunk.

2. táblázat: A drón navigációja során megengedett hibahatárok

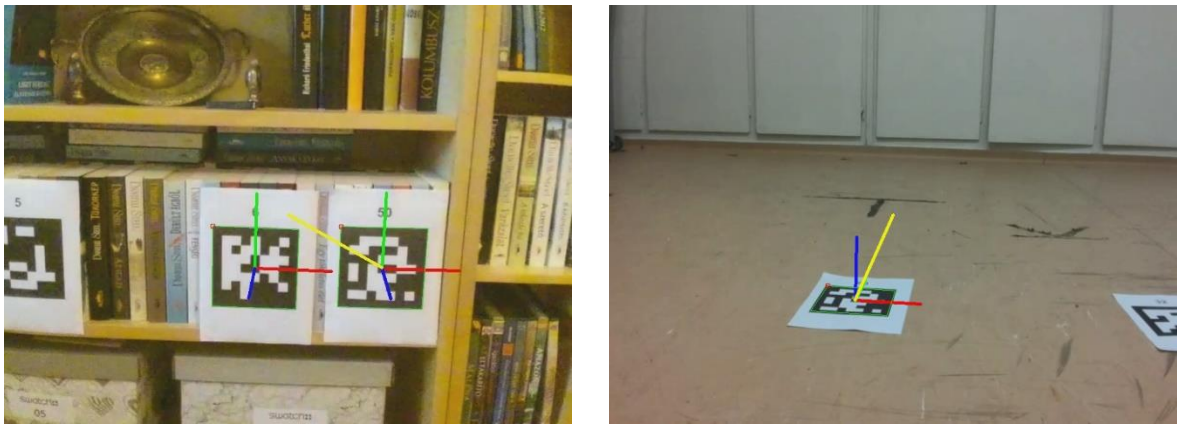
irány	x	y	z	yaw
megengedett hiba abszolútértéke	0.15 [m]	0.15 [m]	0.15 [m]	5 [°]

Amint sikerült a megadott hibahatáron belül beállni, átadja a vezérlést a kamera koordináta-rendszerében nézve legközelebbi, navigáció során még nem használt markernek. (*18. ábra*) Ezt a vektorok hossz-számításával határozhatjuk meg, majd minimumot keresünk. Az átadás során megváltozik a szabályzó alapjele, a célvektor is, ezt a CSV-fájl adatai alapján a *targeter.py* egyik függvénye határozza meg.



18. ábra: A drón balról érkezett, jelenleg épp a középső, 40. markerhez navigál. A bal oldali, 39. marker közelebb lenne a kamerához, azonban ott már járt, ezért a két jobb oldali (6. és 49.), látott marker közül választja ki a neki közelebbit, a 6. sorszámút és annak adja át a vezérlést.

Az 50. sorszámú marker lett kiválasztva végmarkernek, az ehhez való beállás után a program kimenti a gyűjtött adatokat, leállítja a navigációt és kiírja a rögzített videót. (*19. ábra*) Amennyiben a drón nem jut el az 50. markerig, nem képes ahhoz beállni, vagy valamelyik markerek közti transzformáció hiányzik, a félkész adatsor és videó nem íródnak ki. Ezért fontos, hogy a markerek elhelyezése megfelelő legyen, elegendő idő jusson a transzformációk kiszámítására. A program futása során a konzolba íródnak a fontos üzenetek, ezeket a kezelőnek érdemes figyelnie, hogy korrigálhassa a hiányzó számításokat.



a) b)

19. ábra: A vezérlés leállítása előtti pillanat, az 50. markerhez történő sikeres beállás függőleges (a) és vízszintes (b) elhelyezésű markerek esetén.

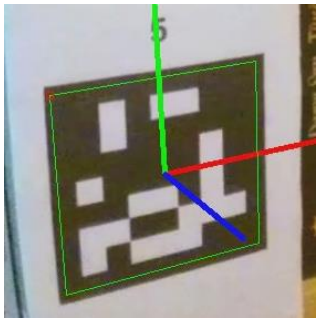
Mivel a drón kamerája nem mozgatható, a vízszintes markerekkel történő navigáció esetén csak alacsonyan, 0,3…0,4 méter magasan képes repülni. Így képes a 10,5°-os depressziós szögben elhelyezkedő kamera maga elé és alá látni. A kamera z irányában 0,9 métert választva beállási pozíciónak, a drón a vízszintes markerek közt is biztosan képes navigálni.

4.4. Adatfeldolgozást végző programok

A kameraképet feldolgozó programokból nyert térbeli pontok és forgatások feldolgozását, vagyis az adatgyűjtést a *marker_class.py* program végzi. A szükséges transzformációkat, amiket a *3.6 fejezetben* részleteztünk, a *transformations.py* függvényei végzik. A gyűjtött ponthalmaz szűrését és megjelenítését utólag a *plot3d.py* programban végezhetjük.

4.4.1. A MARKERADATOK TÁROLÁSA

Az első feladat, a *3.4.2 fejezet* 1. szabályának megvalósítása egy szűréssel. Csak azoknak a markereknek az adatai juthatnak tovább az adatfeldolgozásra, melyeknek egyik sarokpontja sincs a kép pixelben mért keretében. A szél definíciójaként a kamerakép mind a négy oldalán 2-2-2-2% vágódik le. A széleken elhelyezkedő markerek egyrészt a torzítás kiküszöbölése után is pontatlanabb adatokat szolgáltatnak, másrészt a markerdetektálás során a 2D-s markerkód külső széleinek jelölését bizonytalanná teheti, ahogy az a *20. ábra* képén is látszik.



20. ábra: Kimetszett képrészlet az egyik rögzített videóból egy helytelenül detektált markerről

A markerek osztályában különböző listákban tárolódnak az addig látott markerek tulajdonságai, így:

- **`ids:`** már használt markerek sorszáma
- **`tvec_origin:`** markerorigók a globális koordináta-rendszerben
- **`rvec_origin:`** marker koordináta-rendszerek forgatása a globálishoz képest
- **`dRot:`** a forgatási mátrix az adott markerből a globális krsz.-be
- **`allow_use:`** az átlagolás során szükséges segédváltozó, adott marker engedélyezésére, ha már elegendő mintát gyűjtöttünk róla, számolhatunk vele
- **`tvec_min, tvec_max:`** segédváltozók az átlagszámítás szűréséhez

Az ArUco markerek felismerése során meghívott *appendMarker()* függvény adja hozzá a markerek osztályához az új markereket. Az 1. sorszámú markert felveszi origóként, az x tengely körüli Euler-szögelfordulása alapján megállapítva, hogy vízszintes vagy függőleges markerről van szó. Továbbá a drón állapotainak kiolvasásából eltárolja a szögelfordulások alapértékeit, ez lesz az Euler-szögek origója. Ez eltárolja a kétféle orientáció közül a megfelelőt egy mátrixban, melynek az utófeldolgozás indexelése során lesz jelentősége, a transzlációs vektor irányainak felcserélésével.

További markereket a koordináta-rendszerek közti transzformációk megteremtésével tárol el a rendszer, a *3.6.5 fejezetben* foglaltakat a *transformations.py* script *getTransformations()* függvénye valósítja meg. A működés során jelenleg 12 érvényes mintából dolgozik a rendszer. Ennél magasabb engedélyezési határral is történtek próbák, azonban úgy az átszámításokat sokszor nem volt ideje a rendszernek elvégezni, míg a drón látóterében szerepeltek a markerek és nem volt képes tovább számolni. Így 12 eltolási vektort gyűjt a két marker között, a minimum és maximum normájút eldobva és a 10 fennmaradó értékből átlagolva. A forgásmátrixok is átlagolással állnak elő 12 mintából, szűrés nélkül. A kapott értékeket eltároljuk az osztály objektumlistáiban.

Amennyiben egy markernek már megvan az engedélyezéshez szükséges mintája, a *4. kódrészletben* foglalt függvény számolhat vele a *3.6.4 fejezet* egyenletei alapján. A program az összes látott markerből kiolvasott helyzettel számol, átlagolja azokat és ezt a pozícióértéket tárolja el. A drón szenzorából kiolvasott szögelfordulásokból vonja ki a szögek origóját, így adódik a drón orientációja. Az adatokat az eltárolásuk időpontjával együtt tömbökbe gyűjtjük, és a navigáció lezárása után írjuk ki NPZ-fájlba.

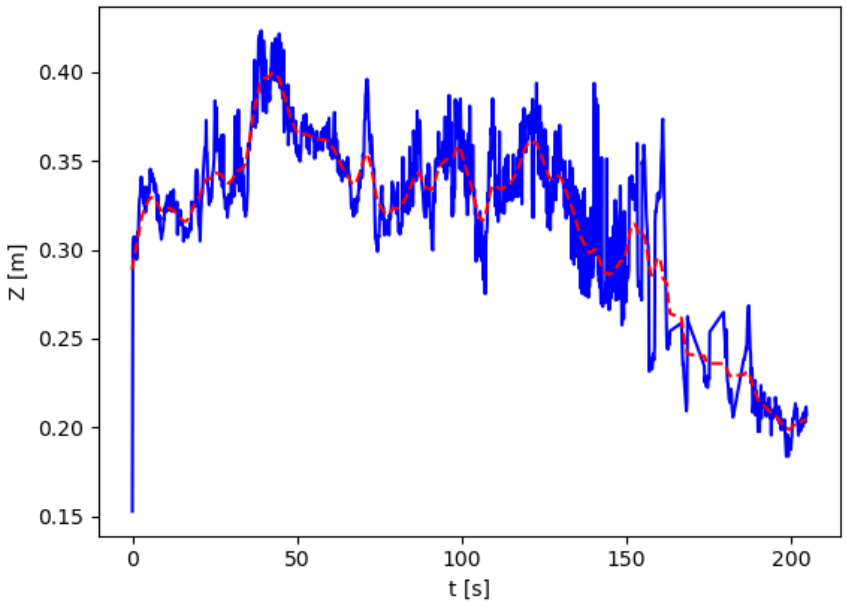
4. kódrészlet: A markerek transzlációs vektorait globális krsz.-be átszámoló függvény (*transformation.py/def calculatePos()*)

```
def calculatePos(tvec, rvec, tvec_orig, dRot):
    tvec = np.transpose(tvec)
    tvec_orig = np.transpose(tvec_orig)
    R = cv2.Rodrigues(rvec)[0]
    tvec = -R.T.dot(tvec) # kamera pozíció a marker koordináta-rendszerében
    tvec = tvec_orig + dRot.dot(tvec) # átszámítás globális krsz.-be
    tvec = np.transpose(tvec)
    return tvec
```

4.4.2. A KIÍRT ADATOK UTÓFELDOLGOZÁSA, MEGJELENÍTÉSE

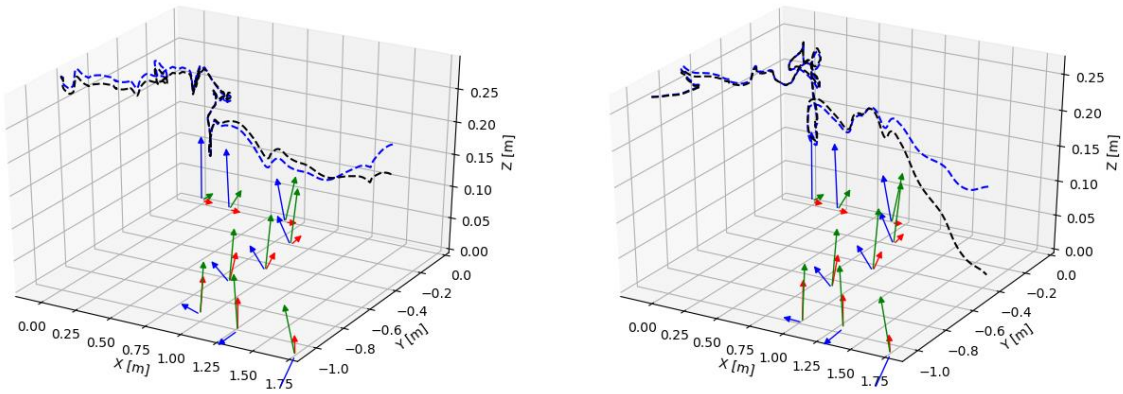
A mérések eredményeit mátrixos formában tárolja az NPZ fájl, így egyszerűen a mátrixok nevei szerint visszaolvashatók. A megjelenítés során a *Matplotlib* programcsomag *Axes3D* osztálya elégíti ki céljainkat. Egy függvény az adatpontok álló megjelenítésére, illetve egy az animációra is készült, melyben a drón szögelfordulásait is ábrázoljuk.

A drón által gyűjtött adatokra egy egyszerű, empirikusan bekalibrált Kálmán-szűrőt alkalmaztunk a mérési zajok tompítására. Az *5. fejezet* 1. mérésének z irányú adatain mutatja a szűrés eredményét a *21. ábra*. A mintán látszik, hogy az adatértékeket zaj terheli, ezt sikerült csökkenteni a Kálmán-szűrő megfelelő hangolásával. Ezen felül terheli még a mérést egyfajta drifthiba, mely a markerek koordináta-rendszereinek egymás után történő felvételéből adódik. Ha tudjuk, hogy a markerek a valóságban egy síkban helyezkedtek el, korrigálhatjuk az eredményeket a marker-középpontokra történő felületillesztéssel.



21. ábra: Az 1. mérés z irányú szűretlen adatpontjai és az alkalmazott Kálmán-szűrő általi közelítés

A korrekcióhoz a SciPy programcsomag *scipy.linalg.lstsq()* függvényét alkalmazzuk, mely a legkisebb négyzetek módszerével illeszt felületet egy ponthalmazra. Ennek előnye, hogy mindenképp ad eredményt, a módszer sikeresen konvergál. Egy másodfokú felület illesztése után rávetíthetjük z irányban a drón pályájának görbéjét a felületre. A vetített pontok z koordinátaértékeivel korrigálhatjuk a mérési pontokat, ahogy az a *22. ábra* görbéin is látszik. A korrekció mértéke az adatpontok vége felé haladva növekszik, mivel a rendszer mérési hibája is így növekszik. Végül ezt az utólagos síkkényszert minden mérési eredményre alkalmazzuk, így jobb magasságértékeket kapva.



3. mérés adatsora **4. mérés adatsora**

22. ábra: A mérések z irányban korrigált értékei (kék) és a Kálmán-szűrt adatsor (fekete) a 3. és a 4. mérés példáján

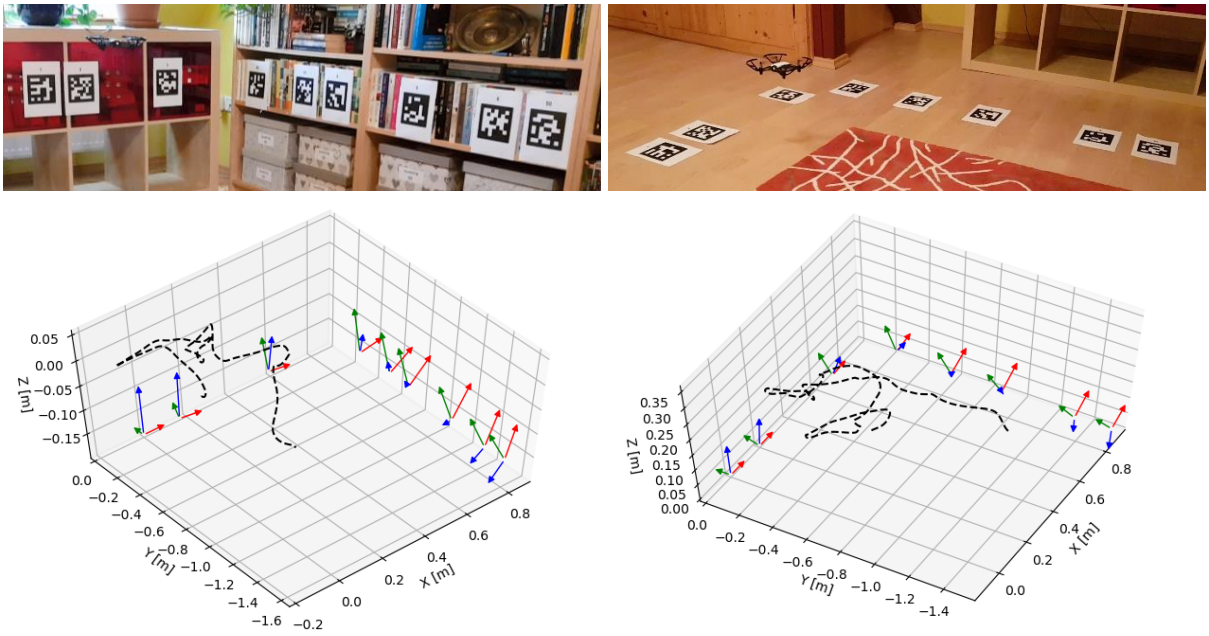
A szűrt értékekből a *matplotlib.animation* csomag *FuncAnimation()* függvényének segítségével videókat is kimenthetünk a mérési eredményeinkről, melyeken a drón forgását is megjeleníthetjük. A videót MP4 formátumban tároljuk el. Az animáció során a drónnak megfelelő koordináta-rendszer átlagolt, „breadcrumb” (kenyérmorzsa) pontokat hagy maga mögött, ezzel mutatva a bejárt pályát.

5. MÉRÉSEK MOTION CAPTURE RENDSZERREL

Az elkészült rendszer értékeléséhez kontrolladatok szükségesek, hogy képet kaphassunk az eredmények használhatóságáról, pontosságáról. Referenciának a MOGI tanszék mozgáslaborját választottuk, melynek *OptiTrack* gyártmányú [30] motion capture rendszere (továbbiakban *MoCap*) kalibrált, pontos adatokat szolgáltat a drónról.

5.1. Otthoni környezetben végzett előzetes tesztek

A tanszéki Motion Capture rendszerrel végzett mérések előtt szükséges volt a program működésének és megbízhatóságának tesztelése. Ennek érdekében otthoni környezetben is reptettük az eszközt kétféle markerezéssel: függőlegesen (falon), illetve vízszintesen (padlón) elhelyezett markerek segítségével. Ezen tesztek során a drón navigációja sikeres volt, ahogy az adatgyűjtés is. A mérési elrendezéseket és az eredményeket a *23. ábra* mutatja. Megjegyzendő, hogy vízszintes markerelhelyezés esetén a drón lassabban áll be a célmarkerhez. Ennek oka, hogy a markerek nagyobb betekintési szög alatt látszanak, így a PnP transzformáció pontatlanabb eredményt ad.



23. ábra: Függőleges (balra) és vízszintes (jobbra) markerelhelyezésben végzett mérések eredményei

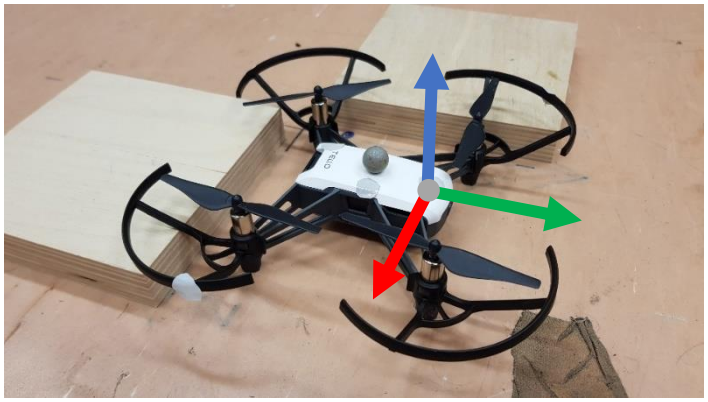
5.2. A rendszerek összehangolása, kalibrálás

A markerekből gyűjtött adatok az 1. sorszámú marker koordináta-rendszerében értelmezettek, ezért a MoCap koordináta-rendszerét is át kell számítanunk a marker szerint. A legegyszerűbb mód, ha az ArUco marker koordináta-rendszerét közvetlenül ráhelyezzük a MoCap koordináta-rendszerének origójára, azonban erre a további vezérlési útvonalak kialakítása miatt nem volt lehetőség. Az origót jelentő 1. markernek mozgathatónak kellett lennie.

Ehhez az 1. ArUco markeren elhelyeztük a MoCap 3 db infratartományban retroreflektív markerét, ezek látszanak a *24. ábra* jobb oldali képén a papírlap három sarkában. Ebből a szoftverben szilárd testet (Rigid Body) alkottunk, melynek koordináta-rendszer középpontját (Pivot Point) eltoltuk egy gömbmarker segítségével. Még a koordinátabázist kellett elforgatni, először y tengely körül -90 fokkal, aztán x tengely körül -90 fokkal, így kapjuk meg a drónprogram által alkalmazott koordináta-rendszert. A két koordináta-rendszer a 24. ábra képein látszik.

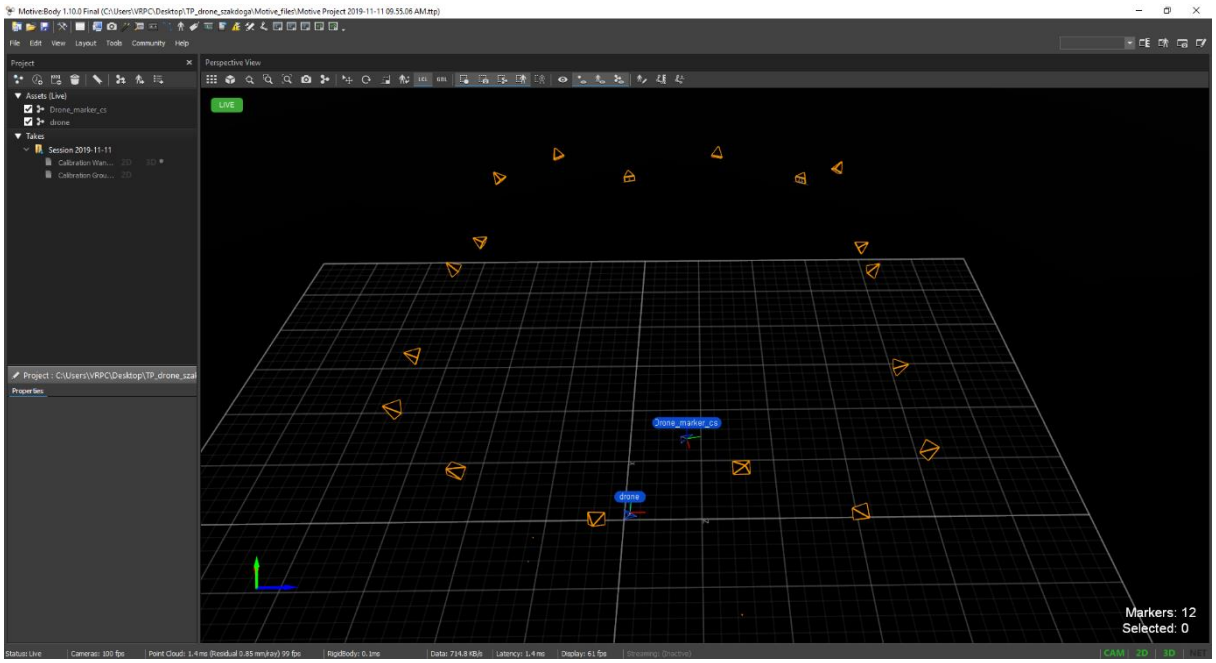
24. ábra: Balra a MoCap, jobbra a marker koordináta-rendszere
(vörös: x tengely, zöld: y tengely, kék: z tengely)

A koordináta-rendszerek beállítása után a drón 6 matricával történő felmarkerezése történt. A tengelypárhuzamosság érdekében a MoCap markereivel megjelölt drónt is a 24. ábra képein látható két fahasáb segítségével állítottuk be. A MoCap rendszerben a drón koordináta-rendszerének Pivot Pointját az eszköz kamerájának középpontjába igyekeztünk eltolni. (*25. ábra*) Ennek pontos kimérése a drón kialakítása miatt nem lehetséges, csak becsült értékekkel dolgoztunk. Így a drónt is Rigid Bodyként kezelve, létrejött a MoCapben is egy leképezés, melynek adatait mérhetjük.



25. ábra: A MoCap rendszer számára felmarkerezett drón krsz.-e

A kalibrált koordináta-rendszerek az OptiTrack rendszer programjában a *26. ábra* képernyőképének megfelelően látszódtak a kalibrálás után. A MoCap rendszer méréseit át kell számítani a *„Drone_marker_cs"*, azaz az 1. ArUco marker koordináta-rendszerébe. Ezt az adatpontokon végzett homogén transzformációval érhetjük el a *(3.5.) egyenlet* szerint. A mérési pontokat elforgatjuk és eltoljuk a *„Drone_marker_cs"* rendszerbe, hogy a drón AR-marker szerinti méréseivel megegyezzenek.



26. ábra: Az OptiTrack rendszer szoftverében a bekalibrált koordináta-rendszerek

A mérési adatok kimentése CSV-fájlba történt. Mivel a MoCap rendszer mintavételezése maximum 120 FPS lehet, azonban a drón kamerája legjobb esetben is csak 25 FPS-re képes, ezért a MoCap mintavételezését 100 FPS-re állítottuk, így négyszer annyi mért adatot kaptunk. Az adatpontok szinkronizálása is csak utólag megoldott, ugyanis a drón programjába már nem került bele az OptiTrack szoftver vezérlése, így nem pontosan egy pillanatban kezdődik a két mérés.

5.3. A mérések összevetése, ArUco markerelrendezések

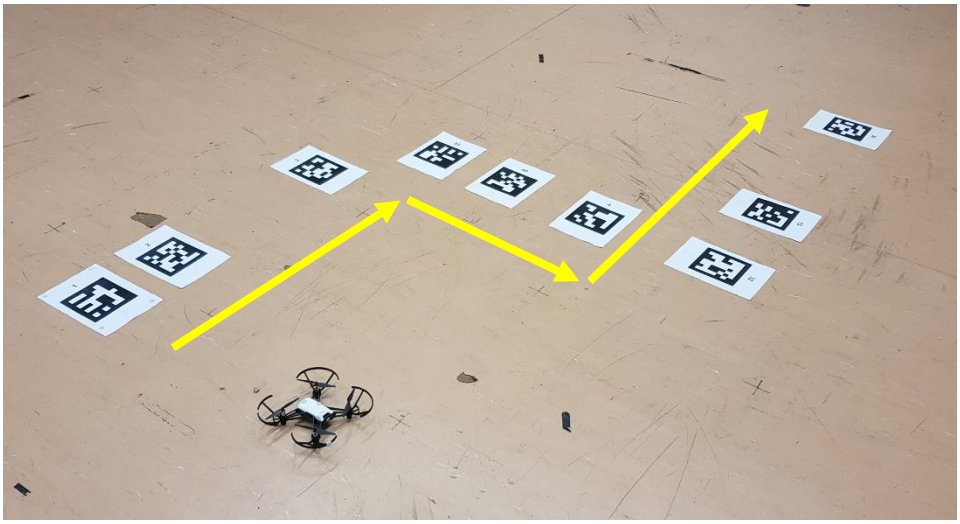
Párhuzamos mérések során hatszor reptettük a drónt, kétféle pálya mentén. A mérések közül csak öt lett felhasználható, mert az egyik mérés során egy markert rosszul helyeztünk fel. Onnantól a drón rossz irányba fordult és nem érte el az ArUco végmarkert, így a program mérési eredményt sem szolgáltatott. Megjegyeznénk továbbá, hogy a mozgáslabor megvilágítása valószínűleg nem kedvezett a drón kameraképének, ugyanis sokkal többször vesztette el a felhelyezett ArUco markereket, mint az előzetes, otthoni tesztkörnyezetben. Emiatt a drón beállásai lassabbak voltak, illetve több helyen hiányosak a mérések.

5.3.1. Első markerútvonalon végzett mérések

Az ArUco markereket típusaiknak megfelelően helyeztük fel, a köztük lévő távolságok az előzetes kísérletezés során alakultak ki. Szükséges a padlón való rögzítésük, különben a drón rotorjai által keltett légáramlat elfújná őket. A *27. ábra* fotóján látszik az útvonal elrendezése. Az elrendezésben használt markereket, jelentésükkel együtt a *3. táblázat* tartalmazza.

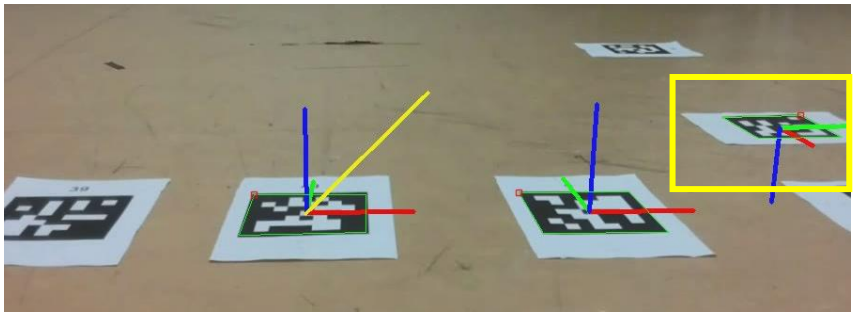
3. táblázat: Az első mérési elrendezésben alkalmazott markerek és jelentésük

Marker sorszám	Jelentés	Célvektor ([m] [m] [m] [°])
1	Origó	[0 0 0,9 0]
2	Jobb egyenes	[0 0 0,9 −45]
3	Jobb egyenes	[0 0 0,9 −45]
39	Jobb fordulat 1	[0 0 0,9 5]
40	Jobb fordulat 3	[0 0 0,9 −20]
4	Jobb egyenes	[0 0 0,9 −45]
13	Bal egyenes	[0 0 0,9 45]
32	Bal fordulat 3	[0 0 0,9 20]
50	Végmarker	[0 0 0,9 0]



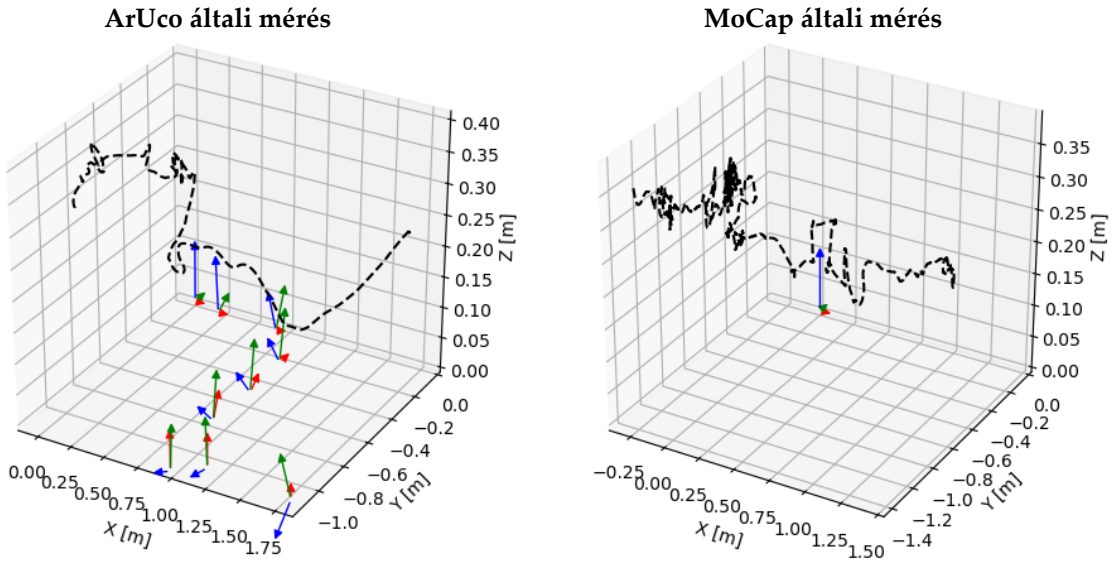
27. ábra: Az első, méréshez kialakított drónútvonal az ArUco markerekből

A drón háromszor is sikeresen bejárta a kialakított útvonalat. Sajnos, a mérései zajosabbak lettek, mint az előzetes tesztek során, a már említett vízszintes markerek roszszabb láthatósága, a labor megvilágítási viszonyai és a wifi kapcsolat jel/zaj viszonya miatt. A perspektív PnP transzformáció egyik hátránya, az éles szögben látott AR markerek esetén, hogy az orientáció átfordulhat és ekkor helytelen adatokat ad. Erre példa a *28. ábra* jelölt markere, melynek tengelyei helytelen orientációban állnak. Jelen esetben ezt éppen leszűrte a sarokpontokra való szűrés, ugyanis egyik sarokpontja a kamerakép szélére esik.

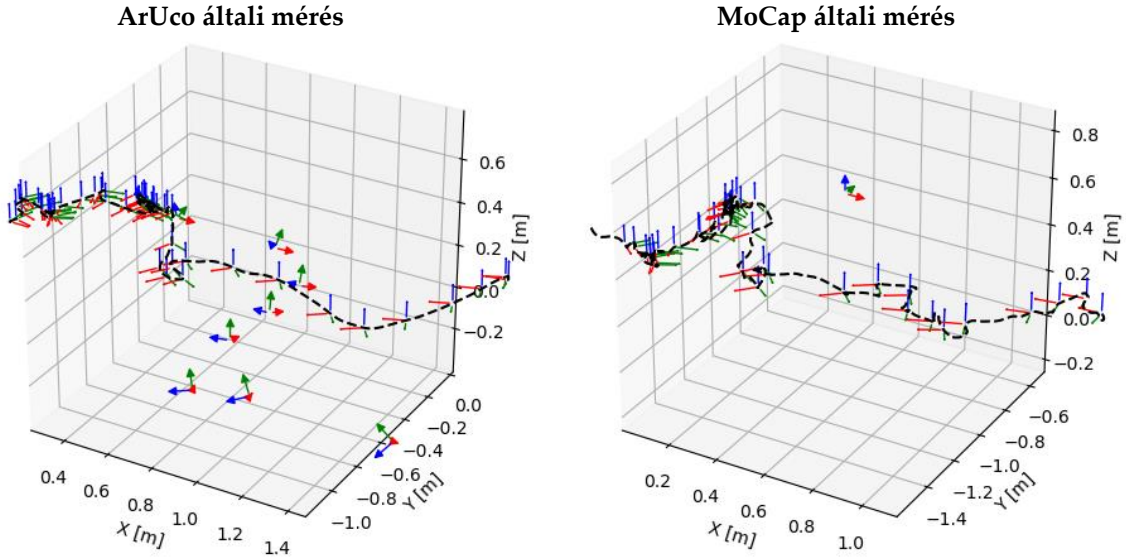


28. ábra: A jelölt marker z tengelye nem felfelé, hanem a képből kifelé és lefelé mutat, mintha a marker a plafonon helyezkedne el.

A MoCap rendszer adatai nem estek át szűrésen, tekintsük őket pontos referenciának! Hogy a mérések frekvenciája közel egységes legyen, a MoCap 100 Hz-cel mintavételezett adatsorából csak minden 4. elemmel dolgozunk, bár drón kamerából történő mérése csak elméletben ad 25 Hz-es mintavételezéssel értéket. Így az adatpontok már közel azonos sűrűségűek. A drón által mért adatok szűrt értékei pontatlanok a MoCap adataihoz képest, azonban jellegre helyesek. Az adatsorokra alkalmaztuk a 22. ábra adatsorain is látható drifthiba korrekcióját z irányban. A *29. ábra* két megjelenített adatsorán látszik, hogy a tengelyek tartománya közelítőleg megegyezik, ahogy a drón által bejárt útvonal jellege is. A drón szöghelyzeteinek megjelenítését mutatja a *30. ábra* a drónútvonal egy kimetszett részletén. Jól látszik, hogy a rendszerünk által eltárolt szöghelyzetek megegyeznek a MoCap által mértekkel.

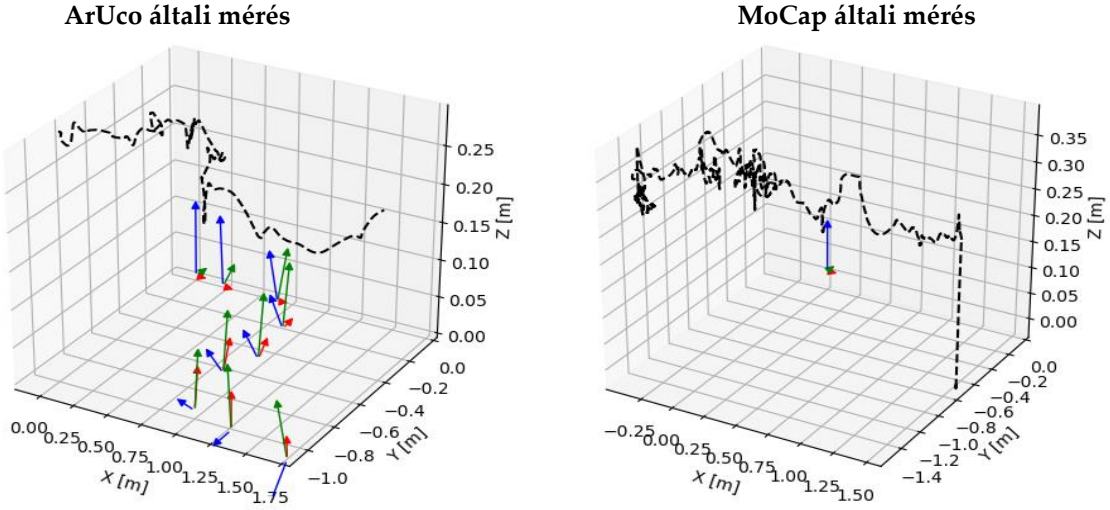


29. ábra: Az 1. mérés eredményei a drón- és a MoCap-rendszer alapján

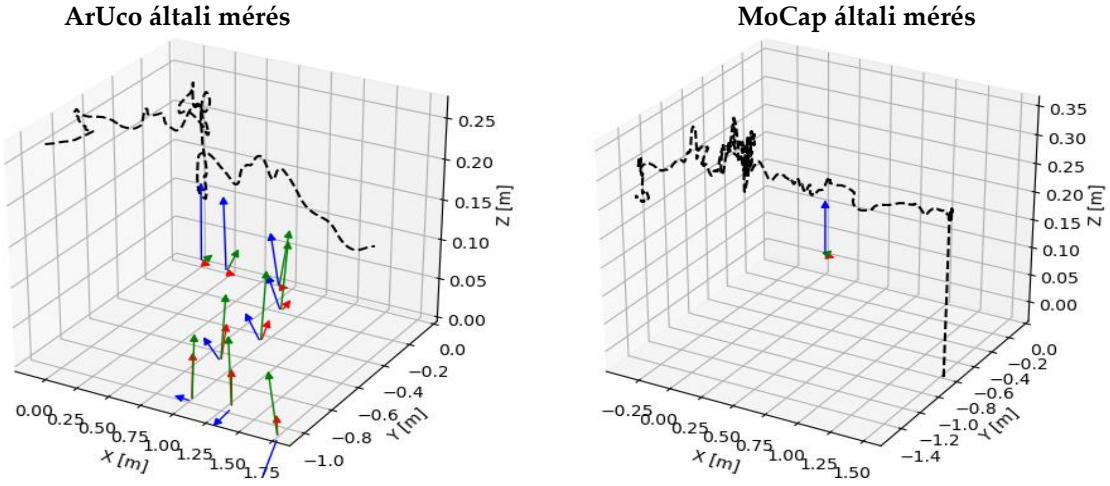


30. ábra: Az 1. mérés adatpontjainak egy részletén elhelyezett, a drónnak megfelelő orientációjú koordinátabázisok

A 2. mérés adatai nem készültek el, egy rosszul felhelyezett marker miatt, de a 3. mérés már zavartalanul lefolyt. Egyetlen hibája a koordinátarendszerek z irányú tárolási hibája volt, melyet a már említett kényszerezéssel sikeresen korrigálhatunk. Hasonlóan tehetünk a 4. mérés során is, hogy az adatpontokra teljesüljön a drón ismert magasságtartása. A *31. ábra* és *32. ábra* eredményein látszik, hogy a korrigált AR markeres mérési pontok jellege megegyezik a MoCap eredményével. Igaz, így is adódik magasságbeli különbség, de már nem jelentős mértékű.



31. ábra: A 3. mérés adatsorai (a MoCap-ben a drón leszállása is rögzítve lett)



32. ábra: A 4. mérés adatsorai (a MoCap-ben a drón leszállása is rögzítve lett)

A drifthibák sikeresen korrigálhatók a felületillesztéses módszerrel, így útófeldolgozás során megfelelő eredményt kapva. Ez a probléma a *3.5.1 fejezetben* megfogalmazott várakozásaink között is szerepelt. Valós időben működő megoldást a jobb hardver alkalmazása, az ArUco markerfelismerő algoritmus javítása és jobb környezeti viszonyok jelenthetnek.

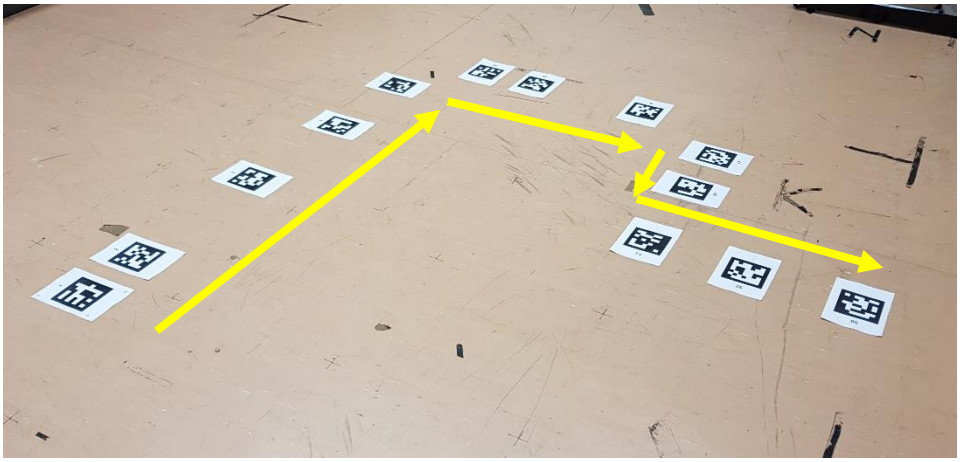
A markerek koordináta-rendszereinek orientációja is jelentős hibával terheltnek látszik, azonban ezek a Matplotlib nem egyentartományú tengelymegjelenítésének köszönhetően torzulnak. Ezért látszanak például a 32. ábra koordináta-rendszer nyilai függőleges irányban hosszabbnak.

5.3.2. Második markerútvonalon végzett mérések

A második mérési elrendezésben 13 db AR markert használunk, ezek jelentését sorrendben a 4. táblázat mutatja, míg elhelyezésük a *33. ábra* fotóján látható. Itt a nagyobb alkalmazott darabszám miatt valószínűleg nagyobb lett az előző bekezdésekben taglalt drifthiba a végmarkerek esetén, de ezekre is alkalmazhatjuk az utólagos síkkényszerezési korrekciót.

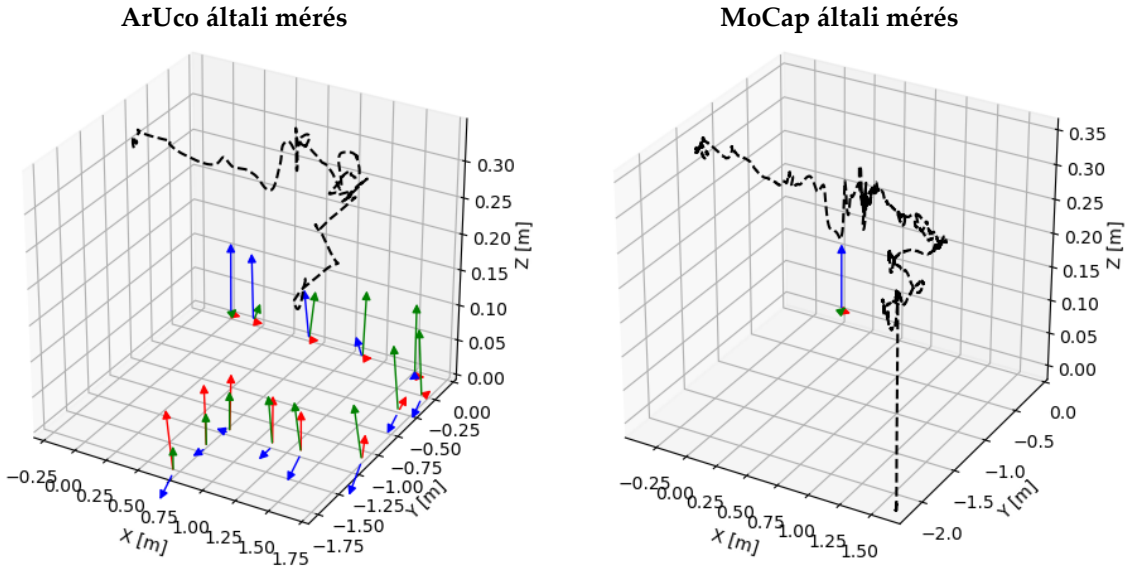
4. táblázat: A második mérési elrendezésben alkalmazott markerek és jelentésük

Marker sorszám	Jelentés	Célvektor ([m] [m] [m] [°])
1	Origó	[0 0 0,9 0]
2	Jobb egyenes	[0 0 0,9 −45]
3	Jobb egyenes	[0 0 0,9 −45]
4	Jobb egyenes	[0 0 0,9 −45]
5	Jobb egyenes	[0 0 0,9 −45]
39	Jobb fordulat 1	[0 0 0,9 5]
40	Jobb fordulat 3	[0 0 0,9 −20]
6	Jobb egyenes	[0 0 0,9 −45]
41	Jobb fordulat 1	[0 0 0,9 5]
42	Jobb fordulat 3	[0 0 0,9 −20]
13	Bal egyenes	[0 0 0,9 45]
32	Bal fordulat 3	[0 0 0,9 20]
50	Végmarker	[0 0 0,9 0]



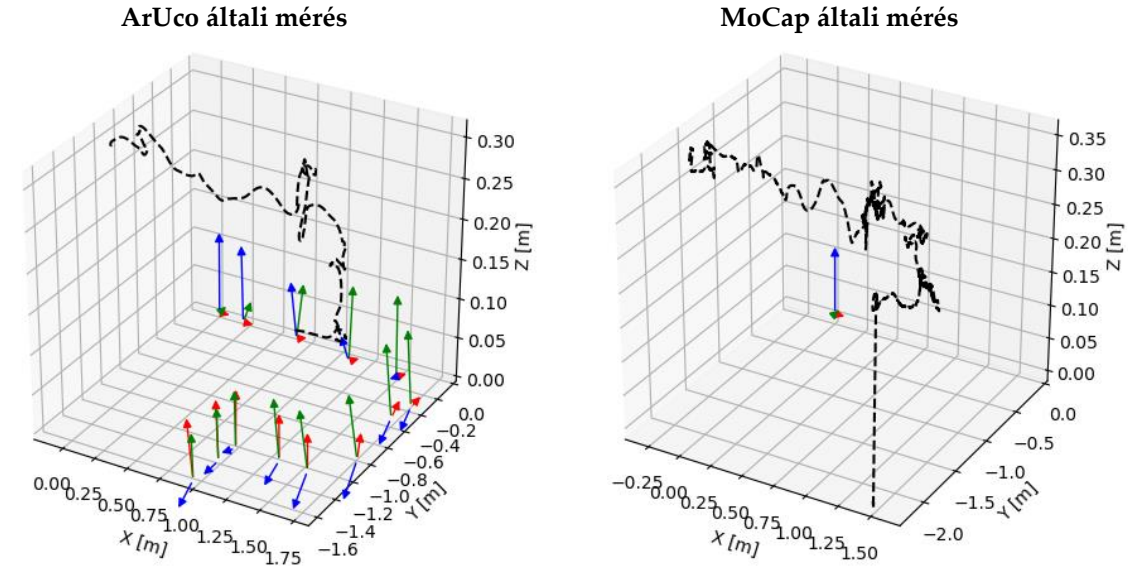
33. ábra: A második, méréshez kialakított drónútvonal az ArUco markerekből

Az 5. mérés háromdimenziós adatain látszik talán a legjobban, a két módszerrel kapott repülési görbe jellegének egyezése. Az első markerhez való beállás, illetve az első balkanyar szinte egyhelyben történő mozgása a *34. ábra* két adatsorát összevetve megfigyelhető. A tengelyek megjelenítési tartománya is körülbelül egyezik, a MoCap esetén egyes kiugró értékek növelik például z irányban a tartományt. A markerek középpontjainak egymáshoz képesti elhelyezkedése is nagyban egyezik a 33. ábra fotóján láthatóval.



34. ábra: Az 5. mérés adatsorai (a MoCap-ben a drón leszállása is rögzítve lett)

A 6. mérés során sokkal kisebb drifthiba keletkezett, az utólagos síkkényszer előnye ilyenkor, hogy kisebb mértékben vagy szinte egyáltalán nem korrigál az adatsorokon. A markerek origói hozzávetőlegesen egy síkban helyezkednek el, így a mérési pontok alapból megfelelők voltak. A *35. ábra* két adatsorán látszik, hogy a mért pályák jellege ezen mérés során is igen hasonló: az első bal fordulásnál látszik mindkét esetben az egyhelyben ragadás, ahogy a mérés vége előtt is. A 6. és az 5. mérés adatait összevetve jól látszik, hogy majdnem ugyanazt az útvonalat járta be a drón. Talán az 5. mérés AR-adatsora egy kicsivel nagyobb bizonytalansággal rendelkezik, de jól látszanak mindkét mérési soron a kritikus navigációs pontok, ahol a drón fordulatot tesz. Az értékek pontosságát nagyban javítja, ha több markert egyszerre látva, átlagoljuk a kamerából mért értékeket, ahogy azt előzetesen az *Babinec et al.* [17] tanulmány alapján is megfogalmaztuk.

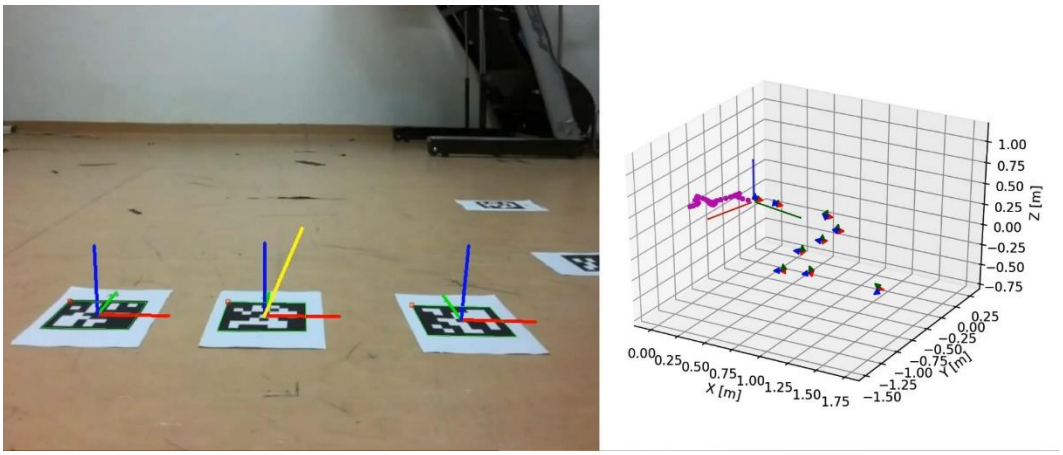


35. ábra: A 6. mérés adatsorai (a MoCap-ben a drón leszállása is rögzítve lett)

5.4. A mérések videós kiértékelése

Ahogy már a kialakított program részletezése során említettük, a drón által látott képet automatikusan rögzíti a rendszer. Utólag pedig a gyűjtött adatpontokat és szögelfordulásokat egy animációban jeleníthetjük meg, egyentartományú tengelyek használatával. A szakdolgozathoz csatolt DVD *2. mellékletében* szerepel a mérések videója.

Az első mérés videós eredményéből mutat egy párhuzamos képkockát a *36. ábra*. A drónt egy koordinátabázis jelöli, irányai a dolgozat elején leírtaknak felelnek meg, így a zöld színű, *y* tengely mutat a drón orrából kifelé. Látszik, ahogy a videón éppen a középső AR-markerhez vezérel a drón, de a két másik látott markerből is gyűjt adatokat és három marker méréseit átlagolja. A drón orientációjának tárolása teljes mértékben megfelel a rögzített videón látottakkal, esetleges elcsúszás az animáció frissítési frekvenciájához való közelítés és a rosszul összeszerkesztett videók miatt lehetséges. Általánosan megállapítható, hogy a Tello drón szenzora szűretlen szöghelyzetértékek esetén is megfelelő adatokat szolgáltat.



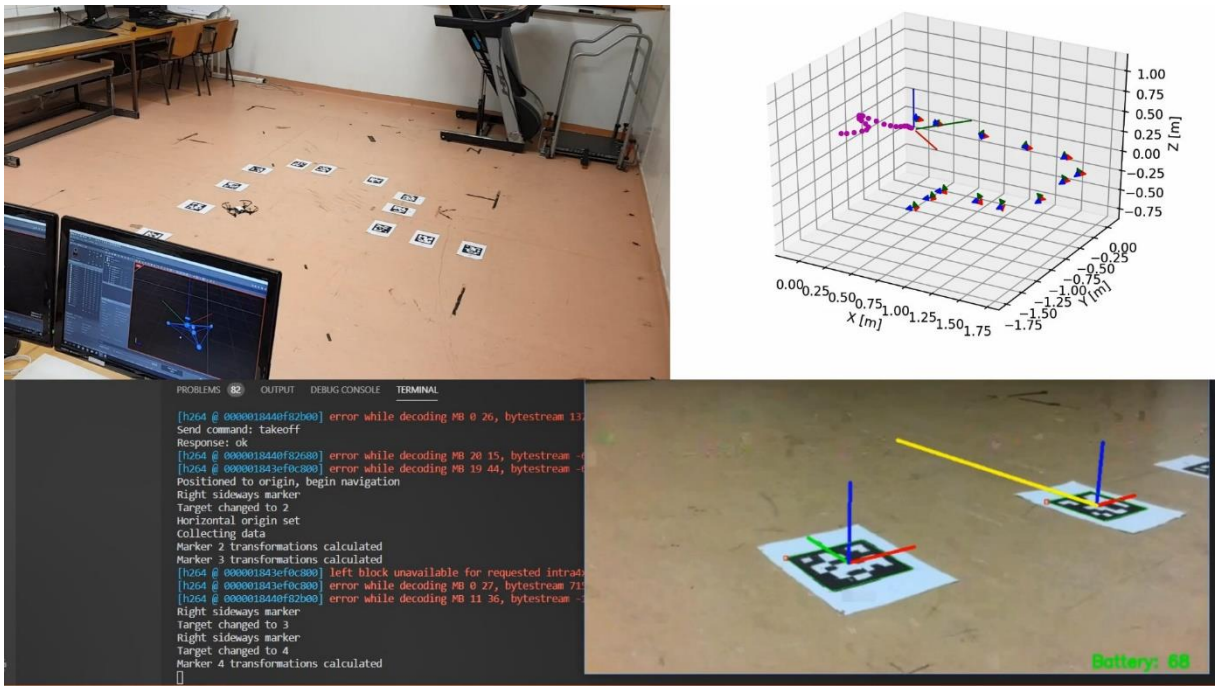
36. ábra: Az 1. mérés során rögzített videó és az animált repülési útvonal azonos időpillanatban rögzített képkockája egymás mellett

Az animáció során a mentési időpillanatokat használó tömböt is alkalmazzuk. A Matplotlib animációjának ugyan időfrissítést nem lehet betáplálni, de az lehetséges, hogy az ismert FPS érték mellett egyes képkockákat hosszabb ideig jelenítsünk meg. A mérések során eltárolt idővektor adataiból látszik, hogy a mérési adatpontok tárolását átlagban 0,07 másodperc alatt végezte a rendszer. Ez lassabb, mint a 25 FPS mellett megkövetelt 0,04 másodperc, azonban ez az elméleti érték a képi megjelenítés során sem teljesült a drón teljesítményéből adódóan. A lassulás további oka lehet az egy ciklus alatt végzett mátrixműveletek száma. Például a markerhelyzetek eltárolása során sokkal több számítást végez a program, mint amikor már csak tárolt markerekadatokból dolgozik.

A drón útvonalát a Kálmán-szűrt adatokból jelenítjük meg, a „breadcrumb” pontok hozzáadása 12 adatpontonként történik. Elméletben 0,48 másodpercnek megfelelő mérési időközönként jelenik meg egy útvonalpont az animáción, ám ez az előző bekezdésben taglaltak alapján nem igaz, hosszabbak az időközök.

37. ábra: Dekódolási hiba által okozott képzaj a videóadás során

A 6. mérés során a konzolon közölt üzenetek és a drón repülése külső nézőpontból is rögzítve lett. Ezeket egymás mellé helyezve mutatja a *38. ábra*. A konzolüzenetek között látható pár videóadás dekódolási hiba. Erre a hibára mutat példát a *37. ábra* képkockája, ilyenkor egyetlen marker sem felismerhető a kép zajossága miatt, a ciklus üresen fut. Ezt a wifis kapcsolat jel/zaj viszonyának romlása idézi elő az eszköznél, amelynek kialakítása és teljesítménye mindössze ilyen képsugárzási megbízhatóságot tesz lehetővé. Az adatgyűjtő algoritmus sikeresen átlendül ezeken a hibákon és pár hiányzó adatponttól eltekintve nem terheli hiba a mérést.



38. ábra: A 6. mérés során rögzített képernyőfelvétel, videó és mérési pontok párhuzamosan

6. ÖSSZEFOGLALÁS

6.1. Eredmények értékelése

A kialakított rendszer, teszteredményei alapján, alkalmas a drón vezérlésére, megfelelően kialakított környezetben. A drón pozíciómérései azonban bizonytalanok, egyes esetekben nagy hibával terheltek, más esetekben pedig megfelelően pontos értékeket szolgáltatnak. Mivel a rendszer kamerakép elvén működik, a környezeti fényviszonyok is befolyásolják működését.

A program megírása során cél volt, hogy a rendszer minél általánosabban és adaptívabban képes legyen működni. Természetesen ehhez szükséges a megfelelően felmarkerezett repülési környezet kialakítása. Amennyiben a megfelelő markereket helyeztük fel, a drón az öt mérésből ötször képes volt végignavigálni a kialakított pályán. Ellenben egy hibás markerfelhelyezés, ahogy az a 2. mérés során is előfordult, már a navigáció és a mérés leállását vonja maga után. Jelenleg talán problémát jelenthet, hogy a félkész adatokról nem kapunk mentést, azonban a szakdolgozat egyik célja volt, hogy a drón automatikus működésre legyen képes, ahol az adatgyűjtés feltétele, hogy a drón végighaladjon a pályán.

A markerek koordináta-rendszerei közötti transzformációk kiszámítása során hibák keletkeznek, amelyek körülbelül egyenlő eséllyel korrigálhatják, vagy növelhetik egymás hibáját. A rendszer ezen része még bizonytalan, megfelelő korrekciója kidolgozásra szorul. A transzformációk megállapítása után, a két kapott kamerapozíció átszámítható a markerekéből a globális koordináta-rendszerbe. Ezek eredményein látható, hogy a kapott pozíciók eltérése ±0,05 méter tartományba esik, ahogy azt az *5. táblázatba* kigyűjtött 9 db számítási eredmény is jól mutatja. Az eltérések körülbelül az ArUco markerek mérési hibáival esnek egybe, azonban az ilyen eltérések az átszámítások során halmozódnak. Érthető, hogy nagyobb hibákat tapasztalhatunk a mérések végén, mikor már akár 12 marker értékein keresztül számoljuk a globális koordinátát.

5. táblázat: Két szomszédos marker értékeiből számolt globális koordináta-vektorok
(A vektorok értékei páronként kell megegyezzenek.)

$\mathbf{v_{1_{m0}}}$[m]	$\mathbf{v_{1_{n0}}}$[m]	$\mathbf{v_{2_{m0}}}$[m]	$\mathbf{v_{2_{n0}}}$[m]	$\mathbf{v_{3_{m0}}}$[m]	$\mathbf{v_{3_{n0}}}$[m]
$\begin{bmatrix}-0{,}3736\\0{,}2307\\0{,}7553\end{bmatrix}$	$\begin{bmatrix}-0{,}3816\\0{,}2160\\0{,}7580\end{bmatrix}$	$\begin{bmatrix}-0{,}4002\\0{,}2180\\0{,}7498\end{bmatrix}$	$\begin{bmatrix}-0{,}4004\\0{,}2207\\0{,}7444\end{bmatrix}$	$\begin{bmatrix}-0{,}4510\\0{,}2209\\0{,}7415\end{bmatrix}$	$\begin{bmatrix}-0{,}4690\\0{,}2141\\0{,}7164\end{bmatrix}$
$\mathbf{v_{4_{m0}}}$[m]	$\mathbf{v_{4_{n0}}}$[m]	$\mathbf{v_{5_{m0}}}$[m]	$\mathbf{v_{5_{n0}}}$[m]	$\mathbf{v_{6_{m0}}}$[m]	$\mathbf{v_{6_{n0}}}$[m]
$\begin{bmatrix}-0{,}0762\\0{,}0986\\1{,}0246\end{bmatrix}$	$\begin{bmatrix}-0{,}0865\\0{,}0443\\1{,}0374\end{bmatrix}$	$\begin{bmatrix}-0{,}0138\\0{,}0464\\0{,}8750\end{bmatrix}$	$\begin{bmatrix}0{,}0023\\0{,}0304\\0{,}8749\end{bmatrix}$	$\begin{bmatrix}0{,}0165\\0{,}0344\\0{,}4578\end{bmatrix}$	$\begin{bmatrix}0{,}0172\\0{,}0325\\0{,}4528\end{bmatrix}$
$\mathbf{v_{7_{m0}}}$[m]	$\mathbf{v_{7_{n0}}}$[m]	$\mathbf{v_{8_{m0}}}$[m]	$\mathbf{v_{8_{n0}}}$[m]	$\mathbf{v_{9_{m0}}}$[m]	$\mathbf{v_{9_{n0}}}$[m]
$\begin{bmatrix}0{,}1657\\0{,}0114\\0{,}5741\end{bmatrix}$	$\begin{bmatrix}0{,}1740\\0{,}0320\\0{,}5812\end{bmatrix}$	$\begin{bmatrix}0{,}1098\\-0{,}0285\\0{,}5635\end{bmatrix}$	$\begin{bmatrix}0{,}1131\\-0{,}0289\\0{,}5532\end{bmatrix}$	$\begin{bmatrix}0{,}1547\\-0{,}0084\\0{,}3571\end{bmatrix}$	$\begin{bmatrix}0{,}1452\\-0{,}0540\\0{,}3892\end{bmatrix}$

Az esetleges hibás mérésekre megoldást jelenthet az ArUco markerfelismerés optimalizálása, javítása. Ebbe beletartozik a marker széleinek jobb érzékelése (20. ábra), mely szintén okozhat mérési hibát, illetve a PnP transzformáció szűrése a tengelyátfordulások kiküszöbölésére (28. ábra). Az ezekhez hasonló hibák helytelen mérési pontokat eredményeznek, melyek az algoritmus javításával kiküszöbölhetők lehetnének. Már vannak is javított algoritmusok, melyek jobb szűrést ígérnek bizonyos programkörnyezetek (pl. ROS) alatt. [31]

Szintén továbbfejlesztésre és szűrésre szorul a szöghelyzetek megállapítása, illetve Euler-szögekké alakítása. A forgásmátrixból való átalakítás [32] során használt *atan2()* függvény 180 fokos periodicitása miatt az eredmények szűrés nélkül sajnos használhatatlanok lettek. A szűretlen forgásmátrixokkal való transzformációk is okozhatnak mérési hibákat, melyeket az alkalmazott átlagolás nem tud korrigálni.

A rendszer, hibái ellenére képes bizonyos mértékig megfelelően számítani a drón röppályáját, automatikusan vezérelni a drónt a markerek között és összesíteni a gyűjtött adatokat. A drónból az UDP-szerveren keresztül kiolvasott adatok már eleve megfelelően szűrtek, jól mérhető a szögelfordulások értékeivel például a drón orientációja. Az erre megírt kiolvasó program külön szálon fut, nem terheli a fő vezérlést, csak a háttérben gyűjt adatokat.

Az animáció megjelenítése során kiderült, hogy a ciklusonkénti számolás az adatok eltárolását átlagban 0,07 másodperc alatt végezte el, mely lassabb az alkalmazott 25 FPS mellett futó videóadástól. Ez a lassabb adatgyűjtés is okozhat hibát, ha máshol nem, akkor a megjelenítés során. A mellékelt videók 0,04 másodperces képfrissítéssel jelenítenek meg mérési adatokat, egyes adatpontokat így hosszabb ideig szükségesek mutatni.

A használt Tello drón méreteiből adódóan nem profi eszköz, a wifi kapcsolata sokszor nagy zajjal terhelt, illetve legnagyobb problémája a kamerakép közel 2 másodperces késleltetése. Ezek ellenére az állapotkiolvasás zökkenőmentesen zajlik. Bár néhány szenzor értékei a tapasztalatok szerint olykor helytelen adatokat szolgáltathatnak - így a ToF kamera magasságértéke is -, a drón kialakításához képest megfelelő a szenzorok működése. A vezérlési módok közül az alkalmazott RC-irányítás a sebességértékekkel megfelelő pozícionálásra képes, nagyobb hibát okozott a már említett késleltetés.

Egy beltéri mikrodrón vezérlésére megfelelő a program, sikeresen képes volt a kitűzött célok megvalósítására. Pontossága mind a program valós idejű, akár képi, akár pozícióadatokból nyert szűrésének hiányából, mind a hardver hiányosságai miatt nem minden esetben megfelelő. Azonban megállapítható, hogy a drón nagy biztonsággal teljesítette a pályát, teljesen automatikus navigációt alkalmazva. A kialakított rendszer akár nagyobb méretekben, ipari környezetben is alkalmas lehet navigációra. Például egy csarnokon belüli drónvezérlés esetén, ahol a markerek egymáshoz viszonyított helyzete ismert és a drón a látott markerek alapján tájékozódik és gyűjt adatokat a csarnokon belül.

6.2. Továbbfejlesztési javaslatok

Mint azt már az előző bekezdésben is megfogalmaztuk, egy valós idejű szűrés mindenképp szükséges. A mérési adatpontok utólagos szűrése nem elegendő, ha a markerek koordináta-rendszereinek tárolásában hiba van. Ha egy valós időben működő Kálmán-szűrőt sikerülne beállítani, az után akár az adatpontok azonnali megjelenítése is lehetséges, hogy közel real-time megoldással megjeleníthessük a drón pályáját.

A valós idejű működéshez szükség van még az Nvidia CUDA programcsomagjára, melynek segítségével feltölthetők a mátrixok a számítógép grafikus kártyájára és azon számolva gyorsabban elvégezhetők a mátrix- és képfeldolgozási műveletek. Ez jelenleg a Python programnyelv alatt még nincs kidolgozva, a közeljövőben viszont elkészülhetnek az OpenCV Python-verziójához is a linkelő könyvtárak. Alternatívaként át lehet dolgozni az egész programot C++ vagy C# nyelvre, ahol már létezik CUDA-összeköttetés az OpenCV-vel.

A szakdolgozat csak viszonylag olcsó, akár otthoni használatra is beszerezhető drónnal dolgozik. Ugyanúgy, ahogy az ArUco markerekből történő helyzetmeghatározás is „low-cost" módszernek tekinthető. Természetesen jobb hardver esetén jobb működés is érhető el, ehhez azonban ipari- vagy saját építésű drón szükséges. Így kiküszöbölhető a kamerakép késése a megfelelő akadályelkerülés érdekében. Gyorsabb vezetéknélküli kapcsolat esetén közel azonnal megérkezhetne a sugárzott adás és a drón vezérlése is nagyobb sebességen működhetne. Ugyanezért a szabályozó beállási hibája is jelentősen csökkenthető lenne.

Egy komolyabb alkalmazás során az elkészült rendszert külső adatokkal lehetne támogatni. Így például egy ismert térképet alkotni a markerek egymáshoz viszonyított helyzetéről, hogy a drón ismerje őket, mint viszonyítási pontokat. E módszer hosszabb előkészítést igényel, de hosszabb távon alkalmasabb lehet navigálásra, mivel a markerek helyzetei ismertek, nem terheli őket mérési hiba. Ipari környezetben való alkalmazásra ez lenne megfelelő.

7. FELHASZNÁLT FORRÁSOK

1. ROBERTO SABATINI, SUBRAMANIAN RAMASAMY & ALESSANDRO GARDI (2015): *LIDAR Sensor Based Obstacle Avoidance System for Manned and Unmanned Aircraft*. Journal of Science and Engineering. 4. 1-13.
2. VERŐNÉ WOJTASZEK MALGORZATA (2010): *Fotointerpretáció és távérzékelés 3., A lézer alapú távérzékelés,* Nyugat-magyarországi Egyetem, Székesfehérvár
3. NILS GAGEIK, PAUL BENZ & SERGIO MONTENEGRO (2015): *Obstacle Detection and Collision Avoidance for a UAV With Complementary Low-Cost Sensors*. IEEE Access. 3. 1-1. 10.1109/ACCESS.2015.2432455.
4. SEBASTIAN SCHUON, CHRISTIAN THEOBALT, JAMES DAVIS & SEBASTIAN THRUN (2008): *High-quality scanning using time-of-flight depth superresolution.* CVPR Workshop on Time-of-Flight Computer Vision. 1 - 7. 10.1109/CVPRW.2008.4563171.
5. REFAEL WHYTE, LEE STREETER, MICHAEL CREE & ADRIAN DORRINGTON (2015): *Application of lidar techniques to time-of-flight range imaging*. Applied Optics. 54. 9654. 10.1364/AO.54.009654.
6. ASHUTOSH SAXENA, SUNG HEE CHUNG & ANDREW NG (2008): *3-D Depth Reconstruction from a Single Still Image*. International Journal of Computer Vision. 76. 53-69. 10.1007/s11263-007-0071-y.
7. ALI AMIRI, SHING YAN LOO & HONG ZHANG (2019): *Semi-Supervised Monocular Depth Estimation with Left-Right Consistency Using Deep Neural Network.* 1905.07542v1
8. ASHUTOSH SAXENA, JAMIE SCHULTE & ANDREW NG (2007): *Depth Estimation Using Monocular and Stereo Cues*. IJCAI International Joint Conference on Artificial Intelligence. 2197-2203.
9. CLÉMENT PINARD (2019): *Robust Learning of a depth map for obstacle avoidance with a monocular stabilized flying camera*. NNT: 2019SACLY003, Université Paris-Saclay, Saint-Aubin, France
10. ROSITSA BOGDANOVA, PIERRE BOULANGER & BIN ZHENG (2016): *Depth Perception of Surgeons in Minimally Invasive Surgery*. Surgical Innovation. 23. 10.1177/1553350616639141.
11. ANDREW J. BARRY, HELEN OLEYNIKOVA, DOMINIK HONEGGER, MARC POLLEFEYS & RUSS TEDRAKE (2015): *Fast Onboard Stereo Vision for UAVs*, MIT
12. XIAOYU CUI, KAH BIN LIM, YUE ZHAO, AND WEI LOON KEE (2014): *Single-lens stereovision system using a prism: position estimation of a multi-ocular prism*. J. Opt. Soc. Am. A 31, 1074-1082
13. H. ALVAREZ, L.M. PAZ, JÜRGEN STURM & D. CREMERS (2016): *Collision Avoidance for Quadrotors with a Monocular Camera.* 10.1007/978-3-319-23778-7_14.

14. MOHAMMAD FATTAHI SANI & GHADER KARIMIAN (2017): *Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors*. 102-107. 10.1109/ICONDA.2017.8270408.
15. KONSTANTIN YAKOVLEV, VSEVOLOD KHITHOV, MAXIM LOGINOV & ALEXANDER PETROV (2015): *Distributed Control and Navigation System for Quadrotor UAVs in GPS-Denied Environments*. 10.1007/978-3-319-11310-4_5.
16. RÉVÉSZ LEVENTE, SZEPESSY TAMÁS (2019): *Gépi látás alkalmazása az üvegházi paradicsomtermesztés területén*, Mechatronika projekt feladat, Mechatronika, Optika és Gépészeti Informatika Tanszék, Budapesti Műszaki és Gazdaságtudományi Egyetem
17. ANDREJ BABINEC, LADISLAV JURIŠICA, PETER HUBINSKÝ & FRANTIŠEK DUCHOŇ (2014): *Visual Localization of Mobile Robot Using Artificial Markers*. Procedia Engineering. 96. 10.1016/j.proeng.2014.12.091.
18. DAMIÀ FUENTES ESCOTÉ (2018): *DJITelloPy*, https://github.com/damiafuentes/DJITelloPy 2019. VIII. 20. 10.25 h.
19. RYZE ROBOTICS (2018): Tello SDK 1.3.0.0, https://dl-cdn.ryzerobotics.com/downloads/tello/20180910/Tello%20SDK%20Documentation%20EN_1.3.pdf/ 2019. VIII. 29. 16.30 h.
20. DJI (2018): DJI-SDK/Tello-Python/doc/readme.pdf, https://github.com/dji-sdk/Tello-Python/blob/master/doc/readme.pdf 2019. VIII. 8. 13.04 h.
21. FATTAHI SANI, MOHAMMAD & KARIMIAN, GHADER (2017): *Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors.* 102-107. 10.1109/ICONDA.2017.8270408.
22. MARK EUSTON, PAUL COOTE, ROBERT MAHONY, JONGHYUK KIM & T. HAMEL (2008): *A Complementary Filter for Attitude Estimation of a Fixed-Wing UAV*. 340 - 345. 10.1109/IROS.2008.4650766.
23. MATIAS TAILANIAN, SANTIAGO PATERNAIN, RODRIGO ROSA & RAFAEL CANETTI (2014): *Design and implementation of sensor data fusion for an autonomous quadrotor*. Conference Record - IEEE Instrumentation and Measurement Technology Conference. 1431-1436. 10.1109/I2MTC.2014.6860982.
24. ZHENGYOU ZHANG (2000): *A Flexible New Technique for Camera Calibration*. Pattern Analysis and Machine Intelligence, IEEE Transactions on. 22. 1330 - 1334. 10.1109/34.888718.
25. MOHAMMAD FATTAHI SANI & GHADER KARIMIAN (2017): *Automatic navigation and landing of an indoor AR. drone quadrotor using ArUco marker and inertial sensors*. 102-107. 10.1109/ICONDA.2017.8270408.
26. SCIPY.ORG (2014): *Scipy.org/Docs/SciPy v0.14.0 Reference Guide/Interpolation (scipy.interpolate)/scipy.interpolate.splprep*, https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.interpolate.splprep.html 2019. IX. 20. 15.03h

27. XIAO LU (2018): *A Review of Solutions for Perspective-n-Point Problem in Camera Pose Estimation*, Journal of Physics: Conference Series. 1087. 052009. 10.1088/1742-6596/1087/5/052009.
28. OPEN SOURCE COMPUTER VISION (2019): *OpenCV Docs, Camera Calibration and 3D Reconstruction*, https://docs.opencv.org/4.1.0/d9/d0c/group__calib3d.html#ga61585db663d9da06b68e70cfbf6a1eac 2019. XI. 08. 15.33h
29. DIN (1990): *DIN 9300-2 - Luft- und Raumfahrt; Begriffe, Größen und Formelzeichen der Flugmechanik; Bewegungen des Luftfahrzeugs und der Atmosphäre gegenüber der Erde*, Deutsches Institut für Normung, Berlin
30. BME MOGI: *Mozgáslabor Wiki*, https://sites.google.com/mogi.bme.hu/mozgaslabor-wiki/mocap-rendszer?authuser=0 2019. XI. 17. 10.31h
31. SMART ROBOTIC SYSTEMS (2015): *aruco_mapping_filter*, Image filter for better performance of ArUco detector, https://github.com/SmartRoboticSystems/aruco_mapping_filter 2019. XI. 17. 11.41h
32. GREG SLABAUGH (1999): *Computing Euler angles from a rotation matrix*

8. ILLUSZTRÁCIÓK JEGYZÉKE