

Pécsi Tudományegyetem
Pollack Mihály Műszaki és Informatikai Kar
Mérnök Informatikus Szak

TDK DOLGOZAT

KVADKOPTER REPÜLÉSI ADATOK VALÓS
IDEJŰ KINYERÉSE

Készítette: Schäffer Zsolt
Témavezető: Dr. Várady Géza, PhD

Pécs

2014

Tartalomjegyzék

Tartalomjegyzék	2
1 Bevezetés	3
2 A kvadrotorok aerodinamikai alapvetései	5
3 A DroneProxy implementációja	10
3.1 AR.Drone protokollok	10
3.2 Hálózati megfontolások	13
3.3 Algoritmusok és ötletek a DroneProxyban	14
3.4 C++ implementáció részletei	18
3.4.1 Eszközválasztás	18
3.4.2 Hálózati kommunikáció a wxWidgets-szel	19
3.4.3 Multi-threading a wxWidgets-szel	24
3.4.4 Grafikon rajzolás a wxMathPlot segítségével	26
3.4.5 Kódrészletek	28
4 A repülések elemzése, kiértékelése	39
5 Konklúzió	43
6 Hivatkozások	45

1 Bevezetés

A PTE PMMIK Műszaki Informatika Tanszékén több AR.Drone-nal kapcsolatos projekt van folyamatban, pl. objektumkövetés, beltéri navigáció, épület feltérképezés, szemmozgás követéses irányítás, növényfelismerés, mezőgazdasági célú térképezés, amelyekhez csatlakozva önálló kutatást végeztem. E munkák során mind beltéri, mind kültéri repülésre szükség van. Elkerülhetetlen, hogy az algoritmusok fejlesztése alatt a drón néha akadálynak ütközzön és/vagy lezuhanjon. Amint kiderült, a mechanikus alkatrészek állapotának kis- és közepes mértékű romlása is számottevő romlást okoz a drón stabilitásában, ami nem kívánt sodródáshoz, billegéshez, akaratlan kitérésekhez és végső soron további, gyakoribb karambolokhoz vezet.

Ahogy egyre sürgetőbbé vált a stabilitási problémák orvoslása, szenzorhibákat és firmware gondokat kerestem, mivel a mechanikai hibák nem voltak nyilvánvalóak, szembetűnőek. Valamint észrevehető volt némi stabilitási különbség a különböző időben beszerzett járművek kezdeti, új állapotban mutatott stabilitása között is, ezért a mechanikai okok csak később kerültek a figyelem középpontjába. Eleinte a szenzoradatok kinyerésére irányuló megoldást kutattam, mellyel a drónból kiinduló adatfolyamból valós időben, repülés közben kimenthetők az érdekes adatmezők.

Később rájöttem, hogy a nyers szenzorértékeken kívül sokkal több hasznos információ áll rendelkezésre, mint pl. szűrt, kondicionált mérési értékek. Sőt, a beágyazott szabályzó algoritmus bizonyos belső, számított adatai is elérhetők, mint az elvárt motorteljesítmények, becsült szélsősebesség és irány, a vizuális algoritmus által számított sodródási sebesség, stb... [1]

Idővel az is nyilvánvalóvá vált, hogy a mechanikai elhasználódás bizonyos tipikus fajtái jellegzetes karakterisztikát mutatnak a megfelelő repülési adatok grafikonján, lehetővé téve az ok azonosítását a grafikon kiértékelésével. Még ha a pontos ok nem is fedhető fel egy pillantásra, akkor is megmutatkozik, hogy melyik az alulteljesítő rotor, melyiket érdemes szétszerelni.

Akármi is az okozó, szenzorhiba vagy mechanikai leromlás, mindenképpen szükség van egy jól használható módszerre, amivel a repülések adatainak kinyerése, mentése, ábrázolása megvalósítható. Léteznek, akár ingyenesen is, olyan irányító alkalmazások, melyeket igazán könnyű megtanulni, használni. Néhány perc alatt

elsajátítható a vezérlők működése és néhány 5-10 perces repüléssel kellő rutin szerezhető az irányításban. A gond az, hogy a kényelmes, könnyen használható, „készen kapott” alkalmazások (melyek elérhetők tabletre, okostelefonra, pl.: AR.FreeFlight 2.4) nem támogatják a repülési adatok felhalmozását, mentését, pusztán néhány alapvető információ kijelzésére képesek, mint pl. az akku töltöttség megjelenítése.

Léteznek viszont jól használható diagnosztikai és vizualizációs alkalmazások, mint a *Drone Diagnostics v3* [2] –melynek funkcióira e dolgozat megírása során is többször támaszkodtam–, de be kellett látnom, hogy a diagnosztikai alkalmazások nem biztosítják azt a szintű irányítást, amit a repülésirányító applikációk képesek nyújtani. Csak egy egyszerű haladási parancs lehetséges egyszerre, összetett mozgások (pl. roll és throttle vagy pitch és yaw egyszerre) nem valósíthatók meg. A másik jelentős hátrány, hogy nincs video visszacsatolás, egyszerre kellene a PC képernyőt, az irányítófelületet nézni és a drón mozgását is követni.

Végül úgy döntöttem, elkészíték erre a célra egy saját alkalmazást, így jött létre a DroneProxy. Az alapötlet szerint egy olyan forgalomtovábbító algoritmust kellett készíteni, ami megfelel a következőknek:

- tisztában van a drón által használt protokollokkal,
- transzparens a drón és tetszőleges irányító szoftver számára is,
- amennyire csak lehetséges, rugalmas az alapul szolgáló hálózati infrastruktúra tekintetében,
- platform független

A DroneProxy konstrukciója során egy kiterjeszthető keretrendszerre fejlődött, több lehetőséget kínál, mint amit az elsődleges célja meghatároz, funkciói túlmutatnak egy adat-továbbító/kinyerő eljárásán.

A dolgozat a felépítése a következő:

- A 2. fejezet a kvadrotorok közös jellemzőivel, alapvetéseivel foglalkozik, a leggyakoribb mechanikai meghibásodások bemutatásával, illetve ezek stabilitással való összefüggéséről szól.
- A 3. fejezet a drón kommunikációját elemzi és a DroneProxy implementációját mutatja be.

- A 4. fejezet néhány rögzítésre került repülési folyamatot mutat be grafikonokkal. Rámutat, hogyan lehet a hibára vonatkozó következtetéseket levonni a vizualizált adatokból. Igazolásképpen bemutatom, hogyan változik a grafikon, amikor a javított drónnal újra elvégzek egy azonos repülést.
- Az 5. fejezetben a konklúzióon kívül a DroneProxy keretrendszer további hasznai kerülnek tárgyalásra a jövőre vonatkozó tervekkel együtt.

2 A kvadrotorok aerodinamikai alapvetései

A kvadrotoros légi járművek alulaktuáltak, 6 szabadsági fokkal rendelkeznek, ehhez 4 aktuátor tartozik. A négy beavatkozási pont a négy rotornak felel meg, melyek fordulatszáma egyenként, egymástól függetlenül szabályozható. Amikor nincs semmilyen zavaró tényező, mint pl. a szél, vagy a jármű önmaga által keltett légáramlatai, akkor a „**hovering**” (lebegés) a legegyszerűbben leírható jelenség. Ez azt jelenti, hogy a jármű megtartja térbeli pozícióját és irányultságát. Ha a geometria és a tömegeloszlás teljesen szimmetrikus, ehhez egyszerűen minden rotornak azonos sebességgel kell forognia. A rotorok közül kettő (egy átlós pár) az óramutató járásával megegyező irányban forog, míg a másik kettő ezzel ellentétes irányban. A konkrét szögsebességet, amivel mind a négy rotor pörög, adott jármű esetén csak a hordozott teher (pontosabban az összsúly) szabja meg. A beépített gyorsulásmérő mondja meg, mikor tart egyensúlyt az emelőerő a gép súlyával. Amint azt említettem ez csak elméletben egyszerű, a valóságban a zavaró hatások miatt a beépített algoritmus számára ez az egyik legnehezebb feladat. Nem véletlen, hogy a leromlott állapotú drónok hibái először leginkább lebegés közben válnak szembeűnővé. Összetett látó algoritmusok is működnek az AR.Drone-ban, melyek fontos szerepet játszanak a stabil lebegés megvalósításában, gondoljunk csak arra, hogy a konstans sebességű vízszintes síkban történő sodródás a gyorsulásérzékelők számára megkülönböztethetetlen a rögzített pontban való lebegéstől.

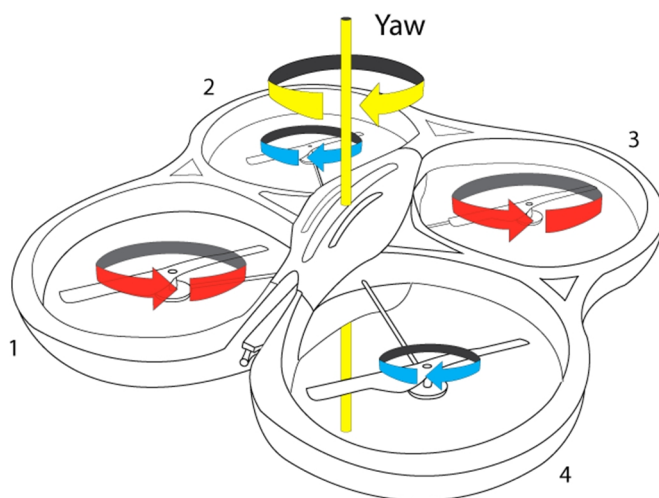
A négy alapvető haladási, elmozdulási forma a következő:

A „**throttle**” a „**hover**”-tól csak abban különbözik, hogy az emelőerő nem tart egyensúlyt a gravitációval, tehát függőleges irányú gyorsulás következik be. A 4 rotor ezúttal is azonos sebességgel forog, nincs dőlés vagy elfordulás.

A „pitch” akkor valósul meg, amikor az első és hátsó rotorpár sebességének aránya 1-től eltér. Pl. a hátsó pár gyorsabban forog, akkor a hátsó kissé megemelkedik, az első rész kissé süllyed, a jármű megdől. Ilyenkor a propellerek emelőerejének van vízszintes irányú összetevője, ami a drónt előre felé gyorsítja (az x tengely mentén). A függőleges összetevők továbbra is egyensúlyt tartanak a gravitációval, tehát a propellerek fordulatszáma adott dőlésszöghöz megfelelően úgy növekszik, hogy függőleges gyorsulás ne történjen. Azt a szöget, melyet a drón ilyenkor a vízszintes síkkal bezár θ -val jelöljük. A „pitch” tehát az y tengely körüli elfordulást jelenti.

A „roll” ennek az oldalirányú megfelelője, vagyis az y tengely menti gyorsulást az x tengely körüli elforgatással lehet elérni. A drón vízszintes síkkal bezárt szögét ϕ -vel jelöljük.

A „yaw” a függőleges tengely (z tengely) körüli forgást jelenti és ψ -vel jelöljük. Ahogy korábban említettem, a négyből 2-2 rotor ellentétes irányban forog (a lapátok is ennek megfelelően vannak kialakítva). Szokásos esetben az óramutató szerint és azzal



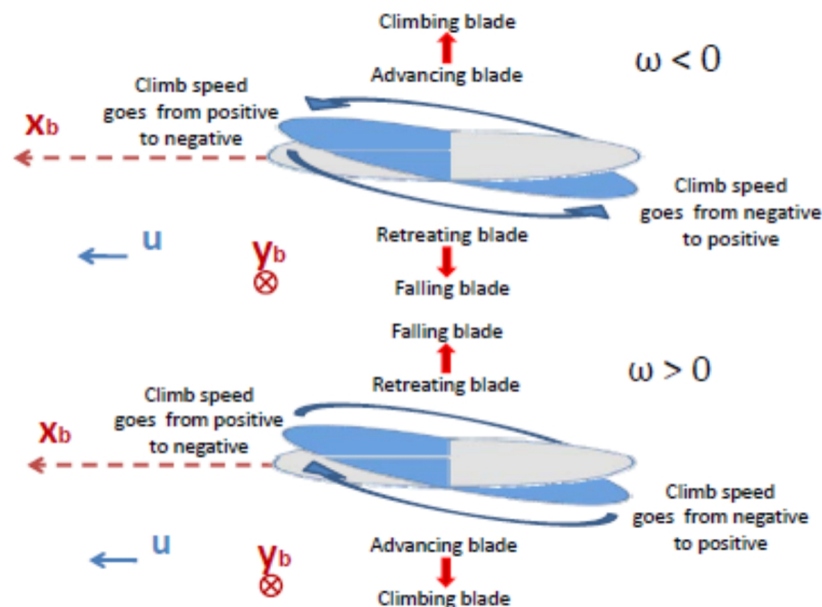
1. ábra - "Yaw" [3]

ellentétesen forgó propellerek járműre gyakorolt forgatónyomatéka kioltja egymást. Az eddig tárgyalt (bal-jobb, illetve első-hátsó) rotor-párok fordulatszám változtatása egyforma mértékben növelte vagy csökkentette a balra vagy jobbra forgató nyomatékot, így az említett

páronkénti kioltó hatás eddig minden esetben megvalósult. A függőleges tengely körüli elfordulást ezen arány megbontásával lehet elérni. Pl. az óramutató szerint forgó rotorok gyorsításával és az ellentétesen forgók azonos mértékű lassításával elérhető, hogy sem a vízszintes síktól való eltérés sem függőleges gyorsulás nem következik be, azonban a jármű az óramutatóval ellentétes irányba kezd forogni a z tengely körüli nem-zérus össznyomaték miatt.

Bármely komplex mozgás megvalósítható az itt említett alapvető manőverek szuperpozíciójaként, bármely pozícióba eljuttatható a drón és bármely orientációt fel tud venni, így biztosítva a 6 szabadsági fokot. Bár tetszőleges orientációhoz nem tartozhat tetszőleges gyorsulás. [4]

E rövid bevezető célja, hogy rávilágítsak, hogy az alapvető manőverek rotor párok sebesség arányaival operálnak. Ezért egy rotornak a tervezettől való (a többi motorhoz viszonyítva jelentős) eltérése olyan aszimmetriát visz be minden alapvető manőver megvalósításába, ami kulcsfontosságú a jármű stabilitásvesztésében. A működési elvből adódik, hogy a szabályozó algoritmus nagymértékű aszimmetriát nem tud kompenzálni.



2. ábra - verdesési szög, dőlési jelenség előre repüléskor [4]

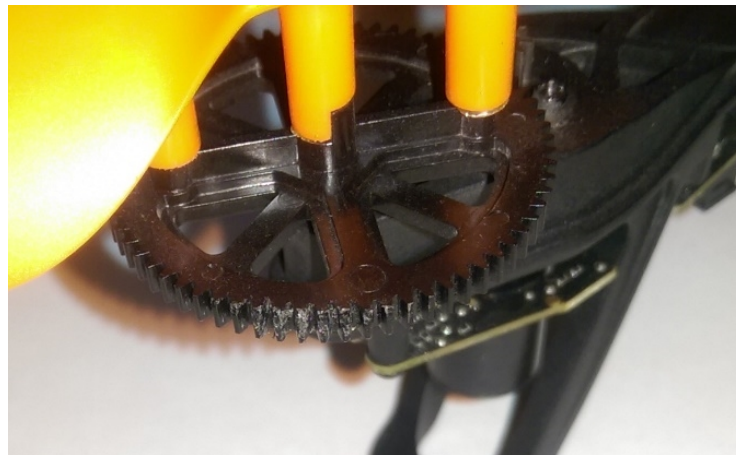
Még egy fogalmat meg kell említeni a mechanikai okok megértéséhez, ez a „flapping angle” (verdesési szög). A propellerek rugalmas műanyagból készülnek, ez a természetük lehetővé teszi, hogy a rotor tengelyre merőleges síktól eltérő síkban pörögjenek. A propeller pengék és az említett sík által bezárt szög a verdesési szög. Ezt a propeller és a levegő közti aerodinamikai kölcsönhatásból származó erő, a centrifugális erő, és a propeller merevségéből adódó erő együttesen határozzák meg. Az AR.Drone beágyazott szabályozó algoritmus csak a stabilizált verdesési szöget veszi figyelembe. A propellerek ütközésből adódó esetleges merevség és alakváltozását az algoritmus nem tudja számításba venni. [4]

Néhány tipikus példa a mechanikus alkatrészek hibáira:



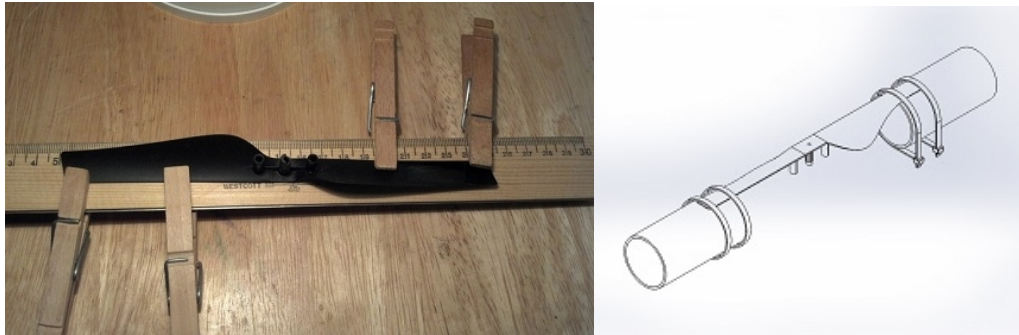
3. ábra - hajlott tengely fotója

- Az **elhajlott tengely** a veresési szög sztochasztikus viselkedését okozza, valamint csökkenti a motor hatásfokát.



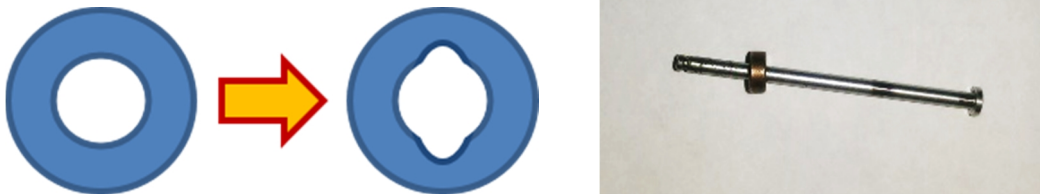
4. ábra - sérült fogaskerék fotója

- **Sérült fogaskerekek**, torzult vagy hiányzó fogak akkor alakulnak ki, amikor a propeller gyors forgása közben a jármű nekiütközik valamilyen akadálynak, mely hirtelen blokkolja azt. A műanyag fogaskerekek fogai gyakran elhajlanak, elnyíródnak. A kerület e részén való átfordulásakor növekszik az ellenállás, esetenként fogak kerülnek átugrásra. Nagy sebességnél a rotor tehetetlensége miatt könnyen átlendül ezeken a szakaszokon, de összességében a motor nagyobb villamos teljesítményére van szükség azonos fordulatszám eléréséhez, továbbá a rotor zaja is jelentősen megnő.



5. ábra - properrelek helyreállítása [5]

- A **hajlott, törött propellerek** miatt kiszámíthatatlan a verdesési szög. Ezek a sérült alkatrészek más irányú és nagyságú erőt fejtenek ki, mint amit az algoritmus elvárna. A törött propellereket érdemes lecserélni, a hajlottak javíthatóak egy megfelelő átmérőjű fémcsőre való ráhajlítással, forrólevegős lágyítással.



6. ábra - bronzgyűrűk dilatációja, mely látható tengelyszög eltérést okoz

- A **kopott, elhasznált gyűrűk** szintén kedvezőtlenül hatnak a verdesési szögre, mivel megengedik, hogy a tengely szabadon mozogjon, megdőljön, nem tartva ez által a propellereket a kívánt síkban.

Habár kulcsfontosságú a stabilitáshoz e hibák ismerete, csak akkor derülnek ki (főleg kezdeti szakaszban), ha a drón szétszerelésre kerül. Ez volt a fő motivációm egy olyan számítógépes eljárás készítésére, amely szoftveres eszközökkel segít a mechanikai hibák felfedésében.

3 A DroneProxy implementációja

Ebben a fejezetben az AR.Drone 2.0 verzió protokollja kerül bemutatásra. A legtöbb elv az 1.0-s változatra is igaz, de nem minden részlet egyezik. [6] A DroneProxy a 2.0-s változat minden firmware verziójával képes együttműködni.

3.1 AR.Drone protokollok

A drón hat különböző módszert használ a különböző célú információk átadására. Az adattovábbítás IEEE 802.11n szabványú Wi-Fi hálózaton történik a TCP/IP protokollkészlet használatával. Hogy megértsük, mit tesz egy „AR.Drone-t értő” forgalomtovábbító, az alábbiakban tisztázom az alkalmazott elveket.

A **NavData** folyam áll a figyelem központjában, jelen projekt célja ezt az adatfolyamot megcsapolni a továbbítás során. Ez egy előre definiált adatmezőkből álló rekordokat tartalmazó folyamatos adatáram. Példa az adatmezőkre: magnetométer irány (magneto_heading_fusion), látásból számított x irányú sebesség (vx_vision), motorok elvárt teljesítménytényezője (motor[n]_pwm), euler szögek (theta, fi, pszí), nyers gyorsulás értékek (gyro_raw[axis]), stb...

Egyes mezők nyers szenzoradatokat tartalmaznak, néhány szűrt, számított érték is helyet kapott, valamint megtalálhatók a beágyazott szabályozó algoritmus egyes belső változói is. Az adatmezők 27 csoportba szervezettek. A csoportok („tag”, címke) külön-külön ki és bekapcsolhatók a parancs interfészen keresztül, a tekintetben, hogy belekerüljenek-e az adatfolyamba vagy sem. Az aktivált tag-ek halmazába tartozó mezők friss értékét egy-egy üzenetben konkatenálva küldi ki a drón kb. 30 ms időközönként. A csoportok fejléccel rendelkeznek, az üzenet egésze üzenetfejléccel, hosszal és CRC ellenőrző összeggel van keretezve. A kommunikáció e formája az UDP (User Datagram Protocol)-ra alapozott, mely a TCP/IP protokollkészlet része. Az UDP kapcsolat nélküli protokoll, úgy nevezett „legjobb igyekezet” típusú protokoll. Ez többek között azt jelenti, hogy az üzenetkézbesítés nem garantált, nincs semmilyen visszaigazolás, nem sorszámozottak az üzenetek, így a kézbesítés sorrendje sem garantált, viszont alacsony késleltetéssel jár a használata. A sorrendiség egyértelműségéről egy belső időpecsét gondoskodik, a befoglalt adatmezők egyike a Linux kernel indulása óta eltelt időt tartalmazza, ez az adott

adathalmaz mintavételezésének időbélyegzője. A protokoll kapcsolat nélküli mivolta azt jelenti, hogy a felek elérhetősége, aktív mivolta nem követett, nincs kapcsolat ébrentartás (keep-alive) sem. Az adatfolyamot a repülésvezérlő állomás kezdeményezi egyetlen, tetszőleges tartalmú csomag küldésével a drone 5554-es UDP portjára. Válaszul ez első üzenet feladójának címére a drón elkezd az adatok folyamatos, 30 ms-ként ismétlődő kiküldését. A kommunikáció kvázi-unidirekcionális. Kezdetben ez a folyam csökkentett adattartalmú 24 byte-os üzenetekből áll, ez a rendszertöltő (bootstrap) mód. A drónt konfigurálni kell a megfelelő tag-ek kiválasztásával, hogy kilépjen ebből az üzemmódból. Az összes elérhető tag kiválasztása esetén egy üzenet mérete 2190 byte (a 2.4.8-as drón firmware esetén).

A **video** folyam H.264 eljárással tömörített mozgóképet továbbít, általában 1500 kbit/s körüli bitrátával. Ez kapcsolat alapú TCP protokollt (Transmission control protocol) használ. Ez a protokoll operációs rendszer szintjében megvalósított támogatást nyújt a csomagok sorrendjének és kézbesítettségének garantálására, a hiba felismerésre és szükséges újraküldések megszervezésére. Úgy gondolhatunk rá, mint egy csővezetékre, amiből adatokat lehet kivenni, amikor a másik a végén adatokat töltenek bele. A kiolvasás darab mérete nem kell, hogy egyezzen betöltés feldarabolásával, a puffer ésszerű határain belül tetszőleges. Az alapul szolgáló hálózat fizikai követelményeiből adódó fragmentáció az alkalmazások előtt rejtve marad. A video folyam olvasásába tetszőleges ponton be lehet kapcsolódni, a kulcskockák megjelölésre kerülnek, a video értelmezéséhez szükséges tokba zárás (encapsulation) az alkalmazási rétegben kerül értelmezésre a hivatalos AR.Drone függvénykönyvtárban megfogalmazott irányelvek szerint. A drón TCP figyelőként működik (kiszolgáló szerep) az 5555 TCP porton. Amikor a vezérlőalkalmazás kapcsolatlétesítés céljából felkeresi (3 lépéses TCP kézfogás), azonnal elindítja a video folyamat a kezdeményező felé. Ez a kommunikáció is unidirekcionális, leszámítva a kapcsolat kezdeményezést és az operációs rendszer által megvalósított csomag visszaigazolásokat.

Az **AT parancs interfész** az elmozdulási utasítások és a konfigurációs beállítások fogadására szolgál. Az AT parancsokat a drón 5556-os UDP portjára kell küldeni. Ez az adattovábbítás szigorúan egyirányú (vezérlőről drónra), a drón ebben a folyamatban sohasem válaszol vagy igazol vissza. Mivel az UDP nem garantálja az üzenetek

kézbessítési sorrendjét, egy üzenet belsejébe foglalt számozás akadályozza meg, hogy sorrendből kiesett parancsok ne kerülhessenek végrehajtásra.

A datagramok ember által is olvasható ASCII karakterláncot tartalmaznak, amely egy sorszámból, egy parancsból és a parancsnak megfelelő számú, vesszővel elválasztott paraméterből áll. A parancsokat redundánsan kell továbbítani, az utolsó érvényben lévő parancs folyamatos ismételtetésével. Ez kettős célt szolgál. Az egyik, hogy a ritkán bekövetkező üzenetvesztésnek ne legyen látható következménye, a másik, hogy a csomagok sorozatos kiesése észlelhető legyen. Amikor másodperc nagyságrendű ideig egyetlen parancs sem érkezik, a drón a Wi-Fi kapcsolat kiesését feltételezi és biztonsági intézkedés gyanánt lebegés módra kapcsol, vagyis megpróbál egyhelyben maradni, nehogy bárminek is nekiütközzön.

A **konfigurációs interfész** egyetlen célja a drón beállításainak lekérése, beállítás elvégzése itt nem lehetséges. A drone TCP figyelőként működik az 5559 porton. Amikor egy kliens csatlakozik és a parancs interfészen a megfelelő parancs kiadásra kerül, akkor a drón egy ASCII karakterláncot küld, mely tartalmazza az összes beállítási lehetőséget *kulcsnév = érték* formában, egymás után felsorolva.

A **hitelesítő interfész** a drón felfedezhetőségét szolgálja. Egy szórásos üzenetre („PARROT AUTH”) a drón a saját címéről küldve unicast üzenettel („PARROT AUTH OK”) válaszol, jelezve, hogy jelen van a hálózaton és az adott címmel rendelkezik. Ez mind TCP, mind UDP szórásos üzenetek esetén működik az 5552 porton. Csak néhány repülésvezérlő alkalmazás használja ezt a módszert, mások egyszerűen feltételezik, hogy a drón a 192.168.1.x IP címen található, ahol $x=y-y \bmod 8$, y a kliens címének utolsó oktetje.

Két független **FTP** (File Transfer Protocol) kiszolgáló is fut a drón beágyazott Linux operációs rendszerén. Az FTP alapjául szolgáló protokoll a TCP. Az AR.Drone passzív FTP-t használ, amely külön dinamikusan életre hívott adatcsatornák létrehozásával jár, melyet ASCII szövegben egyeztetnek a felek, ezért követése és továbbítása körülményesebb.

Az egyik FTP kiszolgáló a 21-es porton fut, és a drónhoz csatlakoztatható USB flash meghajtó tartalmát szolgálja ki, mely rögzített videókat tartalmazhat. A másik kiszolgáló az 5551-es porton üzemel, mely a verzióinformációkkal és firmware frissítéssel kapcsolatos fájlrendszer mappa tartalmát teszi elérhetővé hálózati kapcsolaton

keresztül. Ez furcsának tűnhet, de a fájlátvitel ismeretét is meg kell valósítani a DroneProxyban, mert a repülésirányító alkalmazásnak szüksége van rá. (Az AR.FreeFlight nem engedélyezi a „Piloting” menüpontot, amíg nem ellenőrizte az FTP kapcsolatok meglétét, az aktuális firmware változat számát is egy itt található szövegfájlból olvassa ki.)

További portokra is hivatkozik az AR.Drone függvénykönyvtár [6] forráskódja, ezek a video rögzítő port (a rögzített video más minőségű tömörítést használ, mint a valós idejű megjelenítésre szánt video folyam, de a rögzítés ideje alatt ez is továbbításra kerül a hálózaton), a nyers felvételi port, és a printf port. Az itt említett portok gyengén vagy egyáltalán nem dokumentáltak, de nem is szükségesek a repüléshez, így a DroneProxy megvalósításában sem játszanak szerepet.

3.2 Hálózati megfontolások

Az adatátadás UDP datagramoknak és TCP szegmenseknek két külön alhálózat közti továbbításával valósítható meg. Alaphelyzetben a drón 192.168.1.1-es IP címre van konfigurálva, 24 bites hosszúságú alhálózati maszkkal. Ez egy egyszerű paranccsal megváltoztatható, illetve tetszőleges számú alias cím megadására is lehetőség van, ezek a parancsok a drón automatikus indítási szkriptjéhez is hozzáadhatók. A 192.168.1.0/24-os alhálózat fixen kódolva van az AR.Drone függvénykönyvtárba, ezért valószínűsíthető, hogy bármely harmadik fél által készített alkalmazás (amely külön nem ad lehetőséget a drón IP címének beállítására) 192.168.1.1-es címen fogja keresni a kvadkoptert. Ehhez a DroneProxy kialakításakor alkalmazkodni kellett.

Fentiek miatt döntöttem úgy, hogy ezt választom az egyik alhálózatnak a kettő közül, melyek között a DroneProxy működik. Ez az ál-drón alháló. A PC, mely a DroneProxy-t futtatja, felveszi a 192.168.1.1-es címet és a kommunikáció minden megnyilvánulásában úgy viselkedik, mintha a drón lenne. Az operációs rendszer segítségével hoztam létre Wi-Fi hozzáférési pontot, mivel a néhány évnél nem idősebb notebookok nagy része olyan Wi-Fi chipet tartalmaz, amely megfelelő eszközmeghajtó támogatással Wi-Fi AP-ként is képes üzemelni. A táblagép vagy okostelefon ehhez a

hozzáférési ponthoz kapcsolódik, ugyanúgy, mint a közvetlen drón-telefon kapcsolat esetén tenné, és a drónnak tűnő PC-vel kommunikál, számára a csere észrevehetetlen.

A másik alhálózat, melyben a Drone-Proxy operál, az az alhálózat, ahol a valós drón jelen van. Ez az alhálózat szabadon konfigurálható, mert sem a DroneProxy, sem a jármű beágyazott operációs rendszere nem szab korlátokat. Átláthatósági okokból .1-es végződésű címet és /24-es alhálózati maszkot alkalmazok itt is (amikor csak egy drón van jelen a hálózaton), de ez nem kötelező. Ennek az alhálózatnak a szabad konfigurációja azt is lehetővé teszi, hogy tetszőleges számú drónt üzemeltessünk egy időben ugyanazon a hálózaton. A drón beépített hozzáférési pontját használjuk arra, hogy a PC hozzacsatlakozzon. A drón operációs rendszere lehető teszi (bár ez hivatalosan az AR.Drone gyártója által nem támogatott), hogy egy meglévő Wi-Fi infrastruktúrához kapcsolódjon, pl. olyan épületekben, ahol már teljes lefedettségű Wi-Fi hálózat van kiépítve.

A bevezetőben említettem, hogy a DroneProxy funkcionalitása túlmutat az adatkinyerésen. Az egyik kedvező tulajdonsága, hogy nagymértékben kibővíti az alkalmazható hálózatok skáláját, megszünteti az eredeti tervezésből adódó hálózati kötöttségeket. Közvetítő hálózatként bármilyen, helyesen konfigurált IP alapú infrastruktúra használható. Lehet az vezetékes, vezeték nélküli, akár interneten vagy virtuális magánhálózaton keresztül is áramoltatható az adat, sőt a DroneProxy még átjáróként is képes működni olyan helyzetben, ahol az IP forgalomirányítás vagy címfordítás nem megfelelően konfigurált. Az egyetlen követelmény, hogy a DroneProxy olyan PC-n fusson, amelynek mindkét érintett alhálózatban van hálózati interfésze. A DroneProxy láncolt struktúrában is alkalmazható, több példányban futtatva. Ez használható például a vezeték nélküli hatósugár megnövelésére.

3.3 Algoritmusok és ötletek a DroneProxyban

A DroneProxy platform független, C++ nyelven íródott, csak standard könyvtárbeli függvényekre és a wxWidgets nyíltforrású, cross-platform függvénykönyvtárra támaszkodik, utóbbi grafikus felhasználói felületi elemeit és socket műveleteit használva, valamint a wxMathPlot-ra, mely a kinyert adatok vizualizációja során nyújt segítséget. A wxWidgets lefordítható Windows, Linux és MacOS operációs

rendszerekre. Az egyetlen különbség a DroneProxy különböző operációs rendszereken történő futtatása esetén abban rejlik, hogy hogyan valósul meg a vezeték nélküli hozzáférési pont kialakítása. A Windows 7 és újabb változatai támogatják a wlan hostingot, a netsh parancssori utasítással lehet AP-t létrehozni. Linux felhasználók alkalmazhatják a hostapd daemon-t (mely sok otthoni, kirodai Wi-Fi router beágyazott szoftverében is fut). Természetesen bármely hardveres Wi-Fi AP is használható, ha a DroneProxy-t futtató PC nem rendelkezik megfelelő Wi-Fi chippel.

A hivatalos AR.Drone Library forráskódját alaposan tanulmányoztam a szükséges ismeretek megszerzéséhez, az AR.Drone protokoll megértéséhez, de a libARDrone nem szükséges a program linkeléséhez, kompilálásához. A kommunikáció gyengén dokumentált aspektusainak tisztázásához a Wireshark és a Shark for root (Android) ingyenes alkalmazásokat használtam. Ezek betekintést biztosítottak a hálózati párbeszédbe és lehetővé tették bizonyos ismeretlen részletek visszafejtését is.

A 1. pontban bemutatam a 6 legfőbb kommunikációs módot, amit a drón alkalmaz, most szeretnék leírni néhány érdekesebb megvalósítási részletet az említett protokollokhoz.

Az UDP továbbítás a legáttekinthetőbb. Csak a vezérlőalkalmazástól beérkező első datagram érdekes. A küldő címét és port számát az első küldés fejlécéből nyerem ki. A célcímét és portot az AR.Drone dokumentáció alapján már előre ismerjük. A dróntól érkező válaszokat már csak el kell juttatni az így feltérképezett úton visszafelé, oda ahonnan a kezdeti csomag érkezett. A navData folyam nem csak továbbításra, hanem fájlba mentésre is kerül. Az alkalmazott formátum az msh-tools Drone Diagnostics [2] alkalmazásával egyezik, aminek erőssége a repülési adatok vizualizációja. A Drone Diagnostics használatának szükségszerűségét szeretném elkerülni saját megjelenítő kód megírásával, mely független Sharanov alkalmazásától és lehetőséget biztosít valós idejű, élő grafikonrajzolásra is, erről részletesen a 4. pontban írok.

A parancsmanipuláció a szintén UDP-t használó AT parancs interfészen megy végbe. A parancsok elfogására és módosítására is lehetőség van a DroneProxy keretrendszerben, mielőtt az továbbítaná őket. Ezzel szűrhetők a parancsok, vagy módosíthatók a paraméterek. Így történik ez a következő paranccsal is:

AT*CONFIG=(*sorszám*,"general:navdata_options",*paraméter*). Ez kiválasztja azokat a navData csoportokat, melyeket az adott repülésirányító alkalmazás hasznosnak

tart, melyek belekerülnek a navData folyamba. Minden alkalmazás csak egy részét kéri a lehetséges adatoknak, ez mindig az adott alkalmazásra jellemző. (pl. egy egyszerűen kezelhető „távirányító” jellegű alkalmazás grafikusan kijelzi a megmaradt akkutöltést és a drón padlótól, talajtól mért magasságát, de kevésbé érdekelt gyorsulásmérők pillanatnyi nyers kiolvasási értékeiben.) Egy diagnosztikai program viszont szeretne ennél többet - ha nem az összeset- megkapni az elérhető adatokból. Tehát amikor egy alkalmazás beállítja, mit kíván megkapni, akkor ennek az utasításnak a paraméterét le kell cserélni egy olyanra, mely megfelel a diagnosztikai céljainknak. A paraméter egy decimális szám, számjegyekből álló karakterlánccal ábrázolva, amit egy 32-bites nem negatív egészzé konvertálva olyan bitminta adódik, melyben az alsó 27 bit mindegyike egy-egy ki-be kapcsolható navData adatcsoportnak felel meg ez elérhető 27-ből. Az összes navData tag kiválasztása a következő *paraméter* behelyettesítésével lehetséges: $2^{28}-1=268435455$

A TCP továbbítás egy kicsit összetettebb, mert a kapcsolatoknak állapota van, ezeket követni kell. A kapcsolatokat csak az egyik irányból lehetséges kezdeményezni. (Egy kiszolgáló socket, mely figyelőként működik, nem kezdhet 3 fázisú TCP kézfogást, azt az AR.Drone példájában csak a vezérlőalkalmazás kezdeményezheti.) Ezzel az a probléma merül fel, hogy amikor az irányító vár valamire és nem kapja meg a kért adatot határidőn belül, lezárja a kapcsolatot, mire az adat készen állna, így a válasz a proxy-ban ragad, az már nem tudja tovább adni és nem kezdeményezhet új kapcsolatot az irányító alkalmazás felé. Az AR.FreeFlight elég türelmetlen ilyen tekintetben, amikor a konfigurációs interfészről van szó. Amikor a FreeFlight új kapcsolatot kezd, már egy másik forrásportot fog használni, ezt is követni kell. A csomagok nem mindig ugyanazon az úton közlekednek visszafelé, mint előre felé. További nehézség, hogy a wxWidgets socket műveletekkel foglalkozó kódrésze nem mindig érzékeli helyesen vagy időben a kapcsolat lebontását, illetve a wxWidgets nem tud ébrentartó üzeneteket (olyan TCP szegmens, aminek hasznos adattartalma 0 méretű, ezt az időzítő nullázására használják, amikor nincs más hasznos közölnivaló egy kapcsolaton) küldeni, amelyek a kapcsolat nyitva tartását meghosszabbítanák. A problémát kezdetben úgy oldottam meg, hogy aktív lekérdező hurkot alkalmazok a blokkoló socket műveletek helyett. Ez nem takarékos a CPU idővel, de csak másodpercekig van rá szükség az inicializáció során. Amint a konfigurációs karakterláncok átvitele megtörtént, az interfészt használaton kívül kerül a következő repülési ciklusig. Valamint a video TCP folyama is megindul eddigre, ami

csak egy irányú adattovábbítás az inicializációt követően. Ilyenkor már nyugodtan vissza lehet térni a blokkoló íráshoz, olvasáshoz, ami kedvez a CPU kihasználtságnak. Ezt később egységesítettem, hogy a TCP forgalmat ugyanaz az osztály, ugyanaz az algoritmus valósítsa meg, adattartalomtól, portszámtól függetlenül. A végső megoldás már egyáltalán nem igényel aktív lekérdező hurkokat (active polling loop) és üzemmód váltásra sincs szükség, ezért robusztusabb. Ezt több, csak egyirányú továbbítást végző programszál alkalmazásával értem el, kapcsolatkövetéssel, szálak közti kommunikáció segítségével algoritmizáltam, részletek a negyedik pontban.

Az FTP implementációja szimulációval történt, nem adattovábbítással. Azért döntöttem így, mert a továbbítás megvalósításához a két kiszolgáló esetében összesen 8 TCP kapcsolat nyomon követése volna szükséges. További nehézség, hogy a PASV válaszok, melyek a dinamikus adatcsatorna létrehozásában játszanak szerepet, a drone valós IP címét tartalmazzák egy dinamikus port számmal, ráadásul decimálisan ábrázolva. Ezt mindkét irányban címfordítással szerkeszteni kellene a DroneProxy hálózati konfigurációjának megfelelően. Ez szegmens méretek változásával is járhat, mert a decimális ábrázolásban nincsenek vezető nullák. (Még a hardveres NAT címfordító útválasztók sem mind képesek passzív FTP átadásra). Ezek az érvek mind a továbbítás ellen szólnak. A szimuláció mellett szól, hogy az FTP párbeszéd mindig azonos –ezt a Wireshark segítségével derítettem ki–, az egyetlen eltérés a firmware verziószáma, mely a párbeszéd végén hangzik el. Az AR.Drone FreeFlight arra használja az FTP-t, hogy meghatározza a drón fő verziószámát (1 vagy 2), mielőtt engedélyezné az irányítást. Ezen kívül ellenőrzi, hogy vannak-e kimásolható video felvételek a csatlakoztatott flash memórián. Mivel van kényelmesebb módja a videofájlok másolásának és a firmware frissítés sem olyan feladat, amit a DroneProxyn keresztül volna érdemes elvégezni, úgy döntöttem, hogy valódi üzenettovábbítás helyett a DroneProxy csak utánozza az FTP párbeszédet, egy üres video mappát mutatva, verziószámként pedig egy tetszőleges, előre beállított értéket fog közölni. A DP az FTP folyamat során minden kérdésre előre rögzített választ ad, nyugtázza a bejelentkezést, könyvtárváltást, fájl megnyitást, létrehoz passzív csatornát és közli az előre beállított karakterláncot a version.txt tartalma gyanánt. Ez sokkal egyszerűbb, mégis a valóban használt funkciók szempontjából egyenértékű a drónnal történő párbeszéd közvetítésével, elegendő, ahhoz, hogy a FreeFlight a felszállást engedélyezze.

Ehhez hasonlóan a hitelesítő interfész is szimulált, hiszen a táblagépnek, okostelefonnak nem a drón valódi címére van szüksége, hanem arra a címre, amin keresztül kommunikálni tud vele. Ez jelen esetben a DroneProxy ál-drón hálózaton elfoglalt címe. Anélkül, hogy a „PARROT AUTH” üzenetet továbbküldené a drónnak a DroneProxy „PARROT AUTH OK” üzenettel válaszol a vezérlőalkalmazás kérésére, úgy feltüntetve magát, mint a valós kvadkopter. A hitelesítő interfész TCP és UDP socketekre is megvalósításra került, mivel egyes alkalmazások az előbbit, mások az utóbbit várják el.

3.4 C++ implementáció részletei

3.4.1 Eszközwálasztás

A DroneProxy megírásakor hangsúlyos szempont volt, hogy ne kötődjön feltétlenül, kizárólagosan a Microsoft Windows operációs rendszerhez. Annak érdekében, hogy ne kelljen különböző kódot írni az eltérő platformokra, választottam egyet az elérhető multiplatform függvénykönyvtárak közül. Azért a wxWidgets-et választottam, mert elég kiforrott, jól dokumentált, bizonyított eszközkészletről van szó, melyet már sikeresen alkalmaztam korábban mind Windows, mind Linux platformon, bár ez az első alkalom, hogy ugyanazt a programot szeretném mindkettőre lefordítani. Az osztályhierarchia alacsonyabb szintjén a wxWidgets hívja meg az adott platformra specifikus rendszerhívásokat (pl. makrók használatával: `#ifdef MSW`, stb...), míg a hierarchia magasabb szintjein az osztályok már csak más wxWidgets osztályokból származtatottak, így a Droneproxy kódját is módosítás nélkül le lehet fordítani mindkét platformon, ha csak wxWidgets osztályok kerülnek alkalmazásra. Ennek a kényelemnek azonban ára van, az alacsony szintű funkciók közül néhány rejtve marad, kihasználatlanul, de ezekkel a korlátokkal meg lehet bírkozni. (csak néhány, amivel találkoztam: a wxWidgets Socket osztályaival nem lehet 0 méretű adattartalommal rendelkező hálózati üzenetet küldeni, a blokkoló I/O műveletek alacsony szinten a `select()` függvényre épülnek, ennek időtúllépését nem lehet változtatni., ...)

A wxWidgets-szel készült a Code::Blocks fejlesztőkörnyezet, mely tartalmaz egy grafikus felhasználói felület tervezőt is, a wxSmith-t. Ez a Visual Studiohoz hasonlóan

lehetővé teszi a GUI komponensek egérrel történő hozzáadását, szerkesztését statikus objektumok esetén. Természetesen a megfelelő kód begépelésével a GUI osztályok is létrehozhatók dinamikusan. Valójában a wxSmith C++ kódot generál, melyet akár módosítani is lehet fordítás előtt, ez a kód lefordítva teljesen hasonló felhasználói felület eredményez, akár Windows, X11, GTK+ vagy MacOS X ablakkezelőre kerül lefordításra. Természetesen a többi fejlesztőkörnyezet szokásos szolgáltatásait is biztosítja (szövegszerkesztő, projekt kezelő, debugger, fordító, linker integrált kezelése, stb...). A Code::Blocks maga is nyílt forráskódú, ezért lefordítható akár az intel x86-os családtól eltérő processzorokra is. A korábbiakban már sikeresen használtam egykártyás ARM számítógépeken (BeagleBoard, Raspberry Pi, stb...) futtatva, így akár natív környezetben is lehet vele közvetlenül fejleszteni. A C::B számos fordítóprogrammal együtt tud működni (akár cross-compilerrel is), ezek közül a GCC-t választottam. Ez Linux disztribúciók esetén szinte mindig a repository része, windows platformon a MinGW csomag része, tehát mindkét esetben néhány kattintással telepíthető. A dolgozat megírásakor a DroneProxy elkészítéséhez a Code::Blocks 13.12-es változatát, a wxWidgets 2.8.12-es változatát, a wxMathPlot 0.1.2-es változatát és a GCC 4.8.1-es (mingw-gcc) verzióját használtam, ezek mind nyílt forrásúak. Végso soron ezt az eszközkészletet azért választottam, mert néhány hátrány ellenére, gyorsan és megbízhatóan lehet vele platformfüggetlen kódot készíteni, az előnyök nagyban meghaladják a hátrányokat.

3.4.2 Hálózati kommunikáció a wxWidgets-szel

A socket (szoftvercsatorna, csatlakozó) a TCP/IP protokollkészletben lehetővé teszi egy hálózati párbeszéd processzushoz, programszálhoz való hozzárendelését. A sockettel kapcsolatos műveleteket a szoftverek az operációs rendszer API-án keresztül érhetik el. Egy állomást (számítógépet) a TCP/IP hálózaton a 32 bites IP cím azonosít (IPv4 esetén), ez az ISO/OSI modell 3. rétegének felel meg. Minden számítógépen több folyamat, applikáció futhat egy időben, melyek egyszerre több párbeszédet folytathatnak a hálózaton keresztül. Azt hogy egy állomáson belül futó processzusok közül éppen melyik számára kell egy üzenetet kézbesíteni a port (kapu) cím mondja meg, ez a cím 16 bites (IPv4 esetén), ez felel meg a ISO/OSI modell 4. rétegbeli címnek. Egy állomásnak

a TCP/IP 4-es protokollkészletben 65535 UDP és ugyanennyi TCP portja van. Egy socket cím egy IP címből és egy port cím együtteséből áll. Az operációs rendszer nyilvántartást vezet arról, hogy mik a szabad és lefoglalt socketek, a lefoglalt socketeket melyik folyamat foglalta le, milyen a socket állapota (established, listening, connected, stb...). A wxWidgets 5 említésre méltó osztályt nyújt ezek kezelésére.

A socket fogalmát gyakran használják úgy, mint socket címet, azonban az operációs rendszerek és a wxWidgets szóhasználatában ez a kifejezés nem egy címre, inkább egy csővezetékre utal, melynek két vége van. Mindkét vége rendelkezik egy socketcímmel, az egyik cím az állomás számára mindig helyi (a socket cím IP cím része egy olyan cím, ami egyezik az adott állomás egyik konfigurált interfészének IP címével), a másik pedig rendszerint (de nem feltétlenül) távoli cím, mely egy másik állomás egy másik kapujához tartozó címet jelent. A csővezeték így módon egy címkézett, azonosított csatorna, melynek egyik végén betöltött adat a másik végéből kiolvasható. Ez mindkét irányra igaz, egy socket általában egyetlen párbeszédnek biztosít logikai továbbító közeget. A következőkben az említett 5 osztályt mutatnám be:

1. A wxIPv4Address egy socketcímnek felel meg. Két legfontosabb adattagja a Hostname és a Service, melyek az IP- illetve portcímet hordozzák.
2. A wxDatagramSocket UDP üzenetek fogadására és továbbítására szolgál, ez az UDP socketnek megfelelő objektum a wxWidgets-ben. Konstruktorában meg kell adni a socket helyi végpontját (wxIPv4Address). Nincs Connect metódusa. A Read és SendTo metódus értelemszerű használatával lehet párbeszédet folytatni. Az UDP párbeszédnek a protokoll konstrukciójából eredően nem rendelkeznek kapcsolatállapottal (lásd 3.1-es pont), ugyanazon socketcím segítségével egy állomás több párbeszédet is folytathat, az UDP csak lazán köthető a csővezeték hasonlathoz. Egy üzenet küldésekor az üzenet tartalmán kívül a SendTo meghívásának minden alkalmával a cél socketcímét is át kell adni paraméterben. A Read metódus bárholnan érkező üzenetet elfogad, az utoljára érkezett üzenet feladóját a GetPeer metódussal lehet lekérdezni, melynek visszatérési értéke az ellenoldal socketcímét tartalmazó wxIPv4Address típusú objektum. Az üzenetek darabolása az UDP socketek esetében az alkalmazás előtt nem teljesen rejtett. A Read metódus meghívásakor az egyik paraméter a kiolvasni kívánt bájtok száma. Ha nagyobb szám kerül megadásra, mint a socket várakozási sorában lévő

következő üzenet hossza, akkor a kiolvasás a tényleges üzenethossznál abbamarad. Ha kisebb szám kerül megadásra, akkor a kiolvasás ennél a méretnél marad abba, de az üzenet hátralevő része végleg elvész. A Read ismételt meghívásakor a várakozási sorban levő következő üzenet első bájtjával kezdődik az olvasás. Majd látható lesz, hogy a TCP socketek esetében ez nem így van.

3. A wxSocketClient a TCP ügyfélsocketnek megfelelő osztály a wxWidgets-ben. A TCP kapcsolatalapú protokoll (lásd. 3.1 pontban), ez pontosan illeszkedik a csővezeték hasonlathoz. A konstruktorban nem kell helyi címet megadni. A Connect metódus meghívásakor meg kell adni paraméterként a távoli címet. Ennek alapján az operációs rendszer kiválasztja azt az interfészt, amelyen keresztül a cél elérhető és azon keres egy szabad forrásportot, tehát a helyi socketcím generálása automatikus. A Connect kezdeményezi a 3 lépéses TCP kézfogást. Ilyenkor a csatlakozni kívánó állomás SYN flaggel megjelölt adattartalom nélküli üzenetet küld a kiszolgálónak, az SYN,ACK típusú válasszal nyugtázza a csatlakozási kísérletet, melyet ACK flag üzenettel nyugtáz a kezdeményező. A Connect sikeres lefutása után a Read és Write metódusokkal lehetséges párbeszédet folytatni. A hasznos adattartalmú üzenetek PSH flaggel kerülnek megjelölésre, ezeket a másik félnek ACK-kal kell nyugtázni. Ha ez nem történik meg, az operációs rendszer TCP/IP stackje a hiányzó rész újraküldését kezdeményezi. A PSH,ACK típusú üzenetek a nyugtázáson kívül hasznos adattartalmú viszontválaszt is hordoznak. Egy párbeszéd minden eleme sorfolytonosan számozott. Az üzenetek darabolása az alkalmazások előtt rejtve marad, a visszaigazolt szegmensek adattartalma számozás szerinti sorrendben egymás után belekerülnek egy sockethez tartozó pufferbe, ahonnan tetszőleges méretű darabokban kiolvashatók. A kapcsolat alapú protokoll és a kapcsolatállapot követés nem csak azt jelenti, hogy a végpontok közt dedikált csővezeték jön létre, hanem azt is, hogy a kapcsolat épsége is nyomon követett, pl. ha valamelyik fél FIN flaggel jelölt szegmenst küld, az operációs rendszer tudja, hogy ez a kapcsolat lezárult, vagy hiba esetén időtúllépést követően szintén lezárul egy kapcsolat, ha az üzenetváltás többszöri próbálkozás ellenére sem kerül visszaigazolásra. Ezt az IsConnected metódus meghívásával lehet ellenőrizni, melynek visszatérési értéke bool típusú.

4. A `wxSocketServer` a TCP socket kiszolgáló oldali megfelelője. A figyelő szerepű socket létrehozásakor a konstruktornak kell átadni a socket helyi címét. A figyelő vagy kiszolgáló socketek nem tudnak 3 lépéses kézfogást kezdeményezni, csak válaszolni tudnak erre, ezért nincs is `Connect` metódus ebben az osztályban. Helyette `Accept` metódust találunk, melynek visszatérési értéke egy `wxSocketBase` típusú objektum. Amikor valamelyik állomás a figyelőt felkeresi, akkor a csővezeték nem a figyelővel jön létre, helyette minden megkereséshez létrejön egy-egy `wxSocketBase` objektum. Így megoldható az, hogy egy kiszolgáló egyszerre több ügyféllel tartson fenn logikailag elkülönített párbeszédet (pl. egy vállalati adatbázis szerverrel egyszerre számos állomás tart fenn különböző célú, eltérő hozzáférési jogosultsági szinttel rendelkező adatcserét). Ennek értelmében a `Read` és `Write` metódusokat nem a figyelő osztályban, hanem a figyelő által visszaadott `wxSocketBase` osztályú objektumban kell meghívni. Bár a `wxSocketServer` is rendelkezik `Read`, `Write`, `IsConnected` metódussal, ezek meghívása nem tisztázott eredményhez vezet. Az osztály `Close` metódusának meghívása nem érvényteleníti a folyamatban lévő socketeket, csak lehetetlenné teszi újabb párbeszéd, csővezetékek létrejöttét.
5. Az utolsó említendő osztály a `wxSocketBase`, mely már szóba került az előző pontban. Ez az osztály nem rendelkezik sem `Connect`, sem `Accept` metódussal, ami a fentiek alapján logikus. `IsConnected`, `Read`, `Write` metódusa pedig értelemszerűen használható a fent leírtak szerint. Mind a `wxDatagramSocket`, mind a `wxSocketServer` és `wxSocketClient` ennek az osztálynak származtatott osztályai.

A fenti osztályok ismeretében már lehet üzeneteket cserélni a hálózat más állomásaival. Néhány gondolatot meg kell említenem még a blokkoló (vagy szinkron) és az aszinkron I/O műveletekről. A korábban említett socket osztályok konstruktorának meg kell adni egy flaget, ami erre vonatkozó információt hordoz. Az aszinkron írás, olvasás azt jelenti, hogy a `Read` és `Write` metódusok azonnal visszatérnek, még akkor is, ha egyébként I/O művelet nem történt (pl. a socket nincs csatlakoztatva, vagy olvasási puffere teljesen üres). A ténylegesen átvitt adatok mennyiségét az összes socket osztályban megtalálható `LastCount` metódus meghívásával lehet lekérni, mely nem-

negatív egész értékkel tér vissza. A blokkoló I/O műveleteknél a Read és Write metódusok nem térnek vissza, amíg a kívánt számú bájtot át nem vitték vagy hiba nem következett be. (Pl. ha nincs hiba, csak egyszerűen nem érkezett be üzenet, akkor a Read addig vár, amíg legalább a kívánt számú bájtot valamelyik állomás el nem küldi. Erre az időre a program nem halad tovább a következő utasításra.). Hibrid megoldásként a fenti osztályok kínálnak időtúllépéses metódusokat is, pl. WaitForRead, WaitForWrite, WaitForAccept, WaitOnConnect,...), melyek időparamétert is elfogadnak. Ezek bool visszatérési értékűek, igazat adnak, ha a kívánt művelet azonnal elvégezhető, de a műveletet nem végzik el, vagyis azonnal egy Read, Write vagy más értelemszerű művelet kell, hogy kövesse őket. Hamis értékkel térnek vissza, ha a visszatérés oka az időkorlát lejárt. Ha hiba történt (pl. a socket nincs is csatlakoztatva), akkor azonnal visszatérnek igaz értékkel, ugyancsak ez történik, ha a művelet azonnal, sikeresen elvégezhető, ezért használatuk kétértelmű eredményt ad, ami miatt nem kényelmes az alkalmazásuk.

Meg kell még említeni, hogy az aszinkron I/O műveletek esetén az algoritmus végrehajtása ugyan azonnal tovább halad, az eredményről nem értesül közvetlenül. Ez a wxWidgets keretein belül működő wxEvent-ek (esemény) segítségével történik. Az elsődleges programszámban fut egy eseménykezelő hurok (event loop), melynek a különböző objektumok eseményeket tudnak küldeni, ennek a hatására az meghívja hozzárendelt eseménykezelő metódust (event handler). Ugyanez a rendszer kezeli a GUI komponensek eseményeit, mint pl. a gombokra való kattintás, vagy menüvel való interakció. Itt megjegyezném, hogy az események külön-külön kerülnek kiváltásra és nem események együttese vagy egymásutánja váltja ki, így jelen feladathoz meglehetősen áttekinthetetlen lenne 12-14 socket 4 féle eseményének összes kombinációját helyesen kezelni.

Látszik, hogy egyik előkészített megoldás sem alkalmas önmagában a DroneProxy működtetéséhez. Az lehetőségeket összefoglalva:

1. A blokkoló I/O műveletek megállítják a program futását, amíg az egyik socket I/O műveletre várakozik, a többi figyelmen kívül lesz hagyva, ott pedig csomagok torlódhatnak fel, ami kaotikus késleltetésekhöz vezet.
2. Az aszinkron műveletek esetén az esemény alapú kezelés a feladat méretéből és kommunikációs csatornák összefüggéseit is figyelembe véve áttekinthetetlen volna, a programkód nehezen megfogalmazható lenne.

3. Az időtúllépéses megoldás hosszú időtúllépések esetén az 1. ponthoz állna közel, azzal együtt, hogy nem egyértelmű a „WaitFor...” metódusok visszatérési értéke, további tesztek szükségesek. (pl. LastError, LastCount, stb...)
4. Az időtúllépéses megoldás nagyon rövid időkorlátokkal ellátná a feladatot, mert a késleltetések kicsik maradnának, az egyes socketeket sorban végigkérdezve kiderülne van-e teendő. De ez az aktív lekérdező hurokra hasonlító módszer teljesen leterhelné a processzort, miközben az csak kis időarányban végez hasznos munkát, tehát ez a módszer nem hatékony.

3.4.3 Multi-threading a wxWidgets-szel

Fentiek miatt döntöttem úgy, hogy a feladatot programszálakba (thread) szervezem és a szálakban külön-külön blokkoló I/O műveleteket alkalmazok. Ez azért előnyös, mert a blokkoló művelet esetén a feladatütemező addig nem is oszt időszeletet az adott szálnak, amíg az általa várt I/O nem válik elérhetővé. Ezt nem az alkalmazás szintjén kell kezelni, mert ezt az operációs rendszer jól bevált feladatütemezője kezeli és tartja nyilván. A blokkoló I/O-t a wxWidgets a `select(...)` standard könyvtárbeli függvényre vezeti vissza. Ennek segítségével közölni lehet a feladatütemezővel, hogy mi az az operációs rendszer szintű esemény, amire az adott processzusnak várnia kell. Ennek megfelelően az adott szál nem lesz aktív a várakozó feladatlistában. Amikor a socket I/O elérhetővé válik (erről az operációs rendszer szerez először tudomást, a hálózati eszköz driverén keresztül), a feladatütemező újra aktiválja a processzust, és ha a többi feltétel is engedi, beütemezi a következő kontextusváltáshoz. A többszálúság miatt mindig csak azok a szálak kerülnek ki az aktívak listájából, amelyeknél nincs teendő. A többi szál futása az egyik szál blokkolása miatt nem akad meg.

A szálakat a következőképpen szerveztem: a navdata, a video, a config és az AT command interfészhez tartozó socketekhez irányonként egy-egy szálat hoztam létre. Vagyis amíg az egyik irányból (pl. drone->proxy->tablet irány) nincs mit továbbítani, addig is azonnal lereagálható a másik irányban haladó kérés (tablet->proxy->drone irány). Ez mind a 4 portpáron működik, bár az algoritmus kis mértékben eltér a TCP és az UDP továbbítás esetén. A kód úgy készült, hogy ugyanaz az osztály alkalmas a video és config stream továbbítására, csak eltérő socketeket kell átadni a konstruktornak.

Hasonlóképpen az UDP továbbítással foglalkozó szálból négy példány indul, melyek csak a konstruktornak átadott értékben térnek el, hasonlóképpen közös kódot használ a két FTP szimulációs szál is. Ezen kívül egy-egy szálat kapott a TCP Auth, illetve az UDP Auth sockethez kötődő kommunikáció, valamint mindkét FTP szimuláció egy-egy szálat, amelyek tartalmazzák a passzív FTP csatorna kialakítását és az azon folyó kommunikáció programkódját is. Ezzel összesen 12 db másodlagos szál jött létre, az elsődleges szál kezeli a GUI-val kapcsolatos teendőket és a grafikon rajzolást is. Az így kialakult alkalmazás processzorhasználata egy Core i7-4700MQ 2,4 GHz alacsony fogyasztású mobil processzoron forgalomtovábbítás közben élő grafikon rajzolással 6-7,5%-között változik, miközben nem okoz olyan késleltetést a drón irányításában és video folyamában, ami emberi érzéssel észrevehető lenne.

Ez a szigorúbb konfiguráció, mely a kompatibilitás növeléséhez szükséges, ez elvileg bármely irányító alkalmazással képes együttműködni. Az AR.FreeFligh esetében könnyítésként elhagyható a config és a video streamek tablet->proxy->drón irányú szála, mert a SYN üzeneteket a másik szál kezeli. Nem használt az AT parancsok drón->proxy->tablet iránya sem. Ezenkívül elhagyható a két Auth protokollal összefüggő szál is, mert ezeket az AR.FreeFligh nem igényli, vagyis a 12-ből 5 szál opcionális, ami további erőforrás megtakarítást enged.

A wxWidgets a wxThread-nek két típusát kínálja. Az egyik a *joinable* (illeszthető) szál, a másik a *detached* (elkülönített) szál. Ez a POSIX rendszerek mintájára alakult ki, a Win32API-ban csak joinable thread létezik. A fő különbség a memória foglalásban és felszabadításban van. Az illeszthető szál a verem memóriában és a halomban is létrejöhet, nem törli magát. Az Entry() lefutása után annak visszatérési értéke kinyerhető, azonban mindig meg kell várni míg befejezi a futását. Az elkülönített szálat mindig dinamikusan kell létrehozni a halomban, amikor Entry() metódusuk lefut, azonnal felszabadul a szál által használt memória. Ez az alapértelmezett típus a wxWidgets-ben.

A szálat a wxThread osztályból származtatott osztályként hoztam létre, hozzáadva az adott szárhoz éppen szükséges mennyiségű és típusú adattagot (főként a vonatkozó socketeket és címeket). Az adattagok értékét a konstruktorban másolom be az osztályba, valamint az Entry() metódus kódját kellett megadni, ahol a tényleges feladatvégzés történik. A konstruktor és az Entry kivételével mindenben megegyezik a származtatott osztály a szülővel. Az elkülönített szálak használatára példa 3.4.5. pontban

található. A `new` operátor hatására meghívásra kerül a konstruktor, ami átmásolja a szükséges adattagokat, a szál `Run` metódusának hívása lényegében elindítja a szál futását az `Entry` alatt megadott kód végrehajtásának megkezdésével. Az `Entry`-t közvetlenül hívni tilos. A szál fő ciklusában található `TestDestroy()` visszatérési értéke akkor válik igazzá, amikor valahonnan meghívásra kerül a szál `Delete()` metódusa. Ezt követően, ha a kód megfelelően lett elkészítve, az `Entry` hamarosan a futása végéhez ér. Ezután a szál összes erőforrása felszabadul, tilos alkalmazni a `delete` operátort.

A TCP kétirányú kommunikáció megvalósításához szükség volt a szálak közti adatcsere megvalósítására. A `wxWidgets` a szokásos eszköztárral rendelkezik a szálak közti információcseréhez (szemaforok, mutexek, kondicionális változók, postaláda, de alkalmazhatók az események is az elsődleges szál eseménykezelőjén keresztül.) Ezek közül a legkézenfekvőbb a `wxMessageQueue` használata volt, mely egy `thread-safe C++` sablon, tetszőleges típusú objektumok átadására alkalmas, használata adja magát. Ennek segítségével értesítem az TCP továbbítás egyik szálát, ha a másik szálban a kapcsolat bontásra, újra kiépítésre kerül. Így a két adattovábbítási irányra létrejött szál-páros mindig összhangban van, ugyanannak a socketnek a két ellentétes irányát figyelik. Példa 3.4.5-ban olvasható.

3.4.4 Grafikon rajzolás a `wxMathPlot` segítségével

A `wxMathPlot` könyvtár csak 3 elemét emelem ki, melyek itt alkalmazásra kerültek. Az `mpWindow` egy konténer jellegű objektum, ehhez kell az ábrázolandó rétegeket hozzáadni, valamint ez biztosítja az egérrel való kezelhetőséget, zoom, gördítés, eltolás funkciókat. `Addlayer` metódusával lehet a rétegeket hozzáadni. A rétegek tájékoztatják az `mpWindow`-t arról, hogy mely koordinátákkal határolt területet szeretnék ábrázolni. A `Fit` metódus az összes réteg elvárását figyelembe véve kirajzolja a grafikont, úgy minden réteg minden megjeleníteni kívánt adata az ablakon belülrre kerüljön.

Az `mpScaleX` ill. `mpScaleY` a tengelyek és a skála kirajzolásáért felelősek. Ezeknek nincs elvárt területük, ezért nem befolyásolják a `Fit` által ábrázolt területet, viszont a megjelenített területnek megfelelően, dinamikusan skálázzák a tengelyek beosztását.

Az összes grafikon jellegű objektum közül csak az `mpFXYVector`-t emelem ki. Ez két azonos elemszámú, `double` típusú elemeket tartalmazó vektort fogad el, egyik az x koordinátákat, másik az y koordinátákat tartalmazza egymás után. Az azonos pozícióban levő koordinátapárok egy-egy locust adnak a grafikonon, a szomszédos locus-okat pedig vonallal köti össze, ezzel kirajzolva az ábrázolni kívánt függvény képét. Azért ezt választottam, mert elég kézenfekvő a dróntól érkező mérési pontok koordinátáit sorban egy vektor végéhez hozzáfűzni. Amikor meg kívánom jeleníteni a mérési adatpontokat, csak át kell adni a két vektort. Annyi kiegészítést igényelt a függvénykönyvtár, hogy egy új metódust kellett hozzáadnom a gördített megjelenítés egyszerűsítéséhez. Az `mpFXYVector` `SetData` metódusának kell megadni a két említett vektort. Ez a metódus lemásolja a paraméterként kapott két vektort és végigolvassa azok minden elemét, megkeresve x és y maximumát és minimumát, ez alapján állítja be a réteg `BoundingBox` adattagját, melyet az `mpWindow` `Fit` metódusa az összes réteg `BoundingBox`-ával együtt figyelembe vesz.

Ez nem felel meg teljesen a célomnak, miszerint a valós idejű megjelenítésnél az ablakban az utolsó n (n a felhasználó által beállítható érték) másodperchez tartozó grafikonot szeretném megjeleníteni, a grafikon folyamatos balra gördítésével. Egyik lehetőség lenne, hogy a vektor elejéről folyamatosan eldobok annyi elemet, hogy pont az ábrázolni kívánt rész maradjon meg. Ez azonban azzal jár, hogy a felhasználó utólag nem tud visszagörgetni a már kifutott elemekre, illetve nem is túl gyors, mert a vektor első elemének eldobása miatt az egész iterátort újra végig kell futtatni a vektor hátralévő részén. Másodpercenként kb. 30 új elem esetén ez már nem elhanyagolható. A másik lehetőség, hogy a `BoundingBox` meghatározását kell módosítani, ami a `wxMathPlot` kisebb mértékű módosításával jár, de nyílt forrású könyvtárról lévén szó ez nem okoz gondot.

Ezért készítettem egy `SetScrollBar` metódust, hasonló feladatot lát el, de a fentitől eltérően állítja be a `Bounding Boks` méretét. Mivel x vektor szigorúan monoton növekvő, (ez a drón elindítása óta eltelt másodperceket tartalmazza, melyet az egyik `navdata` tagból lehet kinyerni) ezért csak az utolsó elemet kell vizsgálni. Ha ez nagyobb, mint a megjeleníteni kívánt másodpercek száma, akkor ez az új x felső értéke, alsó értéke pedig a maximum- n . Ez gyorsabb eljárás, mint a vektor teljes bejárása és ez alapján a

BoundingBox olyan lesz, hogy a Fit metódus sokszori meghívása egy folytonosan gördülő függvényablakot fog mutatni. A kódrészlet 3.4.5-ös pontban látható.

A mpWindow kapcsán már csak egy dolgot említék meg. A grafikon rajzolásakor a felhasználói felület a wxEventLoop szünete miatt nem reagál kb. 2-400 ms ideig, ez kissé zavaró. Ami sokkal zavaróbb, hogy ilyenkor az adattovábbítás is megáll, vagyis a másodlagos programszálak sem dolgoznak, ettől láthatóan megakad a video stream az okostelefon kijelzőjén. Nem sikerült pontosan tisztázni, hogy a képernyőre való rajzolás miért akasztja meg az egész alkalmazást, még akkor is, ha a rajzolást is másodlagos szálból végeztetem el. Szerencsére azonban sikerült a jelenséget más módon kiküszöbölni. Az mpWindow-ban alapértelmezés szerint ki van kapcsolva a kettős pufferelés (erről a dokumentáció nem szól), de az osztály forráskódjának olvasásakor láttam, hogy könnyen bekapcsolható. Ennek hatására nem közvetlenül a képernyőre, hanem egy memoryDC objektumra készíti el a grafikont, majd a képkocka kirajzolása után ezt teszi ki a képernyőre. Ez sokkal hatékonyabb, villogásmentes megjelenítést biztosít és az említett zavaró jelenség is megszűnt miatta.

3.4.5 Kódrészletek

Ebben a pontban kiragadok néhány részletet a forráskódból, ami az előző pontok alapján valami miatt említésre méltó és rövid magyarázatot próbálok írni az egyes kódrészekhez. A teljes forráskód itt nem kerül bemutatásra, azt a dolgozat mellékletét képező lemezen lehet megtalálni.

- a. Az adatpontok kinyerése a navdata streamből.

```

1. void DronePlotXyFrame::LoadVectsFromFile(wxFile navlogfile)
2. {
3.     uint16_t tag_id, tag_size;
4.     uint8_t ui8val;
5.     uint16_t ui16val;
6.     int16_t si16val;
7.     uint32_t ui32val;
8.     int32_t si32val;
9.     float fl32val;
10.    double locus_y;
11.    float timestamp=0;
12.    wxFileOffset tag_st;

```

```

13.   for (int i=0;i<MAXPLOTS;i++) {
14.       xvect[i].clear();
15.       yvect[i].clear();
16.       Vector[i]->SetData(xvect[i],yvect[i]);
17.   }
18.   if (!navlogfile.Open(wxFile::read)) {
19.       navlogfile.Close();
20.       wxMessageBox("File open error");
21.       return;
22.   }
23.
24.   while (navlogfile.Tell()<navlogfile.Length()-16){
25.       navlogfile.Seek(16,wxFromCurrent);
26.       tag_st=navlogfile.Tell();
27.       navlogfile.Read(&tag_id,2);
28.       while (tag_id!=0xFFFF) {
29.           navlogfile.Read(&tag_size,2);
30.           if (tag_id==1) {
31.               navlogfile.Read(&ui32val,4);
32.               timestamp = (ui32val >> 21) + ((ui32val & 0x1FFFFFF)/
33.                   1000000.0f);
34.           }
35.           else for (int i=0;i<MAXPLOTS;i++)
36.               if ((plot_props[i].tag==tag_id)) {
37.                   if (!plot_props[i].enabled) {
38.                       odlg->CheckBox[i]->Set3StateValue
39.                           (wxCHK_UNDETERMINED);
40.                       continue;
41.                   }
42.                   navlogfile.Seek(tag_st+plot_props[i].offset,
43.                       wxFromStart);
44.                   switch (plot_props[i].decodem) {
45.                       case ui32:
46.                           navlogfile.Read(&ui32val,4);
47.                           locus_y=ui32val;
48.                           break;
49.                       case si32:
50.                           navlogfile.Read(&si32val,4);
51.                           locus_y=si32val;
52.                           break;
53.                       case ui16:
54.                           navlogfile.Read(&ui16val,2);
55.                           locus_y=ui16val;
56.                           break;
57.                       case si16:
58.                           navlogfile.Read(&si16val,2);
59.                           locus_y=si16val;
60.                           break;
61.                       case ui8:
62.                           navlogfile.Read(&ui8val,1);
63.                           locus_y=ui8val;
64.                           break;
65.                       default :
66.                           navlogfile.Read(&f132val,4);
67.                           locus_y=f132val;
68.                           break;

```

```

67.         }
68.         yvect[i].push_back(locus_y);
69.         xvect[i].push_back(timestamp);
70.     }
71.     navlogfile.Seek(tag_st+tag_size,wxFromStart);
72.     tag_st=navlogfile.Tell();
73.     navlogfile.Read(&tag_id,2);
74. }
75.     navlogfile.Seek(6,wxFromCurrent);
76. }
77.     navlogfile.Close();
78.
79.     for (int i=0;i<MAXPLOTS;i++)
80.         Vector[i]->SetScrollBar(plot_fit.miny,plot_fit.maxy,
            plot_fit.window,xvect[i],yvect[i]);
81.     MathPlot1->Fit();
82. }

```

A navdata felépítéséről: az msh-tools fileformátuma egyszerű, az egymás után érkező navdata csomagok adattartalmát tartalmazza konkatenálva. Egy csomag mindig 0x55667788 fejléccel kezdődik, majd 12 bájt drón állapotra vonatkozó információ követi, ezután jönnek a ki-be kapcsolható tagek. A tagek mindig egy 2 bájtos id-vel és 2 bájtos méretinformációval kezdődnek. A tagen belüli mezők offszetje rögzített, a libARDrone navdat_keys.h fájljában leírt struct-ok szerint. Indítás után a navdata bootstrap módban van, ekkor egyetlen tag aktív, a checksum, aminek id-je 0xFF. Ebből kilépve csak a navdata_demo_tag aktív, aminek id-je 0. Az AT parancsok segítségével tetszőleges navdata tagek bekapcsolhatók. Az időbélyek az 1-es tagben szerepel. Felső 11 bitje az egész másodperceket adja, alsó 21 bitje a mikroszekundumok számát. Ebből alakítható ki a timestamp float típusú értéke. Az összes többi tag közül kikeressük azokat, amelyek szerepelnek a grafikonban, és a tag pozícióján belül a megfelelő címről kimásolódnak a releváns mezők bájról bájtra. A típuskonverziót a switch-case konstrukció végzi a beállított adatábrázolási formának megfelelően. A külső while ciklus a fájlban mindig a következő navdata üzenet pozíciójához ugrik, amíg a fájlvég nincs túl közel. A belső while ciklus minden megtalált tagre lefut, ha releváns, akkor másolás is történik. A tagek láncolt lista jellegűek. Az első mindig ugyanott kezdődik, a következő kezdőcíme pedig az aktuális tag méretének megfelelően nagyobb. Az utolsó tag mindig a checksum, aminek id-je 0xFF, és ez mindig jelen van. Az x koordináták lehetnének közösek, de azért

tartozik minden grafikonhoz saját x koordináta sokaság, mert így garantálható, hogy azonos számú x és y koordináta tartozzon minden egyes grafikonhoz, akkor is, ha egy-egy üzenet az UDP továbbítás miatt kimarad, vagy sérül. Ilyenkor a grafikon képe továbbra is illesztett marad, csak egy-egy minta hiányozni fog belőle.

Az élő grafikonrajzolás kódját nem idézem, az nagyon hasonló, csak a külső while ciklus hiányzik, mert az UDP továbbító szál egyszerre mindig egy üzenetet ad át. Az a metódus nem fájlal, hanem char* tömbben kapja meg az üzenet tartalmát. A MAXPLOTS define-ról: A dialógusablakban dinamikusan hozom létre grafikonok beállítására szolgáló komponenseket, ennek megfelelő számú réteget kap az mpWindow is. A kinyerő algoritmusnak sem számít, hány grafikkal kell dolgozni. A widgetek tömbbe szervezettek így könnyen bejárható az összes kezelőszerv. Nincs elméleti korlát, a 6-os korlátot gyakorlati megfontolásból állítottam, be mert sok grafikon áttekinthetetlenné teszi a megjelenítést, nincs algoritmikus akadály annak, hogy a #define MAXPLOTS 6 helyett más értéket adjak meg.

- b. A 3.4.3 pont szerint egy szál teljes leírásán keresztül mutatom be a wxThread-ek használatát.

```

1. class UDP_filtering_fwdThread : public wxThread {
2. public:
3.     UDP_filtering_fwdThread(DroneproxyFrame *handler, wxDatagramSocket* l
st_sock, wxDatagramSocket* fwd_sock, wxIPv4address* lst_peer_addr, wxIPv4
address* fwd_peer_addr, int* num_pkt, int* lastp_size)
4.         : wxThread(wxTHREAD_DETACHED)
5.         { m_pHandler = handler;
6.           listen_on = lst_sock;
7.           forward_via = fwd_sock;
8.           listen_peer = lst_peer_addr;
9.           forward_peer = fwd_peer_addr;
10.          num_packets = num_pkt;
11.          last_pkt_size = lastp_size;
12.        }
13.     ~UDP_filtering_fwdThread();
14.
15. protected:
16.     wxDatagramSocket *listen_on, *forward_via;
17.     wxIPv4address *forward_peer, *listen_peer;
18.     int* num_packets, *last_pkt_size;
19.     virtual ExitCode Entry();

```

```

20.   DroneproxyFrame *m_pHandler;
21. };

```

Az osztály definíciójában helyben kifejtettem a konstruktort is, mely csak az adattagok másolásából áll.

```

1. wxThread::ExitCode UDP_filtering_fwdThread::Entry()
2. {
3.     char buffer [4096];
4.     char* pos;
5.     int datasize=0;
6.
7.     listen_on->SetTimeout(1);
8.     while (!TestDestroy()) {
9.         listen_on->Read(buffer,sizeof(buffer));
10.        datasize=listen_on->LastCount();
11.        if (datasize>0) {
12.            listen_on->GetPeer(*listen_peer);
13.            if (pos=strstr(buffer,"general:navdata_demo",
14.                           "\"TRUE\""))
15.            {
16.                strncpy(pos,"general:navdata_demo",
17.                        "\"FALSE\\r",31);
18.                datasize++;
19.            }
20.            if (pos=strstr(buffer,"general:navdata_options\"")) strncpy(pos,"general:navdata_options\", \"4294967295\\r",
21.                                   ,39);
22.            forward_via->SendTo(*forward_peer,buffer,datasize);
23.            (*num_packets)++;
24.            if (datasize==forward_via->LastCount())
25.                *last_pkt_size=datasize;
26.            else
27.                *last_pkt_size=-1;
28.        }
29.    }
30.    m_pHandler->TextCtrl1->AppendText(wxString::Format
31.                                       ("UDP thread ended!\\n"));
32.    return (wxThread::ExitCode)0;
33. }

```

Ez a szál folyamatosan olvasni próbálja az okostelefonról érkező AT parancsokat, amikor egy parancs a navtaga tag-eket próbálja módosítani, akkor lecserélem a parancsot olyanra, ami az összes mezőt bekapcsolja. Egyébként a parancs módosítás nélkül halad tovább a proxy-drone interfészen keresztül a drón felé.

```

1. m_pThread_td_cmd = new UDP_filtering_fwdThread(this,cmd_t_socket,cmd_d_socket,
2.   t_remote_cmd,d_remote_cmd,td_cmd_pkt_num,td_cmd_pkt_size);
3. m_pThread_td_cmd->Create();

```

```

4.
5. if ( m_pThread_td_cmd->Run() != wxTHREAD_NO_ERROR )
6.     {
7.         TextCtrl1->AppendText(wxString::Format("Thread run error!\n"));
8.         delete m_pThread_td_cmd;
9.         m_pThread_td_cmd = NULL;
10.    }

```

A thread-re mutató pointer létrehozása new operátorral, majd a Create metódus meghívása allokalja a szükséges memóriát, majd lefuttatja a konstruktort. A Run meghívása elindítja az Entry-ben kifejtett folyamat kódját.

```

1. m_pThread_td_cmd->Delete();

```

Ezzel a hívással lehet a threadet leállítani. A blokkoló I/O műveleteket azért szakítom meg ritkán, 1 másodperces időközökkel, hogy a TestDestroy lefuthasson. Ez a delete hatására hamis értéket vesz fel, az Entry végtelen ciklusa megszakad. A futás befejezése után automatikusan felszabadul minden lefoglalt erőforrás, további művelet nem szükséges. (detached thread miatt)

c. A 3.4.3 pontban említett kapcsolattartás a TCP továbbítás két thread-je között az alábbiak szerinti.

```

1. wxThread::ExitCode TCP_Rq_forwardThread::Entry()
2. {
3.     char buffer [4096];
4.     int16_t datasize=0;
5.
6.     while (!TestDestroy())
7.     {
8.         if (!connection->IsConnected())
9.             if (listen_on->WaitForAccept(1,0))
10.            {
11.                connection=listen_on->Accept(true);
12.                connection->GetPeer(*initiating_peer);
13.                reverse_thread_cpvt->sessionqueue->Clear();
14.                reverse_thread_cpvt->sessionqueue->Post(connection);
15.            }
16.        if (!forward_via->IsConnected()) {
17.            forward_via->Connect(*forward_peer,false);
18.            reverse_thread_cpvt->cliqueue->Clear();
19.            reverse_thread_cpvt->cliqueue->Post(forward_via);
20.        }
21.        if (forward_via->WaitForRead(1,0))

```

```

22.     {
23.         forward_via->Read(buffer, sizeof(buffer));
24.         datasize=forward_via->LastCount();
25.         (*num_packets_bkw)++;
26.         connection->Write(buffer,datasize);
27.         if (datasize==connection->LastCount())
28.             *last_pkt_size_bkw=datasize;
29.         else
30.             *last_pkt_size_bkw=-1;
31.     }
32. }
33. m_pHandler->TextCtrl1->AppendText
    (wxString::Format("TCP request thread ended!\n"));
34. return (wxThread::ExitCode)0;
35. }

1. wxThread::ExitCode TCP_Rpl_forwardThread::Entry()
2. {
3.     char buffer [4096];
4.     int16_t datasize=0;
5.     sessionqueue = new sessionqueueclass();
6.     cliqueue = new cliqueueclass();
7.
8.     while (!TestDestroy())
9.     {
10.         if (!connection->IsConnected())
11.             sessionqueue->ReceiveTimeout(1000,connection);
12.         if (!forward_via->IsConnected())
13.             cliqueue->ReceiveTimeout(1000,forward_via);
14.
15.         if (connection->WaitForRead(1,0)) {
16.             connection->Read(buffer, sizeof(buffer));
17.             datasize=connection->LastCount();
18.             forward_via->Write(buffer,datasize);
19.             (*num_packets)++;
20.             if (datasize==forward_via->LastCount())
21.                 *last_pkt_size=datasize;
22.             else
23.                 *last_pkt_size=-1;
24.         }
25.     }
26.     m_pHandler->TextCtrl1->AppendText
        (wxString::Format("TCP reply thread ended!\n"));
27.     delete sessionqueue;
28.     delete cliqueue;
29.     return (wxThread::ExitCode)0;
30. }

```

Az egyik szál érzékeli a kapcsolatot megváltozását, amikor új wxSocketBase jön létre az Accept metódus miatt, ilyenkor az új socketet a postaládába teszi. A másik szál, amikor kapcsolathibát észlel, a postaládából várja az új kapcsolatinformációkat. A szálak közti összhang így megvalósul. Ezt remekül lehet tesztelni, amikor elaltatom

az okostelefont. Ébresztéskor az AR.FreeFligh kerül a képernyőre, de minden TCP kapcsolatot újraépít, természetesen más forrással. Az algoritmus jól vizsgázott a gyakorlatban.

d. Az FTP emuláció kódja:

```

1. wxThread::ExitCode FTPemu_Thread::Entry()
2. {
3.     char buffer [4096];
4.     int16_t datasize=0;
5.     wxSocketBase* connection = new wxSocketBase;
6.     wxSocketBase* pasv_conn = new wxSocketBase;
7.     wxIPv4address temp;
8.     bool quit=false;
9.
10.    connection->SetTimeout(1);
11.    pasv_conn->SetTimeout(1);
12.
13.    while (!TestDestroy())
14.    {
15.        while (!connection->IsConnected() && !TestDestroy()) {
16.            pasv_conn->Close();
17.            if (listen_on->WaitForAccept(3,0)){
18.                connection=listen_on->Accept(true);
19.                connection->GetPeer(*initiating_peer);
20.                connection->Write("220\x20
                Operation successful\r\n",26);
21.                (*num_packets_bkw)++;
22.                if (26==connection->LastCount())
23.                    *last_pkt_size_bkw=26;
24.                else
25.                    *last_pkt_size_bkw=-1;
26.            }
27.        }
28.
29.        if (connection->WaitForRead(1,0)) {
30.            connection->Read(buffer,sizeof(buffer));
31.            (*num_packets)++;
32.            if (!(*last_pkt_size=connection->LastCount())) continue;
33.            datasize=0;
34.            if (strncmp(buffer,"USER anonymous",14)==0)
35.            {
36.                strncpy(buffer,"230\x20operation successful\r\n",26);
37.                datasize=26;
38.            }
39.            if (strncmp(buffer,"PASV",4)==0)
40.            {
41.                listen_on->GetLocal(temp);
42.                if (temp.Service()==21)
43.                    strncpy(buffer,"227\x20PASV ok (192,168,1,1,132,60)\r
                    \n",34);
44.            }
            else

```

```

45.         strncpy(buffer,"227\x20PASV ok (192,168,1,1,132,59)\r\n",34);
46.         datasize=34;
47.     }
48.     if (strncmp(buffer,"TYPE I",6)==0)
49.     {
50.         strncpy(buffer,"200\x20operation successful\r\n",26);
51.         datasize=26;
52.     }
53.     if (strncmp(buffer,"SIZE version.txt",16)==0)
54.     {
55.         strncpy(buffer,"213 6\r\n",7);
56.         buffer[3]='\x20';
57.         datasize=7;
58.     }
59.     if (strncmp(buffer,"RETR version.txt",16)==0)
60.     {
61.         strncpy(buffer,"150\x20Opening BINARY connection for version.txt (6 bytes)\r\n",57);
62.         datasize=57;
63.         connection->Write(buffer,datasize);
64.         (*num_packets_bkw)++;
65.         *last_pkt_size_bkw=datasize;
66.
67.         if (!pasv_conn->IsConnected())
68.         {
69.             pasv_ftp_listen->WaitForAccept(1,0);
70.             pasv_conn=pasv_ftp_listen->Accept(true);
71.         }
72.         pasv_conn->Write("2.4.8\r\n",6);
73.         (*num_packets_bkw)++;
74.         *last_pkt_size_bkw=pasv_conn->LastCount();
75.         strncpy(buffer,"226\x20operation successful\r\n",26);
76.         datasize=26;
77.     }
78.     if (strncmp(buffer,"QUIT",4)==0)
79.     {
80.         strncpy(buffer,"221\x20operation successful\r\n",26);
81.         datasize=26;
82.         quit=true;
83.     }
84.     if (strncmp(buffer,"CWD /boxes",10)==0)
85.     {
86.         strncpy(buffer,"250\x20operation successful\r\n",26);
87.         datasize=26;
88.     }
89.     if (strncmp(buffer,"LIST",4)==0)
90.     {
91.         strncpy(buffer,"150\x20Directory listing\r\n",23);
92.         datasize=26;
93.         connection->Write(buffer,datasize);
94.         (*num_packets_bkw)++;
95.         *last_pkt_size_bkw=datasize;
96.         if (!pasv_conn->IsConnected())
97.         {
98.             pasv_ftp_listen->WaitForAccept(1,0);

```

```

99.         pasv_conn=pasv_ftp_listen->Accept(true);
100.     }
101.     strncpy(buffer,"226\x20Operation successful\r\n",26);
102.     datasize=26;
103. }
104. connection->Write(buffer,datasize);
105. (*num_packets_bkw)++;
106. if (datasize==connection->LastCount())
107.     *last_pkt_size_bkw=datasize;
108. else
109.     *last_pkt_size_bkw=-1;
110. if (quit)
111. {
112.     pasv_conn->Close();
113.     connection->Close();
114.     quit=false;
115. }
116. }
117. }
118. m_pHandler->TextCtrl1->AppendText
    (wxString::Format("FTP thread ended!\n"));
119. delete connection;
120. delete pasv_conn;
121. return (wxThread::ExitCode)0;
122. }

```

Az eljárás elején az FTP kliens kapcsolódását várom, amikor egy kérés beérkezik, azonnal válaszolni kell egy sikert jelző üzenettel. Ha van kapcsolat akkor a beérkező kérdésnek megfelelő választ kell adni, erről gondoskodik a sok strncpy és strncmp híváspár. Adott kérdésre mindig ugyanaz a válasz a 3.2. pontban leírtak szerint. A két FTP szál ugyanakkor az osztálynak két példánya, csak két különböző socket kerül átadásra a konstruktornak. Ez az osztály „érti” mindkét fajta FTP párbeszédet. Egyetlen különbség, hogy a két szél két eltérő portszámú passzív csatornát nyit, mert ezek ritkán ugyan, de időben átfedhetnek. Ezért a megkülönböztetés a 21-es és az 5551-es FTP port között (lásd listen_on->GetLocal, tmp.Service). A passzív csatornát nem kell olvasni, a kliens tudja milyen jellegű adatot vár, Accept után egyből lehet írni a választ. Az if vizsgálatok után a megfelelőválasz a pufferben van, azt minden esetben csak ki kell írni a socketre. Ha quit kérés érkezett, akkor válasz elküldése után a socketeket még be is kell zárni. Ez a párbeszéd szimuláció frappánsan helyettesíti az FTP továbbításával járó munkát, úgy hogy közben valódi funkcióvesztés nem történik.

e. A 3.4.4. pontban említett kiegészítés a wxMathPlot mpFXYVector osztályához.

```
1. void mpFXYVector::SetScrollBar(int miny, int maxy, int width, std::vector<double> &xs, std::vector<double> &ys)
2. {
3.     m_xs = xs;
4.     m_ys = ys;
5.     if (xs.size()>0) {
6.         if (xs.back()<=width){
7.             m_minX=xs.front();
8.             m_maxX=m_minX+width;
9.         }
10.        else {
11.            m_maxX=xs.back();
12.            m_minX=m_maxX-width;
13.        }
14.        m_minY=miny;
15.        m_maxY=maxy;
16.    } else {
17.        m_minX = -1;
18.        m_maxX = 1;
19.        m_minY = -1;
20.        m_maxY = 1;
21.    }
22. }
```

A miny, maxy, és width változók a felhasználó preferenciája alapján az egyik dialógusablak beállításai szerint kerülnek átadásra ennek a függvénynek.

4 A repülések elemzése, kiértékelése

A következő grafikon egy nem túl összetett repülési folyamat rögzített adatai közül mutat néhányat. A repülést egy Android-os okostelefonnal vezetem az AR.FreeFlight v2.4 futtatásával, a grafikus megjelenítésről az msh-tool Drone Diagnostics v3 gondoskodott. A repülési folyamat 5 elkülöníthető szakaszból áll:

- Felszállás
- Stabilizáció és korrekciók
- Lebegés
- Magneto kalibráció
- Leszállás

A magnetométer kalibrációja során a firmware vezérli a járművet, egy 360°-os függőleges tengely körüli fordulatból áll.

A magnetométer (digitális iránytű) kiolvasáson kívül mind a négy motor PWM értéke került ábrázolásra. A kijelzett pulzusszélesség moduláció (Pulse Width Modulation) értékek arányosak azzal az időhányaddal, amekkora hányadot a motor bekapcsolt állapotban tölt el időegységenként, tehát arányos a motor által felvett villamos teljesítménnyel (a valóságban ez csak közelítőleg igaz), így a görbe alatti terület arányos a motor által elfogyasztott teljes villamos energiával.

A PWM egy szokásos módja az egyenáramú motorok meghajtásának, fordulatszám szabályozásának, amikor csak egy rögzített feszültségű forrás áll rendelkezésre. Az AR.Drone motorjai 8 bites PWM-mel vannak meghajtva, vagyis 256 különböző kitöltési tényező valósítható meg. A 0 érték azt jelenti, hogy a motor ki van kapcsolva, a 255-ös érték esetén pedig teljes teljesítménnyel dolgozik. Egy köztes érték, pl. 63 azt jelenti, hogy az apró időszelvények mindegyikében közel egynegyed időarányban van meghajtva, vagyis a névleges villamos teljesítmény negyed részével működik. Fontos megjegyezni, hogy ezek a belső szabályozó algoritmus által előírt teljesítmény nagyságok, amik az adott körülmények, környezeti hatások mellett a drónt a kívánt helyben és helyzetben tudják tartani, figyelemmel az algoritmushoz eljutó szenzoradatokra és visszacsatolásokra. Ezek nem a valós fordulatszám adatok, az AR.Drone nem rendelkezik fordulatszám-mérővel. A grafikonra (7. ábra) nézve az első szembeötlő dolog, hogy lebegés közben azonos fordulatszámokra, ezek alapján azonos

elvárt motorteljesítmény értékekre számítanánk, az algoritmus által előírányzott teljesítmény értékek mégis eltérnek motoronként. Az algoritmus úgy találta, hogy pontosan ezek az értékek tartják helyben a járművet (ez így is történt, a mérés e szakaszában a drón stabil volt). Az aszimmetrikus meghajtás melletti stabil lebegésből azonnal valószínűsíthető, hogy a 4 rotor mechanikai állapota nem egyforma, ahogy ez a 2. fejezetben kifejtettekéből is következik. A beágyazott szabályozó algoritmus így kompenzál az aszimmetriára.



7. ábra – repülési adatok görbéi, javítás előtt

Következzen néhány észrevétel a repülés 5 szakaszáról. Az első szakaszban a felszállásnál látható, hogy minden motor kb. 70-es PWM értékkel kezd, majd mind a 4 hajtás együtt növekszik addig a pontig, amíg a jármű el nem szakad a talajtól. Innentől szerez tudomást az algoritmus arról, hogy az egyforma értékek nem tartanak vízszintesen a járművet. Ettől a ponttól a repülés végéig megfigyelhető a késsel és pirossal rajzolt motorteljesítmény ofsztetje. A 80 cm körüli magasság elérésekor a rotorok lelassulnak egy pillanatra, majd az algoritmus megtalálja azt a fordulatszám kombinációt, amivel a drón állandó magasságban tud maradni.

A második szakaszban a drónt kézi irányítással a szoba közepére vezetem, a grafikonon látható tüskék a kezelőfelületen történt beavatkozásoknak felelnek meg. A harmadik egy viszonylag stabil szakasz, ennek során elengedtem az okostelefon érintőképernyőjét. Ilyenkor a jármű lebegésre vált, próbálja tartani helyét és helyzetét a

belső szenzorok segítségével, illetve a lefelé tekintő kamerakép minta-eltolódás elemzésének segítségével.

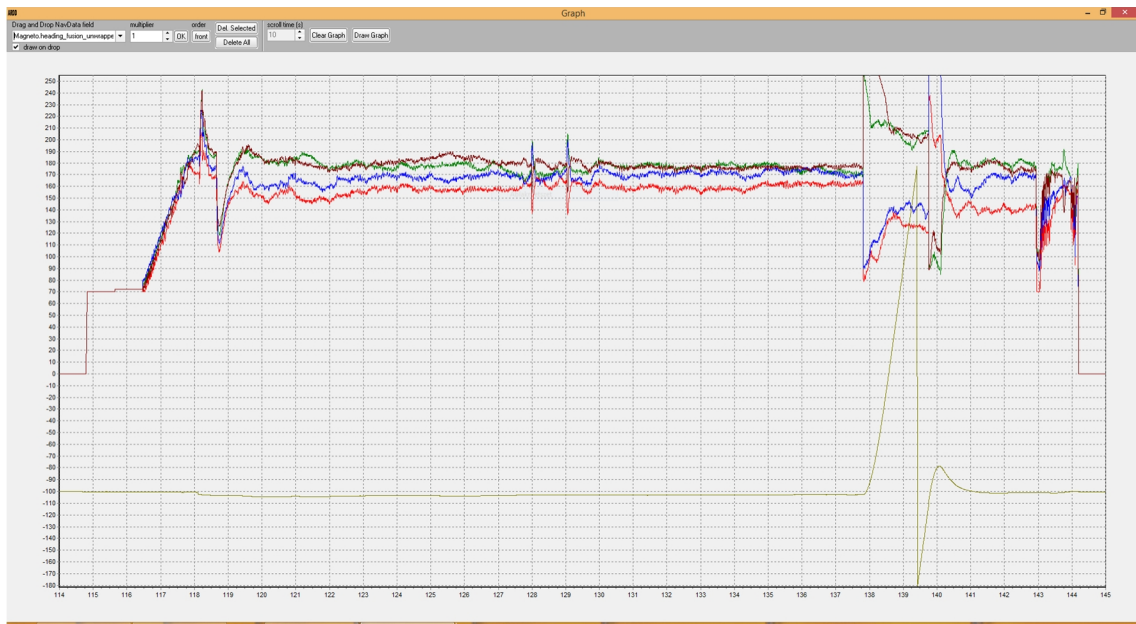
A negyedik szakasz a firmware által vezérelt fordulat. Az itt ábrázoltak megerősítik a 2. fejezetben tárgyaltakat a kvadrotorok „yaw” mozgásáról. Az is látszik, hogy a teljes fordulat végén, a magneto érték túllendülésekor az 1. és 3. illetve a 2. és 4. rotor aránya invertálódik, ellenirányú forgatással próbál a jármű a levegőben fékezni. Az ehhez hasonló adatok elemzésével információhoz lehet hozzájutni az AR.Drone-ban működő szabályozók paramétereiről. A magnetometer görbéje a fordulat közben megszakad és pozitívból negatívba csap át. Ez a mágneses északtól való jobbra vagy balra történő kitérés ábrázolási módja a drón algoritmusában. Egy másik említésre méltó jelenség a függőleges tengely körüli forgás közben a PWM értékek telítődése. A felső határon a telítődés 255-nél a motor teljesítménykorlátja miatt következik be, az alsó határon 70-nél a szaturációt valamilyen belső korlát okozhatja. Az algoritmus nem engedi repülés közben 70 alá csökkenni a motor teljesítményt, felszállásnál is láttuk, hogy 70-nél kezdődött a szabályozás. A telítődés minden bizonnyal azért következik be, mert a mechanikai aszimmetria és a „yaw” mozgás miatti szándékos aszimmetria együttesen túl komoly hatást okoznak, amit az algoritmus már nem tud működési határain belül kezelni.

Ismételt stabilizáció után az ötödik szakaszban, landolás előtt látható, hogy az összes motor teljesítmény a kitüntetett 70-es értékre esik vissza, majd közvetlenül a becsapódás előtt egy rövid erőteljes felfelé történő gyorsulás látszik, amelynek az ütközés tompítása a célja. Ezt követően a rotorok leállnak.

Hogy megerősítsem a dolgozatban tárgyaltakat a mechanikai degradációról, dinamikáról és diagnosztikáról elvégeztem egy újabb mérést (8. ábra). Rögzítettem egy hasonló repülést, miután kisebb javításokat végeztem a piros görbének megfelelő rotoron.

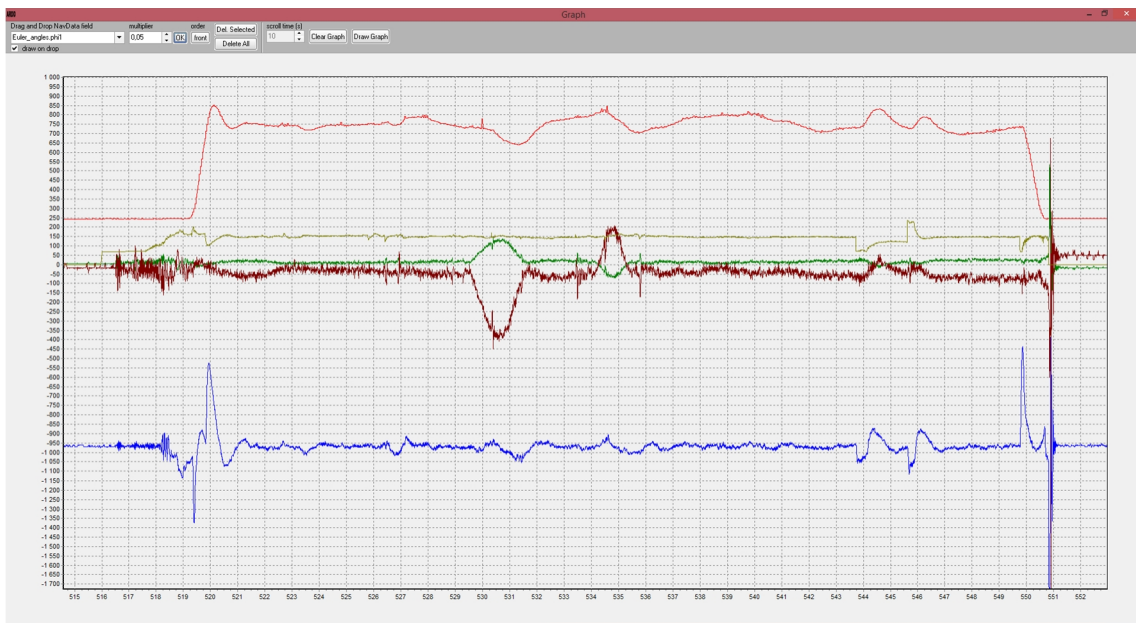
A sérült fogaskereket (3. ábra) egy újra cseréltem és a torzult bronzgyűrűket (5. ábra) megfordítottam. Nem cseréltem, csak megfordítottam, hogy a kevésbé sérült oldala essen kívülre, így az jobban tud erőt kifejteni a kitérni, dőlni szándékozó motortengellyel szemben. Ebben a kísérletben mást nem javítottam, a hatások mégis szembetűnőek.

- A piros színnel ábrázolt motor ofszetje lecsökkent, a kéké majdnem teljesen megszűnt.
- A lebegés során a görbék stabilabbak, „higgadtabbak”.
- A telítődés láthatóan csökkent.



8. ábra- repülési adatok görbéi, javítás után

Érdeemes megjegyezni, hogy csak egy rotort javítottam, mégis ez mind a kék, mind a piros PWM görbe eltolására és telítődésére hatással volt. Ez egybehangzó azzal a ténnyel, hogy a kvadrotor manőverezési műveletek aktuátor párokra épülnek, ahogy azt a 2. fejezetben leírtam.



9. ábra – Példa görbék, adatmezők közti összefüggések

Hogy betekintést nyújtsak a kinyerhető adatok széles skálájába, bemutatok egy példát (9. ábra) egy repülés néhány jellemzőjével a sok közül. Ez a repülési ciklus

felszállásból, balra, majd jobbra oldalazásból, lebegésből, a magnetométer kalibrációjából és a leszállásból áll. Az ábrázolt görbék fentről lefelé:

- Magasság (ultrahangos szenzor alapján)
- Motor teljesítmény (csak az 1. motoré)
- Tömegközéppont gyorsulása (y tengely mentén)
- φ szög (x tengely körüli elfordulás)
- Függőleges gyorsulás (z tengely)

Figyeljük meg az összefüggést a motor teljesítmény, a függőleges gyorsulás és a magasság között a fel- és leszállás során, valamint a φ és a_y között az oldalazás során.

Megfigyelhető, hogy a manőverek során („roll” és „yaw” esetében is) a jármű kissé ingadozik, ezt mutatja a magasságmérés és a függőleges gyorsulás görbéje is, de stabilizálódik, amint a manőver befejeződik. Látható továbbá, hogy az 1. motor nem játszik szerepet az oldalazásban (ebben a fázisban az értéke teljesen stabil), de megváltozik a fordulatszáma a függőleges tengely körüli forgásnál, úgy ahogy ezt elvárnánk. A φ szög zajos és ingadozó a felszállásnál, mert a jármű nem teljesen vízszintes (felszállás közben el is szokott „osonni” valamelyik irányba), amíg el nem éri a felszállási magasságot és stabilizálja magát. A kismértékű degradációt a beépített algoritmus lebegéskor kompenzálja, de az ilyen manőver közbeni jelek, akárcsak a motorteljesítmény kiértékelése, árulkodnak arról is, ha a degradáció még csekély. Bár a leszállás nem okoz szembetűnő megrázkódtatást a járműnek, azért a gyroszkópok és gyorsulásérzékelők nagy tüskével jelzik a finom ütközést is (φ és a_z görbék).

5 Konklúzió

A dolgozat célja, hogy bemutasson egy eljárást, amivel a repülések részletes adatai kinyerhetők az AR.Drone 2.0 NavData adatfolyamából, és rávilágítson, ez hogyan lehet hasznos mechanikai problémák informatikai módszerekkel való gyors azonosításához. Megmutattam a megvalósítás részleteit és azt, hogyan alkalmazható sikeresen az eljárás transzparens módon, akkor is, amikor a harmadik fél által készített repülésvezérlő szoftver nem módosítható.

Elsődleges célján kívül a DroneProxy hasznosnak bizonyul eredeti tervezésből adódó hálózati kötöttségek leküzdésében, legyen szó a hatótáv kiterjesztéséről vagy az IP hálózatok nyújtotta flexibilitás kiaknázásáról.

A drónok szétszerelése, az alkatrészek vizsgálata és a grafikonok kiértékelése után az elméletből eredő elvárások beigazolódtak, eredményes javításokat tudtam végezni. Folyamatban van a továbbfejlesztett alkatrészek beszerzése (fém fogaskerekek, golyós csapágyak, stb..), szándékomban áll dokumentálni ezek jövőbeli állapotromlását is, hogy végül olyan adatbázis készülhessen, melyben megfeleltethetők a repülési adatok jellegzetességei és a drón viselkedésében tapasztalt eltérések egy bizonyos specifikus alkatrészhibának.

Lehetőséget látok abban, hogy a tesztelési eljárás automatizálásra kerüljön, a drónnak számítógép vezérelten kellene végigrepülnie egy bizonyos szekvenciát, miközben gyűjti és rögzíti a szükséges adatokat, emberi beavatkozás nélkül. Az msh-tools Drone Diagnostics már helyettesíthető a saját grafikon rajzoló megoldással, így nincs szükség 3. fél alkalmazásának használatára, a DroneProxy csak nyíltforrású kóddal van kapcsolatban. Az új vizualizációs eljárás, mellyel a DroneProxy-t kiegészítettem, élő grafikonrajzolást is biztosít. Ezzel lehetőség nyílik arra, hogy ha valamilyen szembetűnő jellegzetesség rajzolódik ki a grafikonon, akkor a teszt azonnal ismételhető, a jelenség kihangsúlyozható, anélkül, hogy a diagnosztikai folyamat közben a drónnal le kellene szállni, majd fájlokat megnyitni, stb..

A parancsszűrő működését bemutattam a 3.4. pontban. Ez az algoritmus lehetővé teszi, hogy a drónt egy ultrahangos és/vagy lidar szenzorral felszerelve érzékelhetővé váljanak a környezetben található akadályok. A DroneProxy valós idejű parancselfogó módszere alkalmazható olyan irányító alkalmazástól származó parancsok kiszűrésére, amelyek közvetlenül a jármű akadálynak csapódásához vezetnek. Ez egy transzparens biztonsági réteg lehet, mely együtt használható tetszőleges repülésvezérlővel. Nem csak a drónt először vezetni próbálók számára, hanem a különböző irányító algoritmusok fejlesztése közben is hasznosnak bizonyulhat egy eljárás, mely csökkenti az ütközések, lezuhanások számát. Ezzel az egész drón diagnosztika és javítás irányultságú projekt, melyet a gyakori ütközések, zuhanások tettek megalapozottá, fordulatot vehetne az ütközés megelőzés irányába.

6 Hivatkozások

[1] AR.Drone SDK 2.0.1.

Piskorski, S.; Brulez, N.; Eline, P.; D'Haeyer, F.:

AR.Drone Developer guide, Parrot SA, 2013

<https://projects.ardrone.org/projects/show/ardrone-api> (látogatás ideje: 2014.02.20., 2014.10.08.)

[2] Mikhail Sharonov: ARDrone Diagnostics v.3, 2012

<http://www.msh-tools.com/ardrone/arddiag.html> (látogatás ideje: 2014.10.04.)

[3] Drone Science: How quadrotors work

StemScopes Texas, 2013

<http://stemscopes.com/blog/2013/3/13/drone-science-how-quadrotors-work> / (látogatás ideje: 2014.10.18.)

[4] Bristeau, P-J.; Callou, F.; Vissiere, D.; Petit, N.: The Navigation and Control technology inside the AR.Drone micro UAV, p. 1478-1482

IFAC World Congress, Milano, Italy, 2011.

[5] How to troubleshoot your AR Drone

<http://www.droneflyers.com/2013/03/how-to-troubleshoot-your-ar-drone-1-0-and-2-0-after-a-crash/> (látogatás ideje: 2014.10.06.)

[6] Parrot ASTEROID Developers Zone

ARDroneLib source code

<https://devzone.parrot.com/> (látogatás ideje 2014.10.02.)

[7] wxWidgets documentation

wxSocketBase, wxDatagramSocket, wxSocketClient, wxSocketServer, wxThread, wxEvent, wxFile, wxFrame, valamint számos osztály referenciája

<http://docs.wxwidgets.org/trunk/> (látogatás ideje: 2014.09.28 - 2014.11.20-ig rendszeresen)

<https://wiki.wxwidgets.org> (látogatás ideje: 2014.09.28 - 2014.11.20-ig rendszeresen)

[8] wxMathPlot framework documentation

mpWindow, mpLayer, mpFXYVector, és további osztályok referenciája

<http://wxmathplot.sourceforge.net/docs> (látogatás ideje: 2014.11.01-2014.11.20-ig rendszeresen)

[9] On-the-fly acquisition of flight data for quad copter

Schäffer Zs, Müller P, Várady G.

10th International Miklós Iványi PhD/DLA Symposium, Pécs, 2014

[10] Intelligent Flight Control of an Autonomous Quadrotor

Syed Ali Raza and Wail Gueaieb

University of Ottawa, Canada

Motion Control (Edited by Federico Casolo), 2010. jan.

ISBN 978-953-7619-55-8

[11] Hacking and securing the AR.Drone 2.0 quadcopter: Investigations for improving the security of a toy

Johann-Sebastian Pleban, Ricardo Band, Reiner Creutzburg

Fachhochschule Brandenburg (Germany)

Proc. SPIE 9030, Mobile Devices and Multimedia: Enabling Technologies, Algorithms, and Applications 2014