Széchenyi István Egyetem
Gépészmérnöki, Informatikai és
Villamosmérnöki Kar

SZAKDOLGOZAT

Gulyás Péter

Villamosmérnöki BSc szak

2018

SZÉCHENYI ISTVÁN EGYETEM
GÉPÉSZMÉRNÖKI, INFORMATIKAI ÉS VILLAMOSMÉRNÖKI KAR

SZAKDOLGOZAT

Inerciamérés és pozíció meghatározás az ROS-ben

Gulyás Péter

Villamosmérnöki BSc szak
Automatizálás szakirány

2018

Feladat-kiíró lap szakdolgozathoz

Hallgató adatai
Név: Gulyás Péter *Neptun-kód: HAFSIW*
Szak: Villamosmérnöki BSc.
Specializáció: Automatizálás szakirány *Tagozat: nappali*

A szakdolgozat adatai
Kezdő tanév és félév: 2017/18 2.félév
Nyelv: magyar
Típus: nyilvános

Inerciamérés és pozíció meghatározás az ROS-ben

Feladatok részletezése:

1) Az inerciamérés módszerei
2) A Robot Operációs Rendszer ismertetése
3) Felhasznált hardware-ek ismertetése
    a. Bosch BNO055
    b. Nvidia Jetson TX2
4) Szenzor integráció az ROS-ben
5) Navigációs mérések a BNO055-ös chippel
6) Tesztelés egy közúti járműben
    a. CAN kommunikáció felépítése

Belső konzulens adatai
Név: Dr. Ballagi Áron
Tanszék: Automatizálási
Beosztás: Tanszékvezető

Külső konzulens adatai
Név: Kőrös Péter
Munkahely: Járműipari Kutatóközpont
Beosztás: Kutatómérnök

Győr, 2018 január 12.

_____
Belső konzulens: Dr. Ballagi Áron

_____
Külső konzulens: Kőrös Péter

_____
Automatizálási Tanszék
Dr. Ballagi Áron

Nyilatkozat

Alulírott, Gulyás Péter (HAFSIW), Villamosmérnöki BSc. szakos hallgató kijelentem, hogy a [*Inerciamérés, és pozíció meghatározás az ROS-ben*] című szakdolgozat feladat kidolgozása a saját munkám, abban csak a megjelölt forrásokat, és a megjelölt mértékben használtam fel, az idézés szabályainak megfelelően, a hivatkozások pontos megjelölésével.

Eredményeim saját munkán, számításokon, kutatáson, valós méréseken alapulnak, és a legjobb tudásom szerint hitelesek.

Győr, 2018. 11. 19.

_____

Gulyás Péter

Kivonat

Inerciamérés és pozíció meghatározás az ROS-ben

Napjainkban a közúti járművek navigációja bonyolult elektronikával, és szoftveres háttérrel rendelkezik, melyek főként GPS jelekre és offline térképekre hagyatkoznak. Ezek legnagyobb problémája, hogy GPS jel nélkül, vagy pontatlanul, vagy sehogy sem képesek működni. Ez kiküszöbölhető az infrastruktúra független navigációval, mely során a jármű önmagától képes, pozíciójának megállapítására, külső antennák, és szatellitek nélkül. Ennek gazdasági, és fejlesztési előnyei egyaránt adottak, ellenben kivitelezése még sok mérnök fejét izgatja, nem kiforrott.

Dolgozatom során igyekszem részmegoldással szolgálni, a fent említett jelvesztés problémájára, és a piacon kapható BNO055 inerciamérő chippel eredményes méréseket végezni, majd navigációt megvalósítani, mely pontos orientáció visszajelzésre képes egy járművön. A chip alapvető működését és pontosságát ipari robotok segítségével kívánom validálni, majd integrálni egy központi rendszerbe.

Fontos ezen mérés adatait egy olyan keretrendszer számára átadni, mely majd képes lesz komplex működést koordinálni egy jármű esetén, a járművekre jellemző biztonsági előírásokat figyelembe véve, valamint kialakítva egy olyan belső program struktúrát, mely hosszútávon támogatottságot biztosít. Ennek megoldása a robotika világában már régebb óta jelen van, így helyes példákat alapul véve, kialakításra kerülhet egy élhető rendszer, mely kiszolgálja a következő évtizedeket.

Gyorsan fejlődő autóipari technológiák mellett elképzelhetetlen, hogy egy egész közlekedési járműpark lecserélhető lesz, rövid idő intervallumon belül, ezért fontos a modularitás és a visszafelé megvalósított kompatibilitás figyelembevétele, mely a jelenleg is használt kommunikációs megoldásokon alapszik. Ennek okán egy olyan rendszer kialakítását tartom szem előtt, mely márkafüggetlen megoldást valósítson meg és beépíthető lesz régebbi típusú autókba.

Abstract

Inertial measurement and positioning in the ROS

Nowadays, navigation systems of road vehicles have complex electronic and software background, which are based on GPS signals, and offline maps. The biggest problem with these systems, that they became inaccurate or stop working, if they lose GPS signal. This could be solved with the use of infrastructure independent navigation. Whit this method vehicles are capable of navigating and localizing on their own without any external aerial or satellite signal. This navigation method has not only economical but also development advantages, although its realization fascinates a lot of engineer in the field of electronics, and computer science.

During my thesis, I try to make successful tests with the BNO055 inertial measurement chip, then realize navigation which could give acceptable orientation feedback from the vehicle to solve the issue of navigation failures created by lost signals. To validate the basic measurements of the chip I will use industrial robots, then I would like to integrate the sensor in a centralized software system.

It is very important, to pass these data to a software framework, which will be able to coordinate complex operations in a real-world vehicle. This framework has to consider technical and safety requirements from the automotive industry and also create an inner program structure, which would be supported for a long time. The solution for this main framework is already is exists, but in the robotics industry, which could base our newly evolving system, to be sustainable and able to hold for decades.

With rapidly developing technologies in the automotive industry, it is impossible to imagine that our whole vehicle fleet could be replaced in a short-time period. For this reason, I believe that modularity and backwards compatibility is one of the key elements in this, which is based on communication and nowadays basically used technics. As a conclusion I keep in mind to create a navigation system, which could be added to older vehicles to make them intelligent, and to realize a brand independent system.

Tartalomjegyzék

1. Bevezetés 1
1.1 A fejlesztés célja 1
1.2 Shell Eco-marathon: Autonomous Urban Concept 1
2. A fejlesztés elméleti háttere 2
2.1 Navigáció a háromdimenziós térben 2
2.2 Inerciális navigációs rendszerek 2
2.2.1 A gyorsulásmérő 3
2.2.2 A giroszkóp 4
2.2.3 A magnetométer 5
2.2.4 Kvaterniók 5
2.2.5 Történelmi visszatekintés: Apollo 8 6
2.3 Robot Operációs Rendszer 8
2.4 A felhasznált szenzor: BNO055 10
2.5 „Önvezetés” és annak számítási egysége: NVIDIA Jetson TX2 12
2.6 CAN kommunikációs protokoll 14
3. Fejlesztési lépések 16
3.1 Háttér rendszer inicializálása 16
3.2 ROS - Kinetic Kame 20
3.3 Kommunikáció a szenzorral 21
3.4 Szenzor kalibráció és adatolvasás 21
3.5 Szenzorvalidációs mérések ABB IRB 140 ipari robottal 24
3.5.1 Validációs mérések eredményei 26
3.6 ROS – Node programozás 28
3.6.1 NANO_reader 28
3.6.2 model_odom 29
3.6.3 UNO_writer 30
3.7 ROS – Node-ok futtatása 31
4. Navigációs mérések 32
4.1 Vezérlés tesztelése 32
4.2 Szenzor adatok elemzése 34
4.2.1 Orientáció – Heading 35

4.2.2 Orientáció – Roll – Pitch.....36
4.2.3 ASSN - Számított pozíció.....37
5. Tesztelés egy közúti járműben.....39
5.1 Jelenlegi navigációs rendszer összehasonlítása.....40
5.2 CAN kommunikáció felépítése az autóval.....41
5.3 Kernel rebuild.....42
5.4 CAN interfész konfigurálás.....43
5.4.1 Rendszer szolgáltatás felállítása.....44
5.5 CAN hálózati tesztek.....45
6. Konklúzió.....46
7. További fejlesztési lehetőségek.....47
Köszönetnyilvánítás.....48
Irodalomjegyzék.....I
Ábrajegyzék.....IV
Mellékletek.....I

1. Bevezetés

1.1 A fejlesztés célja

Elsődleges célom, egy olyan navigációs eszköz rendszerillesztése, mely lehetővé teszi a pontos navigációt GPS[1] használata nélkül. Az önvezetés témakörében ez egy rendkívül fontos kérdés, mert a GPS jelek földalatti, vagy alagútban történő vétele a rálátás hiányából fakadóan lehetetlen. Ennek a problémának ma már vannak részleges megoldásai, mert más szenzorok, valamint offline térképek segítségével közelítő becslésekkel kiegészíthetők a navigációs információk, amíg éppen az alagútban haladunk. Ezen megoldások azonban nem tökéletesek, és késleltetéssel működnek csak. Ezen felül a legnagyobb ellenfél a mélygarázsok világa, ahol erre nincs opció a sok, és kis területen végzett manőver miatt, valamint a külvilágtól való teljes elzártságból adódóan. Feltételezve egy nagymértékben önvezetést támogató infrastruktúrát, és az „okosparkolást", a föld alatti navigációra precíz módon van szükségünk járművekbe integrálva, melyre egy inerciális navigációs rendszer megoldásként szolgálhat.

1.2 Shell Eco-marathon: Autonomous Urban Concept

A Shell Eco-marathon [1] története 1939-re nyúlik vissza az Egyesült Államokba, amikor a Shell kutatói fogadást kötöttek, hogy ki tud több utat megtenni egy gallon benzinnel. Akkoriban a legjobb eredmény 4,73 L / 100 km-re jött ki, ami ma már egy jól hangolt kis városi benzines autóval is kivitelezhető. A versenyt azóta is megrendezik, és ma már az egyetemi „kis" kutatók dolgoznak, fejlesztenek és építenek több kategóriában, hogy elnyerjék a legenergiahatékonyabb címet az évben. Ezen verseny sorozat egy új kategóriája indult el az idei évtől, melyben az önvezetés, és a hozzá kapcsolódó feladatok megvalósítása a cél, mint például: egy kör megtétele a versenypályán, parkolás előre felfestett helyre egy fallal szemben, vagy akadály kikerülés.

Rendkívül bonyolult és összetett rendszerekre, valamint nagy teljesítményű számítógépekre van szükség ennek kivitelezésére, ezért nem csak embert, hanem csapatot próbáló versenyszámról van szó. Ebben a kategóriában kíván indulni a Széchenyi István Egyetemen működő SZEnergy Team fejlesztő csapat is, ahol a navigációs rendszer alapja egy összetett inerciamérő chip lesz. Bízok abban, hogy ez nagyban segítheti, és sikerre vezetheti a csapatot, mert ezen szenzor további kiegészítése odometriával[2] és lézeres mérésekkel már precíziós helymeghatározást adhat az ismert térképen.

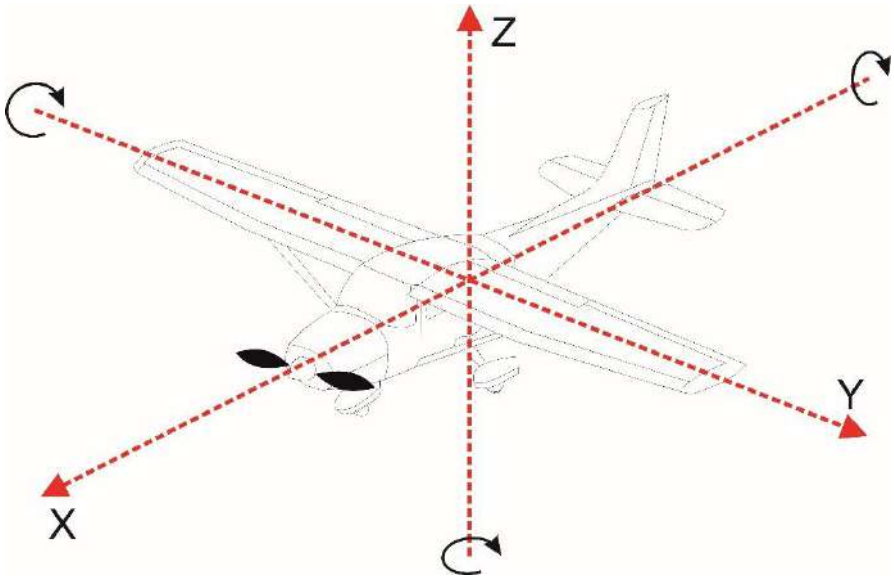
[1] Global Positioning System – Globális Helymeghatározó Rendszer
[2] kerék- vagy motorfordulat méréséből számított pozíció változás a kezdeti ponthoz képest

2. A fejlesztés elméleti háttere

2.1 Navigáció a háromdimenziós térben

Amikor navigációról beszélünk, akkor alapvetően egy fix koordináta-rendszert képzelünk el, melyben három jól ismert tengely - x - y - z helyezkedik el. Ezen tengelyek mentén az elmozdulás egyszerűen megadható és leolvasható. A repülésben és járműiparban [2] ezen a három tengelyen való mozgást az alábbi elnevezésekkel illetik, melyet az *1. ábra* mutat.

- Heading - Z-tengely menti elfordulás
- Roll - X tengely menti borulás
- Pitch - Y tengely menti billenés



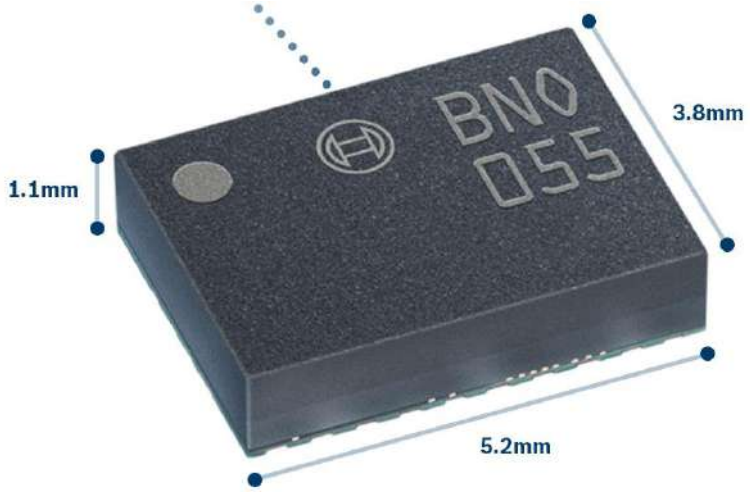
1. ábra: Navigációs tengelyek bemutatása.

2.2 Inerciális navigációs rendszerek

Inerciális navigációs rendszernek hívjuk azokat az útmutatást segítő rendszereket, melyek inercia, azaz tehetetlenség mérésen alapuló eszközök segítségével valósítanak meg. [3] Erre a célra manapság Micro Electromechanical System (MEMS), azaz mikro-elektromechanikus rendszereket használnak. Ezt elképzelni legkönnyebben egy mikrochipként lehet. Apró szilícium kristályon megvalósított integrált áramköri elemek ezek. Az inerciális navigációs rendszereket a rendszer szabadságfokával szokás jellemezni. A szabadságfok alatt azt értjük, hogy a különböző tengelyeken végzett mérések mennyire függenek egymástól. Annál pontosabb egy szenzorrendszer minél függetlenebb méréseket vagyunk képesek végezni a különböző tengelyeken.

Jelen esetben a dolgozatban használt BNO055-ös szenzor (*2. ábra*) minden mérést egyéni tengelyen végez, tehát a gyorsulásmérő, a giroszkóp és a magnetométer együttesen, egy 9 szabadságfokú szenzorcsomópontot valósít meg. Ezen felül hőmérséklet mérést is tud végezni, de a dolgozatom témája kapcsán erre nem térek ki. A több szabadságfokú mikro-elektromechanikus rendszerek belső felépítése rendkívül bonyolult. Nevükből eredően mikrométeres (10^{-6}) nagyságrendbe esnek belső felépítésük alkalmával használt vonalak. Ezen vonalak határozzák meg az elektronikai kommunikáció és áramfolyás útját. Nemcsak diszkrét elemek, mint kondenzátor, tekercs, és ellenállás találhatók meg ezekben az eszközökben, hanem logikai kapcsolások is.

Egy ilyen eszköz felhasználásával megvalósítható az úgynevezett infrastruktúra független navigáció, melynek nincs szüksége GPS/GNSS[3] jelekre, a lokalizációhoz. Ennek következtében a járművek sokkal rugalmasabbak és olcsóbbak lehetnek a jövőben, mert csak a fedélzeti eszközeikre és belső jelfeldolgozásukra fognak tudni támaszkodni a navigációt illetően, és nem pedig az éppen elérhető szatellitekre, melyek rálátása vagy elégséges, vagy éppen problémás.

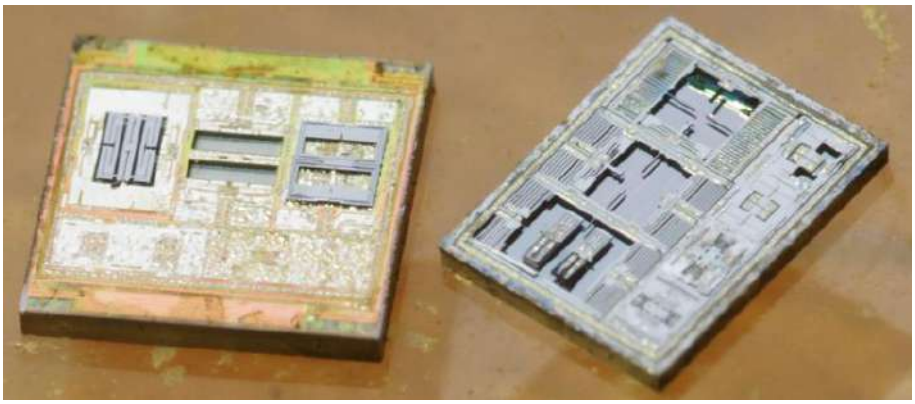


2. ábra. A Bosch BNO055 chip és annak fizikai méretei. [A1]

2.2.1 A gyorsulásmérő

A gyorsulásmérő felépítését tekintve egy rezgésbe hozható tömegből áll. A *3. ábra* alapján látható kialakításban a vékony levegőben lógó szálak gyorsulás hatására elhajlanak, ami ellenállás vagy kapacitás változást produkál a körülötte megvalósított kiértékelő kapcsolásban. Ennek mérésével határozható meg a pillanatnyi gyorsulás. Ennek időszerinti integrálja pedig megadja később a sebességet. A *3. ábra* mutatja egy három-tengelyes gyorsulásmérő MEMS kialakítását.

[3] Global Navigation Satellite System – Globális Navigációs Szatellit Rendszer



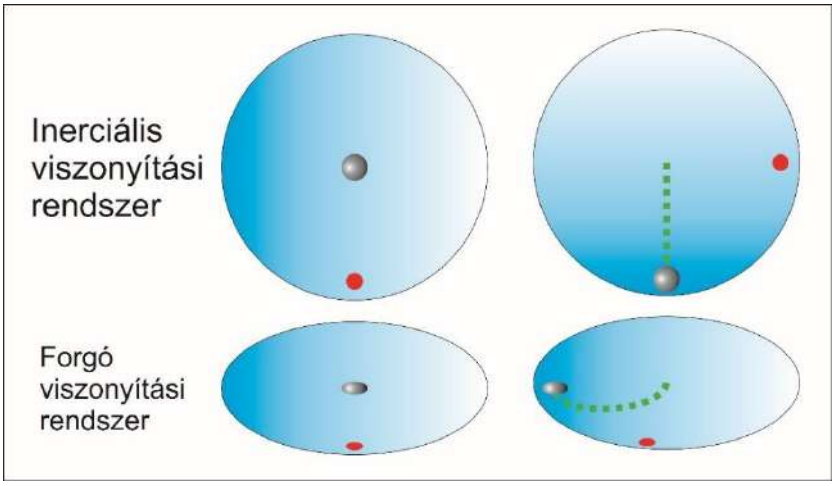
3. ábra. Egy összetett MEMS szenzor metszete. [A2]

2.2.2 A giroszkóp

A giroszkóp a perdületmegmaradás törvényét demonstrálja. A Newton-törvények egy test mozgását inerciarendszerben vizsgálják, melyről tudjuk, hogy nyugalomban van, vagy egyenes vonalú egyenletes mozgást végez. Amikor a Newton-törvényeket transzformáljuk egy egyenletesen, **ω** szögsebességgel forgó vonatkoztatási rendszerbe, megjelenik a Coriolis-erő [4, 5] és centrifugális erő. A Coriolis-erő egy ilyen forgó koordináta-rendszerben ható tehetetlenségi erő, melyet a *4. ábra* szemléltet. Ez abból adódik, hogy egy **v** sebességgel mozgó **m** tömegű testre plusz gyorsulás hat a forgás tengelyére merőleges mozgás esetén. Ezt az erőt az alábbi, 1. egyenletből számíthatjuk.

$$F_c = 2 \cdot m \cdot \mathbf{v} \cdot \boldsymbol{\omega} \tag{1}$$

Egy mikro-elektromechanikus szenzor esetén a giroszkóp megvalósítására [6, 7] egy rezgő rendszer van megvalósítva a szilícium lapkán, amelyre annak forgatása közben Coriolis-erő hat. Ebben az esetben is a hatást ellenállás vagy kapacitás változáson keresztül mérhetjük.



4. ábra. A Coriolis-hatás illusztrációja.

2.2.3 A magnetométer

A magnetométer a föld mágneses terét mérve segít az orientáció pontosabb meghatározásában. Ennek MEMS megvalósítását általában egy áramjárta kerettel kivitelezik. Ezen mérések is három tengelyen zajlanak, annak érdekében, hogy a chip bármilyen pozícióba kerül a mozgás során, képes legyen pontos irányultságot meghatározni a föld északi pontjához képest. A különböző szenzor típusok ezen értékek figyelembevételével számolják az abszolút vagy relatív orientációjukat. MEMS eszközöknél kulcsfontosságú a kalibráció, főleg a magnetométer kapcsán, hiszen az elektronikai eszközök elektromágneses kisugárzása is megzavarhatja annak működését. Ofszet eltolódás, mágneses tér torzulás, ortogonális hibák és zaj, ezek mind-mind negatívan ható effektek a működést tekintve [8]. A gyártó ajánlására fontos, hogy a kalibrációt olyan környezetben végezzük el, ahol később az üzemelni fog. Ezzel elkerülhető, hogy egy elektromágnesesen tiszta környezetben kalibrált szenzor, egy erősáramú környezetben hibás adatokat szolgáltasson majd. Természetesen figyelni kell a szenzor rendszer olyasfajta kialakítására, amiben minél kevesebb zavart gerjesztünk körülötte, de ettől eltekintve fontos a szenzort az adott környezetben bekalibrálni. A helyi hibalehetőségeket a kalibráció során az általam felhasznált chip szenzorfúziós algoritmusa figyelembe veszi, így az adott környezethez lesz illesztve a magnetométerem.

Az űrkutatásban is nagy jelentősége van a magnetométereknek [9], melyekkel az űrben jelenlévő mágneses tereket vizsgálják. Ilyen esetben nem csak egy módszert valósítanak meg, mint amelyet bemutattam (MEMS), hanem atomfizikai elven alapuló proton-processziós és elektrodinamikai elvet követő Flux-gate (fluxus-kapu) mérő egységeket szerelnek fel az űrsiklókra vagy egyéb a világűrben működő vizsgáló berendezésekre.

2.2.4 Kvaterniók

A kvaterniók [10] a háromdimenziós térrel állnak kapcsolatban, mégpedig a komplex számok[4] leképzéseként. Ha egységnyi hosszú komplex számokkal szorzást végzünk akkor a síkra vetítve egy forgatást valósítunk meg. Ezen elmélet kiterjesztése a háromdimenziós térre Sir William Rowan Hamilton tollából származik, miszerint a kvaterniók segítségével a forgatást térben is megvalósíthatjuk. A kvaterniók formális alakja: $a + bi + cj + dk$, halmaz jelölése $\mathbb{H}$, valamint igaz rájuk az alábbi összefüggés (*2.egyenlet*).

$$i^2 = j^2 = k^2 = i \cdot j \cdot k = -1 \qquad (2)$$

[4] Komplex szám: ($a + bi$) formális kifejezések, melyre teljesül az $i^2 = -1$ összefüggés. Halmaz jelölése: $\mathbb{C}$

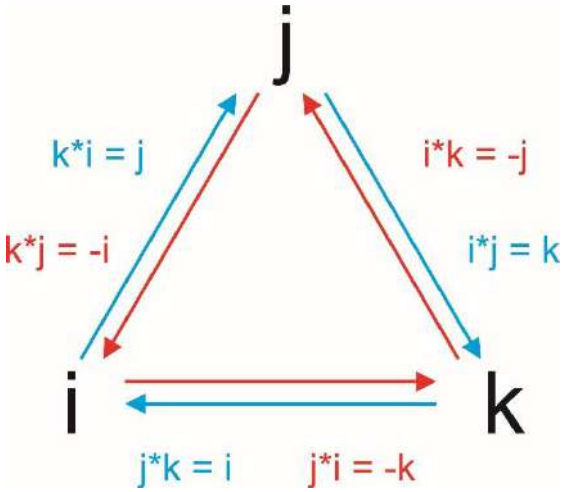
A *2. egyenletből* levezetve megadható az elemek önálló értéke, a másik két elem szorzataként, mely az alábbi egyenletek szerint alakul:

$$i \cdot j = k = -ki, \tag{3}$$

$$j \cdot k = i = -kj, \tag{4}$$

$$k \cdot i = j = -ik. \tag{5}$$

Az összefüggés egy önmagába visszatérő szabályt ad, melyet egy háromszög mentén ábrázolhatunk, ahol az óramutató járásával megegyezően szorozva két elem a harmadikat adja eredményül, óramutató járásával ellentétes irányban pedig a harmadik elem negatívját (*5. ábra*). Ezen elmélet segítségével a térbeli orientáció egy számnégyesként, vagy vektorként kezelhető, mely négy valós számmal adható meg (x,y,z,w).

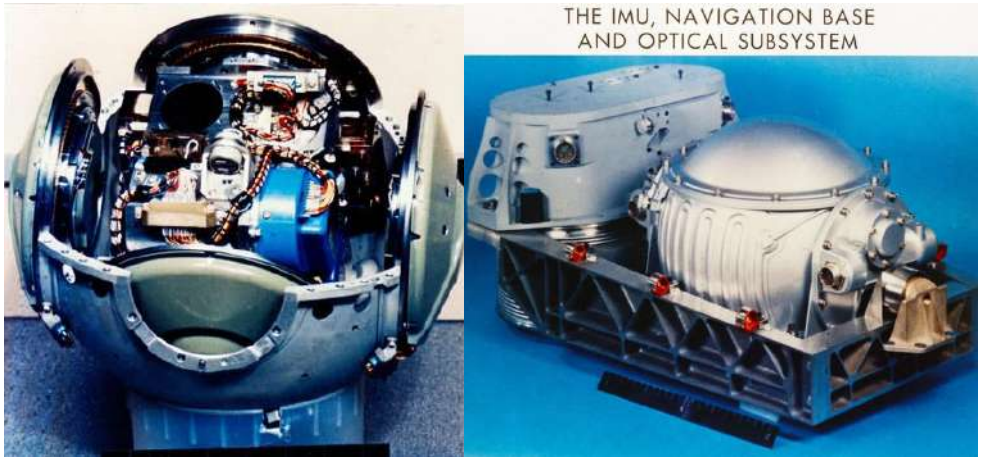


5. ábra: A kvaternió formális alakjának képzetes részeinek szorzási szabálya.

2.2.5 Történelmi visszatekintés: Apollo 8

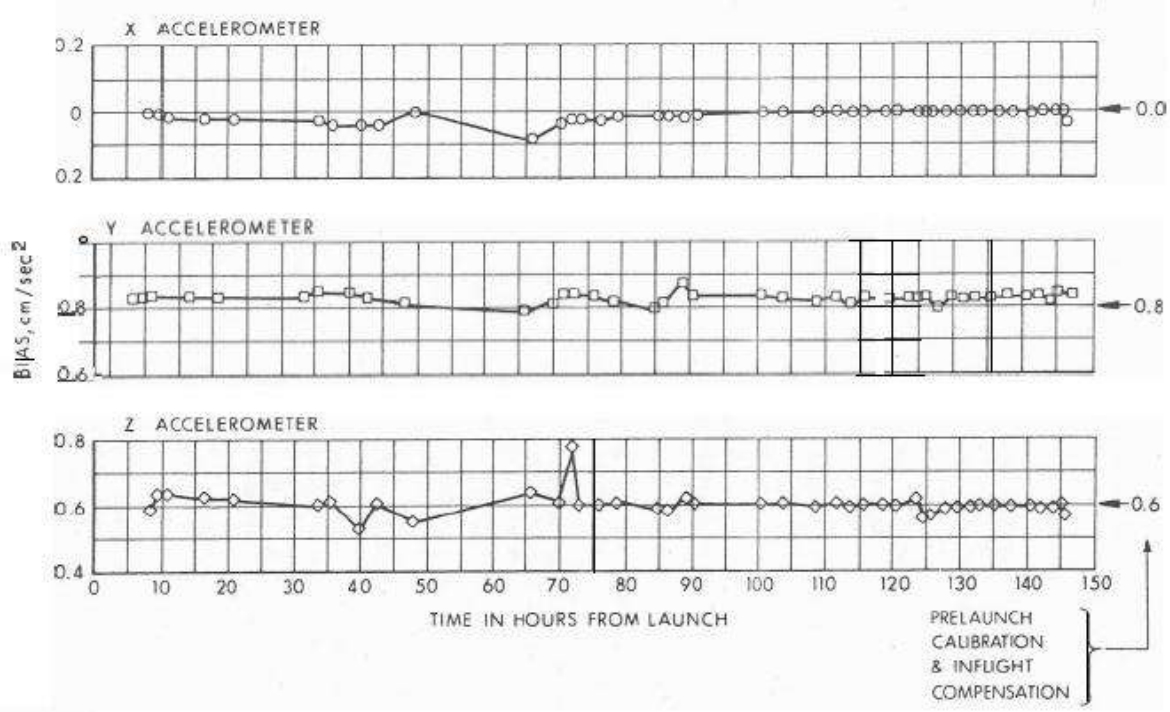
Az 1961-ben kezdődött Apollo projekt az „űrverseny” kiemelkedő fejlesztését indította el, amit a benne dolgozók ma is úgy emlegetnek, mint a legnagyobb előrelépés a számítástechnika és az elektronika történetében. A világűrbe feljutni önmagában nagy feladat, de ott manőverezni és hazatérni a Földre sokkal nehezebb munka. Habár a csillagok szerint lehet tájékozódni, de ezt egy szűk kapszulában rövid idő alatt precízen nem lehet megvalósítani. Ennek okán kezdődött meg az inerciamérőkre épített navigációs rendszer fejlesztése a NASA[5]-nál [11].

[5] National Aeronautics and Space Administration – Az USA kormányzati űrügynöksége



6. ábra. balra: Az Apollo 8 projekt során használt inerciamérő egység központi része. jobbra: A beszerelésre került komplett navigációs egység, az optikai stabilizálóval. [A3]

Az Apollo 8 repülésének idejére már annyira kiforrta magát ez a mérési összeállítás (*6. ábra*), hogy a jegyzőkönyvek szerint, pontosabb volt a szenzor, mint amire szükség lett volna a fellövések alkalmával. Ezt bizonyítják az adatnaplózás kiértékelései, ahol alább (*7. ábra*) látható, hogy a 147 órás repülés alatt a szenzorrendszer gyorsulásmérőinek legnagyobb ofszet hibája 0,02 cm/s^2 volt. A BNO055-ös szenzor esetében nincs lehetőség a cm/s^2-es mértékegység kiválasztására. Ennek oka maga a gyártástechnológia. A MEMS szenzorok ugyan nem képesek ilyen pontos mérésekre, de valójában erre nincs szükség a nagytömegű felhasználás körében, ellenben ezek a szenzorok rendkívül költséghatékonyan előállíthatók és nagy mennyiségben gyorsan gyárthatók.

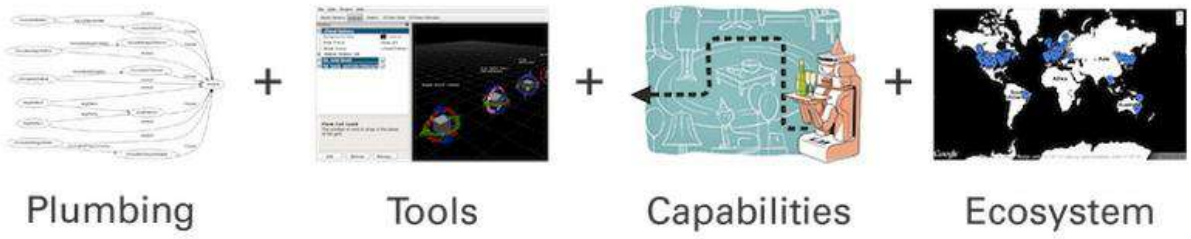


7. ábra. Apollo 8 repülési jegyzőkönyv részlet a gyorsulásmérők ofszet hibájáról. [A4]

2.3 Robot Operációs Rendszer

A Robot Operációs Rendszer - ROS[6] - egy meta-operációs rendszer, elsősorban robotokhoz fejlesztve [12]. Az ROS keretrendszere 2007 óta van jelen a robotika világában, és egyre növekvő tábora még jobban felgyorsítja a nyílt forráskódú rendszer terjedését. Felépítése nagyban hasonlít az Unix rendszerekhez, és részben ezért megköveteli egy Linux disztribúció alapkörnyezetét. A szolgáltatásai lefedik, amit egy operációs rendszertől elvárunk, de egy másik operációs rendszerre épül. Fontosnak tartom a projekt során egy olyan bázis rendszer kiválasztását, mely minél tovább támogatást élvez, ezért nem a legfrissebb verziót választottam az ROS rendszerek közül, hanem a 'Kinetic Kame' verziót, mely 2021-ig biztosít támogatást és ezzel garantálja a fejlesztések lehetőségét a további pár évben. Alább, az *1. táblázatban* látható a rendszer áttekintése és funkciói.

1. táblázat. Az ROS áttekintése.

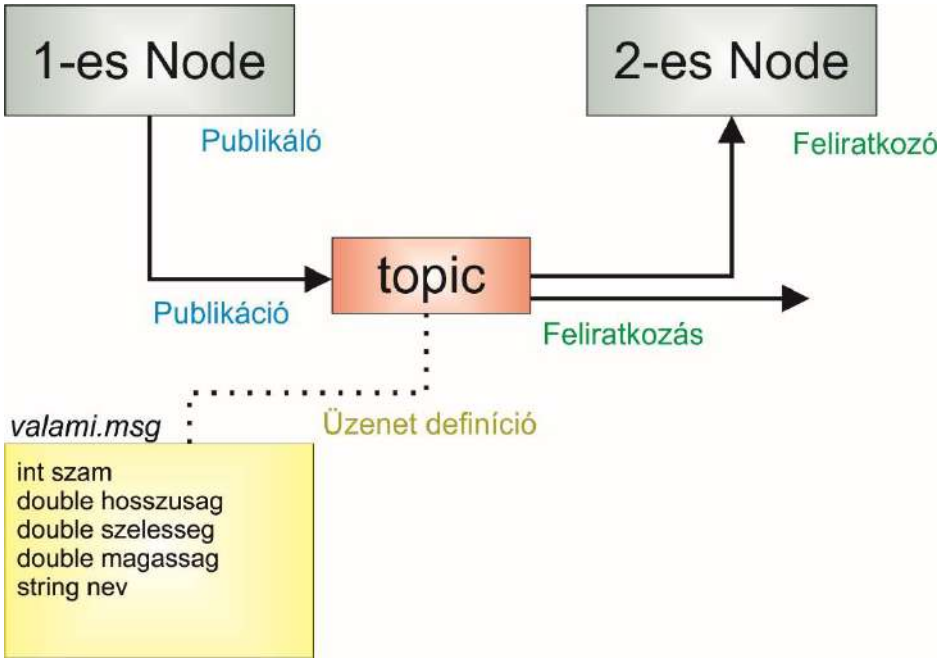


Folyamat menedzsment	Szimuláció	Szabályzás	Csomag rendszer
Kommunikáció a folyamatok közt	Vizualizáció	Tervezés	Szoftver disztribúció
Eszköz driver-ek	GUI	Érzékelés - Megfigyelés	Dokumentáció
	Naplózás	Térképépítés	Példák - Bemutatók
		Adat kezelés	

Az ROS biztosítja számomra az alsó szintű driver kezelést, közvetett rálátást a hardverekre, általánosan használt függvények implementálását, csomag menedzsmentet és nem utolsó sorban a folyamatok közti üzenet közlést. Az ROS futásának folyamatábrája egy Peer-to-Peer[7] (P2P) hálózathoz hasonlítható, ahol nincs kitüntetett szerver, mindenki egyenrangú fél. Így jelennek meg a rendszerben a folyamatok, akár több számítógép között egységesen. A kommunikáció során szinkron és aszinkron adatkapcsolat kezelésére képes. A rendszer a Linuxhoz hasonlóan csomagokból épül fel és ez adja a rendszer egyik legjobb tulajdonságát, mert ettől modulárissá válik. A jelenlegi projektben

[6] Robot Operating System
[7] Közvetlenül összekötött végpontokkal kiépített hálózat, ahol nincs központi kitüntetett csomópont

ez a tulajdonság a fejlesztés szempontjából elsődleges. Az ROS a *publish-subscribe* kommunikációs modellt alkalmazza, melynek szemléletes bemutatása (*8. ábra*) alább látható. A programok valójában *node*-okként jelennek meg és *topic*-okon keresztül zajlik az üzenetközlés, melyekre bárki feliratkozhat (*subscribe*) illetve azokra bárki publikálhat (*publish*). Minden üzenetnek, amit kiadnak a keretrendszeren belül meg kell felelnie valamilyen üzenet definíciónak. Ez biztosítja a rendszeren belüli szabványos adatfeldolgozást, és könnyű integrálhatóságot.



8. ábra: A publish-subscribe kommunikációs modell magyarázata.

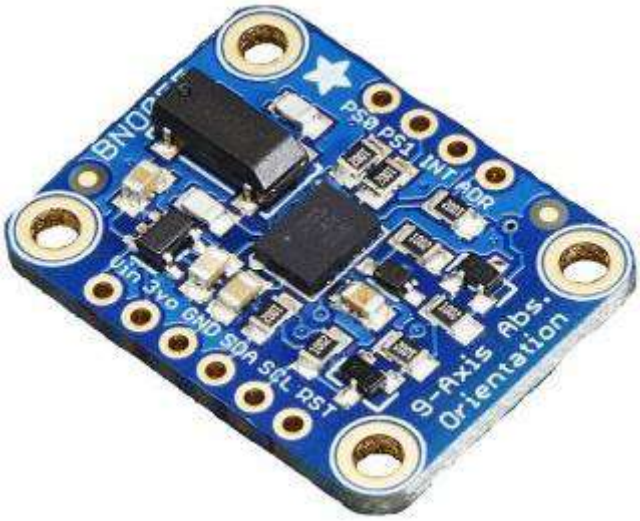
Az ROS fejlesztése forráskódokban történik, majd egy összerendezett csomagként fordításra kerül. A keretrendszer nagy előnye, hogy nem csak egy programozási nyelv használata engedélyezett. A megfelelő modulok használatával a csomagunk egyes részeit írhatjuk különböző nyelveken, a végen egy egészként összeáll majd, amiről a fordító gondoskodik. Akár C, akár C++, vagy akár Python nyelven is fejleszthetünk, ezeket mind tudja kezelni a fordító, ezzel elősegítve több programozói csapat munkáját, ami a projektek gyors előrehaladását segíti. Egy összetett robot működtetéséhez sokszor kedvezőbb egy-egy funkciót más nyelven megvalósítani, ahol esetleg a nyelvezetből adódó előnyök megkönnyítik azon funkciók implementálását. Ez lehet akár hardver kezeléssel kapcsolatos funkció, vagy egy sebesség kritikus részegység, esetlegesen egy webszerver szolgáltatás vagy driver elérhetőségi kérdés. Fontos kiemelni, hogy esetemben a sebesség kritikus működés miatt, amit később indokolok, a C++ nyelv kerül előtérbe, a valós versenyjárműre integrálható programkódok kapcsán.

2.4 A felhasznált szenzor: BNO055

A fenti típusnév egy Bosch által gyártott okos-szenzort, Applikáció Specifikus Szenzor Csomópontot takar – továbbiakban ASSN[8]–, mely képes IIC[9] (vagy I^2C) és UART kommunikációra [13]. Ez az eszköz egy 28 lábú System in Package (SiP), ami tartalmaz:

- háromtengelyes 14 bites gyorsulásmérőt, (m/s^2)
- háromtengelyes 16 bites giroszkópot, (deg/s)
- háromtengelyes magnetométert, (μT)
- 32 bites cortex M0+ mikroprocesszor[10] a Bosch Sensortec fúziós szoftverrel.

A két éve piacra került ASSN különleges tulajdonsága, hogy nem csak gyors belső adatfeldolgozással rendelkezik, hanem a hozzá írt szenzorfúziós szoftver elvégez olyan adatfeldolgozási feladatokat, melyre a felső szintű rendszerben a kiolvasást követően már nem kell számítási kapacitást pazarolni. A chip egy Adafruit által gyártott lapkán (*9. ábra*) érkezett, mellyel I^2C kommunikációs protokollon keresztül kommunikáltam a feladatok elvégzéséhez.

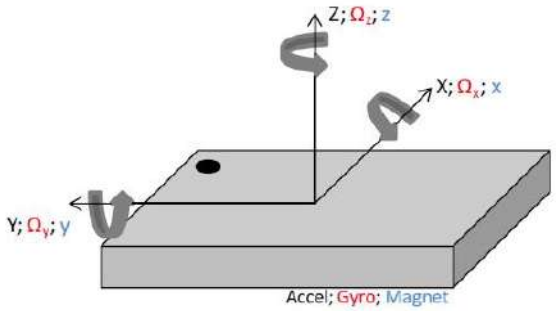


9. ábra. Az Adafruit BNO055-ös lapkája. [A5]

A BNO055 chip esetén a mértékegységek programozhatók, melyre egy külön regiszter szolgál. Ennek a regiszternek az alapbeállítása megegyezik a fentebb említett mértékegységekkel, és az orientációs értékek alapbeállítása is a fok, így nem írtam felül ezen regisztert.

[8] ASSN: Application Specific Sensor Nodes
[9] Inter-Integrated Circuit: Szinkron kommunikációs forma, melyet a Philips Semiconductor fejlesztett ki 1982-ben. Kétvezetékes kommunikáció: SDA és SCL megnevezéssel, ahol SDA-adatkapcsolat SCL–szinkron órajel számára fenntartott vonal. IC-k, processzorok, és mikrokontrollerek köztes kommunikációjára alkalmazzák kifejezetten rövid távolságokon.
[10] Cortex M0+ mikroprocesszor: az ARM processzor gyártó legenergiahatékonyabb mikroprocesszora (4μW/MHz)

A szenzor a beépítés variálhatóságát elősegítve tartalmaz egy külön regisztert, mely a tengelyek konfigurálására szolgál. (*10. ábra*) A mérésekhez én a gyári értékeket használtam, de így az X-tengelyen utólag az értékek inverzét kellett képezni, így ennek a regiszternek a felülírása mindenképp fontos lesz a későbbiekben.



10. ábra: A szenzor tengelyeinek gyári beállítása. [A6]

Az ASSN legnagyobb előnye korábbi versenytársaival szemben, hogy szenzorfúziót alkalmaz már a tokozáson belül. Külső adatfeldolgozás esetén ez a folyamat sokkal több időt és számítási kapacitást emésztene fel, ezenfelül a mérési pontosság romlását eredményezné. Szoftverében több üzemmód választható, melyek lehetőséget biztosítanak az önálló szenzorok használatára és fúziós használatra is. Én az NDOF (Nine Degrees of Freedom) funkciót választottam, mely kilenc szabadságfokos vizsgálatokra ad lehetőséget, valamint csak itt működik a beépített szenzorfúziós algoritmus. Ennek a funkciónak a választása egyelőre a chiphez tartozó header[11] fájlban kerül definiálásra. Ezt az egyszerűség kedvéért alakítottam ki így, mert a szenzorfúziós eljárás alkalmazása és tesztelése az egyik cél. Abban az esetben, ha más módban kívánok méréseket végezni, akkor regiszter szinten kell programozni a chipet. Az első beüzemelések alkalmával ezen regiszterek írásával váltottam módot, de összességében sokkal több műveletet igényel, és a fejlesztés praktikusságát előtérbe helyezve ezt a gyorsabb megoldást választottam. A chip által mért jellemzők:

- Abszolút orientáció 3 tengelyen (Heading-Roll-Pitch) 100Hz-es frissítésre képes
- Szögsebesség/Gyorsulás 3 tengelyen 100Hz-es frissítésre képes
- Mágneses térerősség 3 tengelyen 20Hz-es frissítésre képes

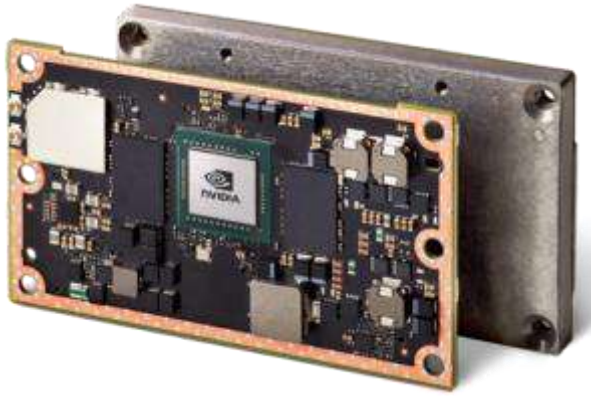
A szenzorfúzió [14] hatására az eszköz robusztus kimenetet ad és nagy kimeneti adatsebesség érhető el, mely fontos alapja egy valós idejű rendszernek, mert minél több információ áll rendelkezésre egy másodpercen belül, annál jobban szűrhető egy-egy hibás mérés. Bízom abban, hogy ez a szenzor biztosíthatja majd a pontos helyzetmeghatározás eszközét beltérben és kültéren egyaránt.

[11] olyan forrás állomány melyben előre definiált függvények, és memória címek szerepelnek

2.5 „Önvezetés” és annak számítási egysége: NVIDIA Jetson TX2

Az autonóm célra fejlesztett járművek körében az egyik legfontosabb paraméter a számítási kapacitás. Az egyszerűnek tűnő, emberek milliói által megtanult műveletek és a KRESZ szabályainak betartása, nem tűnik olyan bonyolultnak, de rendkívül megterhelő az agyunk számára. Olyan gyors reakciókat és adatfeldolgozást igényel, amely számítógépek terén eddig óriási méretet igényelt. Gyakran látható volt, hogy egy autó csomagtartóját teljesen beépítették, mindenféle elektronikus eszközzel annak érdekében, hogy egyszerűbb feladatokat el tudjanak végezni a mérnökök. Ezen funkciók nagyon sokáig csak lezárt, üres pályákon működtek, ahol nem voltak más szereplői a közlekedésnek. Ez abból fakad, hogy ami az ember számára triviális, és könnyen észrevehető, már ismert tudása alapján köthető valamilyen korábbi vagy más témájú ismeretéhez, arra könnyedén reagál, míg egy számítógépnek ekkor nehéz reagálnia. A könnyed reagálás még most sem igazán mondható el a fejlesztés alatt álló rendszerekre, gépekre. Az ember képes párhuzamot vonni az élet különböző területeivel, felmérni aktuális helyzetét, és elemezni azt, majd döntést hozni, egy másodperc alatt. Ezen feladat számítógépek körében, rettentő sokrétű programozást igényel, hisz minden olyan forgalmi szituációra fel kell készíteni az autót, ami az utakon, a mindennapi közlekedésben előfordulhat. Ez rettentő sok előre definiált opció, mely a változatos környezettel és időjárási viszontagságokkal kombinálva óriási méretű lehetőség halmazt ad. Egy gép számára ezekből helyesen választani rendkívül bonyolult. Most persze felmerülhet a kérdés, *„Az ember nem hozhat rossz döntést?”* De hozhat, és sajnos ezekből sokszor származik baleset. Ezek viszont egyedi esetek, mely következtében a személy felelősségre vonható. Egy gép esetében erre nincs lehetőség, és nem megengedhető, hogy egy hibás program okán a közlekedés részvevőit veszélybe sodorja egy-egy fejlesztés. Ebből fakad a tény, hogy a jogi tisztázatlanságok mai napig visszafogják a fejlesztések nagy részét ezen a téren, mert szociológiai, és társadalmi gátak vannak az emberekben. Az „önvezetés” akkor lesz majd valódi önvezetésnek mondható, hogyha kijelenthető lesz a rendszerről, hogy az addig nem ismert forgalmi szituációkban is úgy tud beavatkozni, hogy nem sodorja veszélybe sem utasait, sem a forgalom többi résztvevőjét annak érdekében, hogy reagáljon az aktuális forgalmi helyzetre. Ez a reagálási képesség pedig egészen addig alaptétel lesz, míg lesznek az úton olyan személy- és vagy tehergépjárművek, illetve kerékpárások, és gyalogosok, akik nem programozott módon közlekednek. Ez belátható, hogy sosem fog bekövetkezni, míg az ember gyalogolni képes, ezért sok infrastrukturális fejlesztés már abba az irányba mutat, hogy elszeparálható legyen két közlekedési forma, programozott vagy sem. Ha képesek leszünk elkülöníteni ezen közlekedési formákat, úgy már a programozott szakaszokon belül fejleszthető és kivitelezhető lesz a biztonságos valós önvezetés.

A döntéshozás, és tanulás problémájára a neurális hálózatok adhatják a megoldást, melyek sok-sok minta alapján megtanulhatják azokat a szituációkat, melyeket fel kell, hogy ismerjenek a közúti forgalomban, és ezáltal biztosíthatják a korrekt reakciót. Ezen algoritmusok alapjául szolgálhatnak egy tanulásra képes rendszernek, melyet mesterséges intelligenciának szokás nevezni. Ezen algoritmusok rendkívüli számítási igénnyel rendelkeznek. Itt kerül képbe a modern GPU[12]-k szerepe, melyek feldolgozóképessége megfelel ezen új, óriási igénybevételnek. A mai napig élen járó grafikus feldolgozó egységeket gyártó Nvidia két éve tört be az autóipari piacra, és ma már első számú beszállító, azon autógyártók számára, akik önvezető funkciókat kívánnak implementálni autóikba. Ezen felül nagyon sok fejlesztő központ, és cég is foglalkozik eszközeikkel, ahogy az egyetemen működő versenycsapat is, annak érdekében, hogy kihasználva a nagy számítási kapacitását önműködő járművet alkothassanak. Erre a feladatra a jelenleg kereskedelemben kapható Nvidia Jetson TX2-es fejlesztő környezet biztosít lehetőséget [15]. Az önálló lapkát a *11. ábra* mutatja, illetve adatlapja a *1. számú mellékletben* található.



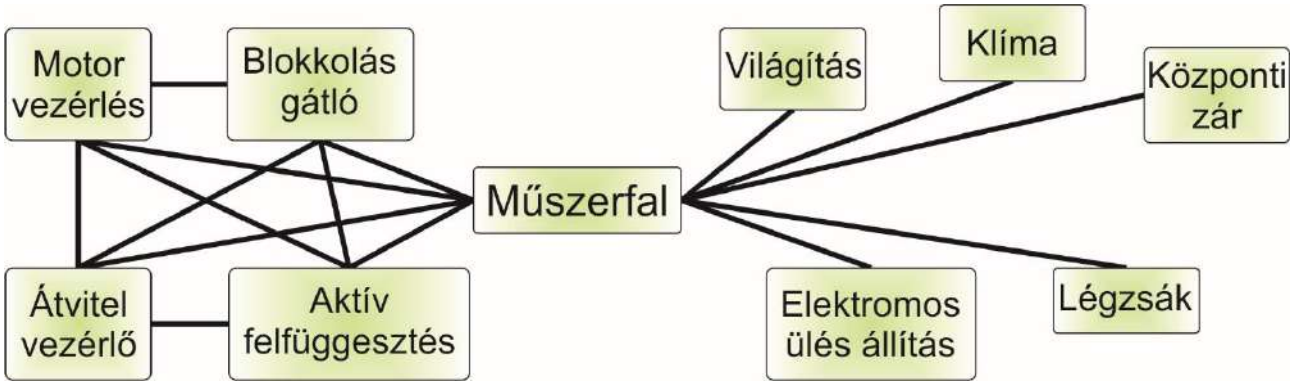
11. ábra: A Jetson TX2 lapka és annak hőközlő eleme. [A7]

A diszkrét feladatmegoldást és egyes számítások elvégzését a számítástechnika világában a számítási kapacitással szokás jellemezni, a közérthetőség kedvéért mondhatjuk úgy, hogy ez tört számokkal végzett matematikai műveletek összessége. A szakirodalomban ennek értékét flops[13]-ban adják meg, azaz az egy másodperc alatt elvégezhető lebegőpontos műveletek számával jellemzik [16]. Ez a Jetson TX2 esetében eléri az 1,5 terraflops-ot, ami megfelel egy mai felsőkategóriás Intel processzor (i9 7960X) teljesítményének. Fogyasztás tekintetében viszont az Nvidia Pascal architektúrájának köszönhetően csak 15W-ra van szüksége, ami az Intel processzorának 11-ed része. Fontos megjegyeznem, hogy egy CPU[14] és egy GPU került összehasonlításra, mely nem teljesen reális, de jól szemlélteti a számítási képességeket és a szükséges teljesítményt.

[12] GPU: Graphical Processing Unit — grafikus feldolgozó egység
[13] flops: Floating-point operations per second — másodperc alatt elvégzett lebegőpontos műveletek száma
[14] CPU: Central Processing Unit — központi feldolgozó egység

2.6 CAN kommunikációs protokoll

A CAN (Controller Area Network) [17, 18] nem más, mint egy soros kommunikációs busz rendszer, melyet a Bosch kifejezetten járműipari felhasználásra fejlesztett ki, és mutatott be először 1986-ban a detroiti SAE[15] kongresszuson. Jelentőségét alátámasztja, hogy az elkövetkező években sok nagyobb autógyártó átvette a rendszert, majd 1993-ban szabványosították, és ma már nincs olyan autógyártó, aki, nem ezt használná. Manapság az ISO-11898-as szabványszám alatt találhatjuk meg a rendszer leírását, melyet azóta is évről-évre csiszolnak. A fentebb nevezett busz rendszer kialakítására azért volt szükség, mert ahogy a közúti járművekbe egyre több elektronikus segéd berendezés és kényelmi funkció került, ezek közös működtetése nagyon bonyolultabbá vált elektronikai szempontból. Minden eszközt össze kellett kábelezni a többi olyan eszközzel, melyhez információt szerettek volna eljuttatni, és így például a műszerfal mögött szörnyű kábelrengeteg volt. A körül írt rendszer, egy elméleti megvalósítását a *12. ábra* illusztrálja.

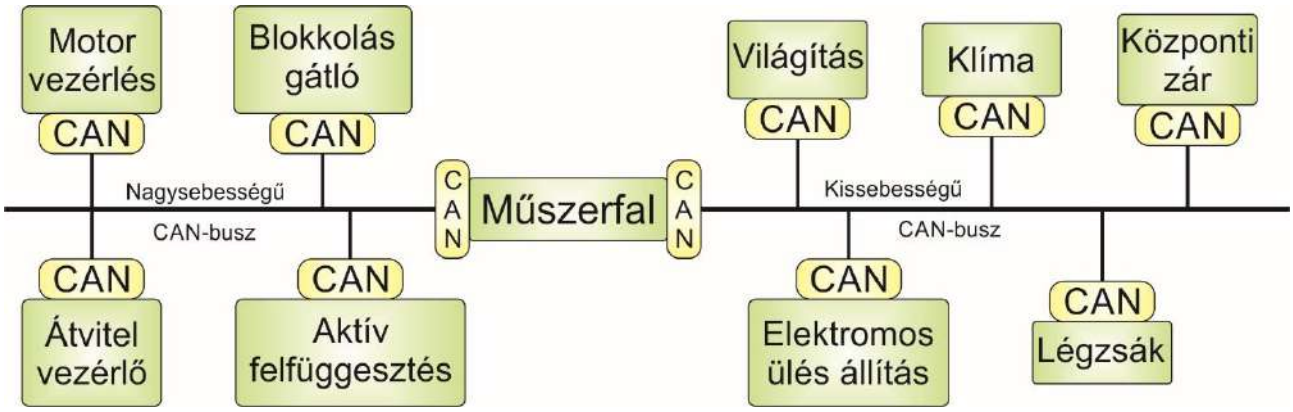


12. ábra: Kommunikációs megoldás a CAN-busz előtt.

A sok kisebb eszköz különböző kommunikációs megoldásokat használt, valamint voltak sebesség különbségek a rendszerek között ezáltal rendkívül bonyolulttá vált ezek összehangolása. A Bosch idejében felismerve a problémát kifejlesztett egy olyan buszrendszert, mely nagy terhelhetőségű, olcsón megvalósítható és rendkívül biztonságos. Ennek egyik alapja, hogy a vezérlő chip-től kapott TTL[16] információkat az adóvevő differenciális jelekre alakítja. Csavart érpáron történik a kommunikáció, melyet adóvevő chip-ek illesztenek a kábelre, ennek köszönhetően akár elektromágneses zavarások közepette is zavartűrő rendszer építhető ki. Az adóvevők ezen felül azért is felelősek, hogy huzalozott „és” kapcsolat épüljön ki.

[15] Autóipari Mérnökök Egyesülete: Az USA-ban, 1905-ben alapított társaság, mely a közlekedés standardizálásával, és fejlesztésével foglalkozik. ~138 ezer taggal rendelkezik.
[16] Tranzisztor-Tranzisztor Logika: Kommunikációs jelkialakítási elv, melyben az első a logikáért a második az erősítésért felelős. Egy adó, egy vevő, és egy földelési pont szükséges hozzá gyakorlati megvalósításban. (RX-TX-GND)

A rendszer az úgynevezett „Több-mester” elvet alkalmazza, ami azt jelenti, hogy nincs kitüntetett busz mester, mindenki egyenrangú. Ennek köszönhetően, ha egyes elemek megsérülnek, esetleg nem tudnak kommunikálni, ugyan a rendszer teljesítő képessége csökken, de nem omlik össze. A rendszer elméleti felépítését az alábbi *13. ábra* szemlélteti. Ma már ipari körülmények között is gyakorta használt protokoll, kedvező tulajdonságai miatt. A mai középszerű autókban – az elektronika bonyolultságát szemléltetve – akár három különböző ilyen busz rendszer is működhet párhuzamosan.



13. ábra: A CAN-busz kommunikációs megoldása.

A SZEnergy Team versenyautója is ezt a belső kommunikációs formát használja, ezért a fejlesztés szempontjából kulcsfontosságú szerepe van a CAN hálózatoknak. Az önvezető funkciókkal kapcsolatosan bármit is szeretnék megvalósítani, a rendszeremnek képesnek kell lennie együtt működni a mostani vezérlőegységgel. Ez a vezérlő egység egy National Instruments myRIO típusú eszköz. Az autóban működő jelenlegi eszközök CAN kommunikációs protokollal cserélnek adatot egymás közt, és az önvezetés kapcsán tervezett számítási egységet – a Jetsont – is ezzel kell tudni összekötni. Ennek segítségével biztosítható a gyors szenzorolvasás, és a referencia parancsok átadása, mind a motor, mind a kormányszervó szabályozásának esetében, amellett, hogy nem kell megbontani vagy megszakítani semmilyen, már kész és megvalósított kommunikációs vonalat. A jelenlegi versenyjármű CAN hálózata 30%-ban van kihasználva körülbelül 70 paraméter állandó monitorozása mellett, így biztos lehetek abban, hogy a felmerülő plusz referencia értékek, és szenzorolvasási műveletek nem fogják túlterhelni a kommunikációs buszt. A két eszköz összekötése a korábban ismertetett busz hálózat jellemzőiből adódóan egyszerűen kivitelezhető, minimális költségekkel jár majd. Ugyan ebből az okból használják, más fejlesztő csapatok is a Jetsont, mert hamar felfűzhető egy a közúti közlekedésben megtalálható személyautó létező CAN hálózatára és elkezdődhetnek a tesztek.

3. Fejlesztési lépések

3.1 Háttér rendszer inicializálása

A rendszer hátterét egy modelljármű adja, mely 1:6 méretarányos mása a SZEnergy Team egyetemi csapat versenyautójának. Kinematikailag teljesen homológ modell, fedélzetén egy Raspberry Pi3 mikroszámítógéppel (*14. ábra*), melyet 2016 februárja óta találhatunk meg a piacon. Ez a kártyaszámítógép képes Linux disztribúciók futtatására, ezáltal éles tesztkörnyezetet biztosíthat a fejlesztés során számomra. A kártyaszámítógép elnevezés abból adódik, hogy a méreteket tekintve a Raspberry Pi3 alapterülete csak egy pár milliméterrel haladja meg egy normál bankkártya méretét (85.60 mm × 56.5 mm × 17mm). A mikroszámítógép 64 bites architektúrával rendelkezik, és 1 GB RAM szolgál memóriaként a programok futtatásakor. Ennek kiterjesztése a Linux segítségével könnyedén elvégezhető egy USB Pendrive formájában, így a nagyobb forráskódok és csomagok fordítása sem jelent majd problémát a memória méretét illetően. A lapkaszámítógép fedélzetén egy 1,2 GHz-es négy magos processzor dolgozik, mely SD kártyáról olvassa be az operációs rendszert.

14. ábra: Raspberry Pi3 mikroszámítógép. [A8]

A projekt célja egy moduláris rendszer felépítése szoftveres és hardveres szempontból egyaránt, amely a modelljárművön és a fejlesztés alatt álló új versenyautón egyaránt működő képes lehet.

A moduláris rendszer felépítés és a projekt alapvető kritériumai:

- Real-Time közeli működés:

  Annak érdekében, hogy gyors reakciója legyen a később összeépítésre kerülő járművünknek, és értelmezni tudja a másodpercek alatt történő változásokat, szükséges a valós idejű feldolgozó képesség.

- Megbízható operáció:

  Nem megengedhető, hogy egy önvezető autó, bármely funkciójának megvalósítása során, olyan állapotba kerüljön, melyben megragad, és nem képes folytatni a küldetését.

- Magas fokú rendszerbiztonság:

  Fontos, hogy a rendszeren belül működő részegységeink, csak azokhoz a perifériákhoz és rendszer eszközökhöz férjenek hozzá, amelyekkel szorosan együttműködnek.

- Minimális memória igény:

  Az óriási számítási kapacitás és a Real-Time működés kapcsán fontos kritérium, egy olyan operációs rendszer kiválasztása, mely nagyon kevés memóriát használ fel a rendszermemóriából, saját működésének fenntartására.

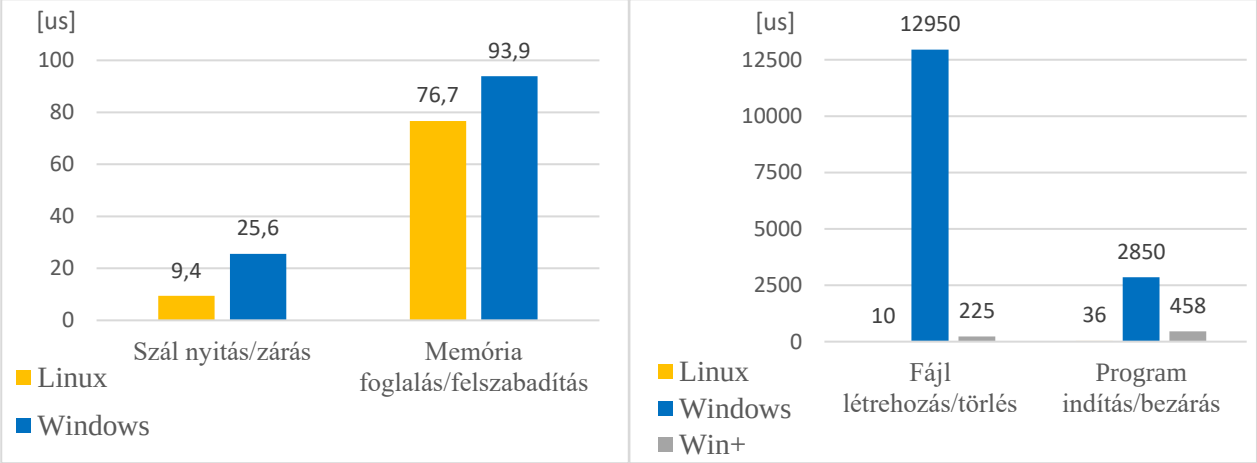
- Nyílt forráskódú:

  A teljes fejlesztést nyílt forrásokból, és minél alacsonyabb költségekkel kívánjuk megvalósítani. Ennek érdekében, ahol lehetőségünk van, nem használunk olyan szoftvereket, melyeket meg kell vásárolni.

Ezen fenti paramétereknek megfelelően az Ubuntu Linux 16.04 Linux disztribúció lett kiválasztva szoftveres alapként, mely 2021-ig hosszútávon nyújt támogatást a ráfejlesztett rendszerünk számára. Egy Windows-hoz képest, már a rendszer követelmények **50%**-kal kevesebb memóriát írnak elő egy Linux számára a rendszer működtetéséhez. Ez annak köszönhető, hogy a Linux alatt működő grafikus megjelenítő jóval egyszerűbb és kevesebb felhasználói effekttel dolgozik, valamint a rendszer mögött működő dinamikus könyvtárakkal lehetőség nyílik arra, hogy több program egyszerre férjen hozzá megosztott könyvtárakhoz, ezzel kiküszöbölve azt, hogy minden program betöltse a teljes saját könyvtárát használat közben [19]. Ezt alátámasztja a két operációs rendszert összehasonlító teszt, mely alapvető műveleteket vizsgált a két rendszerrel kapcsolatosan, azonos hardver kialakítás mellett. Windows 10 Pro és Ubuntu 16.04 verziókkal [20].

Az említett rendszerek közti teszt eredményei alább láthatók. *(15. ábra)* A mérések során az alábbi műveletek kerültek elvégzésre:

- 100 szál létrehozása majd megállítása és bezárása
- Program elindítása, mely egy üres függvényt rejt, majd bezárás
- 65000 fájl létrehozása egy adott mappában. (fájl méret: 32 byte)
- 1 millió memória foglalás, majd felszabadítás (4-128 byte méretben)



15. ábra: Alapvető rendszer feladatok elvégzésének ideje összehasonlítva.

A lefuttatott programok összehasonlításából látható, hogy a szálnyitások esetén **58,02%**-kal gyorsabb a Linux, míg a memória foglalások alkalmával **22,43%**-kal igényel kevesebb időt a feladat elvégzése. A tesztek során kiderült, hogy a Windows beépített víruskeresője a Windows Defender és az indexelés rendkívül visszafogja, és lelassítja a rendszert, olyannyira, hogy fájlok létrehozása és törlése esetén a Windows rendszer ezerszeresen elmaradt. Az indexelés és a Windows Defender kikapcsolását követően -mely a grafikon *Win+* adatsoraiként látható- óriási javulás látható, mert így már csak **20-szoros** lemaradása van a Linuxhoz képest, de beláthatjuk, hogy ilyen esetben a Windows védtelen marad. Program indítás és bezárás esetén az eltérés már csak **12-szeres**, szintén a Linux javára, de itt is kikapcsolt védelem mellett történt a mérés. Összességében elmondható, hogy az általam is kiválasztott Linux operációs rendszer legalább **25%**-kal gyorsabb művelet végzést tesz majd lehetővé, ha belekalkulálom a grafikus rendszer eltávolításával járó további erőforrás felszabadulást.

A Linux környezetében további előny, hogy a rendszerfrissítések teljes mértékben a felhasználók által kontrollálhatók, a frissítések elvégzésének időpontja, valamint a frissítések forrása egyaránt. Alapbeállítás, hogy az operációs rendszerre hatást gyakorló változtatásokat nem lehet végrehajtani külső forrásból származó programokkal. Ennek felülírása olyan engedélyeket és ismereteket követel meg, melyet a külső program nem tudhat. Csak akkor futhat le egy ilyen program,

ha egy a belső rendszert ismerő felhasználó, vagy a rendszergazda erre engedélyt ad a megfelelő rendszerváltozókban. A rendszeren belüli adat- és eszközbiztonság magasfokú, mert nagyon szigorú jogosultsági protokollokkal épül fel a rendszer. Többek közt az egyes hardver eszközök és portok elérésének jogai is definiálhatók. A fejlesztett szoftverek jogosultságai egyenként is könnyen meghatározhatók, így kialakítható egy magas fokú biztonsággal rendelkező rendszer, ahol például a szenzor olvasó csomópontok semmiképp sem kaphatnak írási, vagy futtatási jogosultságot az egyes portok felett.

A választott operációs rendszer LTS (Long Term Support) verzió, mely azt jelenti, hogy hosszútávú támogatással rendelkezik az operációs rendszert fejlesztő csapat részéről. A rendszer egyik előnye, hogy a fejlesztési időszak alatt grafikus rendszerként működtethetjük azt, majd, amikor beépítésre kerül a versenyautóba, akkor ezt könnyen eltávolíthatjuk, ezzel nagy mennyiségű tárhelyet, memóriát és számítási kapacitást szabadítunk fel a rendszerünkben. Telepítése gyorsan és egyszerűen kivitelezhető, nem bonyolultabb, mint egy már korábban ismert Windows telepítése. Egyedül a partícionálás igényel odafigyelést, mert egy Windows rendszertől eltérően, itt biztonsági szempontokat figyelembe véve, több partíciót, többféle fájlrendszerrel érdemes létrehozni, melyek specifikus feladatok számára biztosítanak tárhelyet. Ilyen például a SWAP partíció, mely a rendszerről leválasztott gyors elérésű rész, amit memória kiterjesztés céljából használ a Linux. A felsorolt, gyökér könyvtárban található mappák elkülönítése az alábbi séma szerint ajánlott, melynek megvalósítása logikai partíciókezeléssel kivitelezhető.

- /boot/ boot mechanizmus helye (min20MB)
- /opt/ külső gyártói program fájlok (4GB+)
- /usr/ Linux programfájlok helye (4GB+)
- /var/ log fájlok, Xen virtuális gép fájlok (3GB+)
- /srv/ WEB és FTP szolgáltatás fájlok (szolgáltatás függő)
- /home/ személyes felhasználói fájlok (minimalizálni)
- /tmp/ átmeneti fájlok (min 1GB)

A Linux operációs rendszer parancssori működése rendkívül felhasználóbarát, és sok esetben gyorsabb művelet végrehajtást tesz lehetővé, mint a grafikus felületen elvégezve. A fejlesztés közben sokat dolgoztam parancssorban, így biztos vagyok benne, hogy ez az éles fedélzeti üzem alatt, később sem fog problémát okozni.

3.2 ROS - Kinetic Kame

A rendszer telepítése nagyon gyorsan elvégezhető a projektben használt Linux disztribúciók alatt, legyen szó akár az Ubuntu, vagy az Ubuntu Mate verziójáról. Utóbbi a korábban kiválasztott Ubuntu 16.04 LTS verziójának kisebb számítási teljesítményű eszközökre fejlesztett verziója, amely kifejezetten ARM processzorokra készült, mint amilyet a Raspberry Pi3 mikroszámítógép használ. Ennek lényegi eltérése a grafikus megjelenítés területére terjed ki, ezzel tehermentesítve a processzort. A ROS telepítése [21] az alábbi műveletekkel elvégezhető, és ezzel a mikroszámítógép készen áll a lefejlesztett csomagok használatára.

1. Elfogadtatjuk a rendszerrel a külső forrásból érkező szoftverek telepítését:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -sc) m
ain" > /etc/apt/sources.list.d/ros-latest.list'
```

2. Megadjuk telepítéshez szükséges biztonsági kulcsot a rendszer számára:

```
$ sudo apt-key adv --keyserver hkp://ha.pool.sks-keyservers.net:80 --recv-key 4
21C365BD9FF1F717815A3895523BAEEB01FA116
```

3. Frissítjük a rendszert, hogy a telepítés előfeltételei biztosak legyenek:

```
$ sudo apt-get update
```

4. Telepítjük a kiválasztott ROS verziót:

```
$ sudo apt-get install ros-kinetic-desktop
```

5. A „rosdep” inicializálása, annak érdekében, hogy a későbbi függőségek könnyedén telepíthetők legyenek, illetve ,ha valamilyen forrás fordítása egyes ROS komponenseket igényelne:

```
$ sudo rosdep init
$ rosdep update
```

6. Környezeti változók hozzáadása az alapvető shell[17] beállításokhoz:

```
$ echo "source /opt/ros/kinetic/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
```

A parancsok lefutását követően a mikroszámítógép készen áll az ROS futtatására, melyről meggyőződhetünk, ha annak magját, a mester node-ot elindítjuk.

[17] Unix rendszerhéj: parancssori értelmező, mely összeköti a rendszer magját -a kernelt-, a felhasználóval

3.3 Kommunikáció a szenzorral

Az első tervek szerint a szenzort közvetlenül a mikroszámítógéppel kötöttem volna össze és I^2C kommunikációs protokollt használtam volna, de sajnos ez problémába ütközött, miáltal a Raspberry Pi3 nem támogatja ClockStretching-et. [22] Ez a szoftveres mechanizmus akkor szükségeltetik I^2C kommunikáció esetén, mikor a buszon kommunikáló szolga - esetemben a szenzor - lassabb kommunikációt igényel a mester által biztosítottnál. Ennek következtében egy mikrokontrollert kellett közbe iktatnom, ami megvalósítja az I^2C kommunikációt, majd sorosan továbbítja azt egy USB porton keresztül. A feladat elvégzése a tesztkörnyezet szűkét tekintve (modellautó alváza) valamilyen kis helyigényű fejlesztő környezetet kívánt meg. Erre a célra az Arduino Nano-t találtam a legalkalmasabbnak, mert a kívánt kommunikációs protokollt ismeri, már korábban is dolgoztam vele, és kielégíti a szenzor és a köztes adatfeldolgozás igényeit is. Ezen egy ATmega328-as mikrokontroller található, architektúráját tekintve ez egy 8 bites vezérlő, amely 16 MHz órajelen működik. Az összeköttetést követően, melyet az alábbi *2. táblázatnak* megfelelően létesítettem, elérhetővé vált a chip a hexadecimálisan megjelenített 28-as busz címen, és elkezdhettem a programozást.

2. táblázat: A szenzor és az Arduino Nano csatlakozó kiosztása

Adafruit BNO055		Arduino Nano	
SCL	Pin 12	Analog 5	Pin A5
SDA	Pin 11	Analog 4	Pin A4
VCC	Pin 20	5V DC	Pin 5V
Ground	Pin 19	Ground	Pin GND

3.4 Szenzor kalibráció és adatolvasás

A mikrokontroller kódját a hozzá kapcsolódó fejlesztőkörnyezetben (Arduino IDE[18]) C nyelven írtam, ami egyből lehetőséget nyújt a programok feltöltésére USB-n keresztül. Az arduino és az adatruit által létrehozott programkönyvtárakból dolgozva létrehoztam egy olyan szoftvert, ami fedélzeti üzemre készült, és mind kalibrációt, mind adatolvasást megvalósít.

[18] Integrated Development Environment – integrált fejlesztő környezet

A program először inicializálja a soros kommunikációt, majd elkészíti a naplófájlt, és annak fejlécét a későbbi egyszerű értelmezéshez. A szenzor információk olvasásakor, ha azonos ID-val szólítjuk meg a szenzort, mint amit korábban bekalibráltunk, akkor az EEPROM[19]-ból beolvasott kalibrációs ofszet mátrix használatával elkezdődik a mérés, és adatfeldolgozás. Amennyiben nincs az EEPROM-ban ofszet mátrix tárolva, vagy nem egyezik a szenzor ID, akkor egy önkalibráló ciklusba lépünk. A kalibráció működése közben a program 4 regiszter értékét figyeli. A szoftver segítséget nyújt, mert egy három lépcsőből álló skálán visszajelzést látunk a kalibráció aktuális állapotáról, rendszer és szenzorelem szinten is. Amennyiben minden szenzorelem (gyorsulásmérő, giroszkóp, magnetométer) és a rendszer is eléri a kalibrálás hármas -stabil- szintjét, abban az esetben a program visszaküldi az ofszet értékeket tartalmazó vektort majd az ofszet vektor beírásra kerül az EEPROM-ba, és végül elkezdődik a mérés. A megvalósított program a *2. számú mellékletben* megtalálható. A kód struktúráját tekintve szorosan kötődik az ROS keretrendszer „Imu” üzenet típusához (3. *számú melléklet*). Ez az üzenet típus definiálja, milyen adatoknak kell szerepelnie benne, és a mikrokontrolleres kiolvasást aszerint építettem fel, hogy mire lesz szükség. Ennek érdekében az első négy adat az orientációhoz szükséges kvaterniókat tartalmazza. Ezeket követik a giroszkóp három tengelyen végzett méréseinek adatai, majd végül a három tengelyen mért lineáris gyorsulás adatai. Mindegyik adatot egy szóköz választ el a következőtől, így a kiolvasás alkalmával könnyen darabolható a soronként küldött adathalmaz.

A kalibráció megadott műveletekhez kötött, melyet a szenzor gyártója pontosan megad. Ezeket érdemes a felhasználási környezetben végezni, annak érdekében, hogy a fedélzeten is pontos értékeket kapjunk majd. A szenzorelemek különböző kalibrálási módszereket kívánnak meg. A dinamikus módszereket kézben végezhetjük, vagy lengőkábeles kialakítással már a fedélzetre építve. A gyorsulásmérők kalibrálásához le kell helyezni a szenzort egy vízszintes síkra, míg a giroszkópok beállításához mozgatásra van szükség. Ezen mozgatás közben a szenzort rá kell helyezni egy valós vagy képzeletbeli kocka mind a hat oldalára, ezzel megforgatva minden tengelye körül. A magnetométer helyes ofszet értékének beállításához a szenzort egy nyolcas alakban kell mozgatni. Fontos megjegyezni, hogy az általam használt üzemmódban a kalibráció során írható 23 ofszet értékre nincs ráhatásom a működés közben, azt a szenzorfúziós algoritmus chipen belül módosítja saját egyenletei szerint. Ettől eltekintve a kezdeti állapotot megadhatom, és meg is kell adnom, annak érdekében, hogy minden indulás egységes legyen orientáció meghatározás szempontjából. amihez egyszer mindenképp szükséges a kalibráció elvégzése. Ennek eredménye és az EEPROM-ba beírt ofszet értékek láthatók a következő sorokban.

[19] a memóriachip egy speciális változata, mely elektromos úton újraírható.

```
$ Calibration Results:
$ Accelerometer: 65532 22 65505
$ Gyro: 65535 1 0
$ Mag: 65273 148 65448
$ Accel Radius: 1000
$ Mag Radius: 648
```

A soros kommunikációt 115200-as baudrate[20] mellett valósítottam meg, annak érdekében, hogy a lebegőpontos értékek átvitele később se ütközhessen kommunikációs problémába, ha a jelenlegi 40Hz-es adatfrissítésről 60 vagy 80 Hz-re emelném azt. Az adatok feldolgozása a mikrokontrolleren minimális, egy típuskonverziót követően minden adat készen áll a soros továbbításra. A BNO055-ről olvasott szenzoradatokon kívül még a motor enkóderének olvasását is itt valósítottam meg, mert egyszerű matematikai műveletekkel elvégezhetők a számítások, és így már nincs szükség későbbi feldolgozásra, ezzel időt és energiát spórolva a felsőbb rendszeren. Az enkóder számítása az alábbiak szerint történik:

- Fordulatszám kiszámítása a 3. egyenlet szerint:

$$rpm = pulse\_num \cdot \frac{60}{24} \tag{6}$$

ahol *pulse_num*, a megszakításkezeléssel számolt impulzusok számát tartalmazza, a *60* a perc alapú számításhoz szükséges, míg a *24* az egy csatornára vonatkozó impulzus szám fordulatonként.

- Szögsebesség kiszámítása a 4. egyenlet szerint:

$$velo = \frac{rpm}{9{,}5492965964254} \tag{7}$$

ahol az *rpm* a 3. egyenletből számolt fordulatszám, míg a 9,5492965964254 a fordulatszám és a rad/sec közötti váltószám.

- Megtett távolság kiszámítása az 5.egyenlet szerint:

$$dist = \frac{sum\_pulse\_num}{1185{,}36 \cdot 0{,}282743} \tag{8}$$

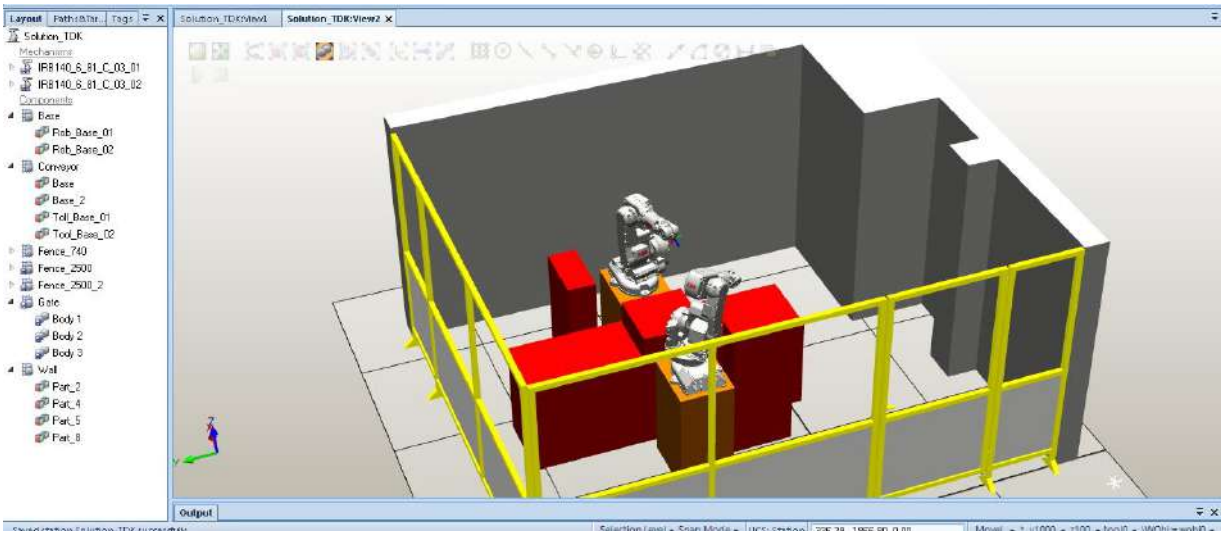
ahol *sum_pulse_num* az eddigi impulzus számok összege, *1185*,36 az áttételhez számított egy csatornára vetített lefutó élek száma egy fordulat alatt, míg *0,282743* a modelljárművön használt Pololu 90D kerék kerülete.

[20] jelarány mértéke, egy adott átviteli média esetén, hogy hány modulált jelet továbbít másodpercenként

3.5 Szenzorvalidációs mérések ABB IRB 140 ipari robottal

A validációs mérések elvégzéséhez olyan eszközt kellett választani, mely elég stabil, és nagy pontosságú, valamint biztosítja a mérések ismételhetőségét. [23] Ezen felül az is fontos szempont volt, hogy képes legyen nem csupán összetett mozgásokat megvalósítani, hanem egyetlen tengely mentén rotációs és transzlációs mozgást is végezni. Ennek megvalósítására az automatizálási tanszék robot laboratóriumában volt lehetőségem, ahol kettő ABB - IRB 140-6/0.8 típusú ipari robot[21] állt rendelkezésemre [24]. Ezek hat tengelyes többcélú ipari robotok. A robotokkal történő mérés további előnye, hogy dinamikus validációra ad lehetőséget, és biztosítja az inerciamérő eszközöm belső szenzorjainak többtengelyes vizsgálatát egy teszt elvégzése alatt. Ennek jelentősége, hogy így az elvégzett manuális kalibráció ellenőrzése gyorsan kivitelezhetővé válik, és ipari minőséget biztosít a későbbi orientációs mérések számára, mely kulcsfontosságú a navigációban.

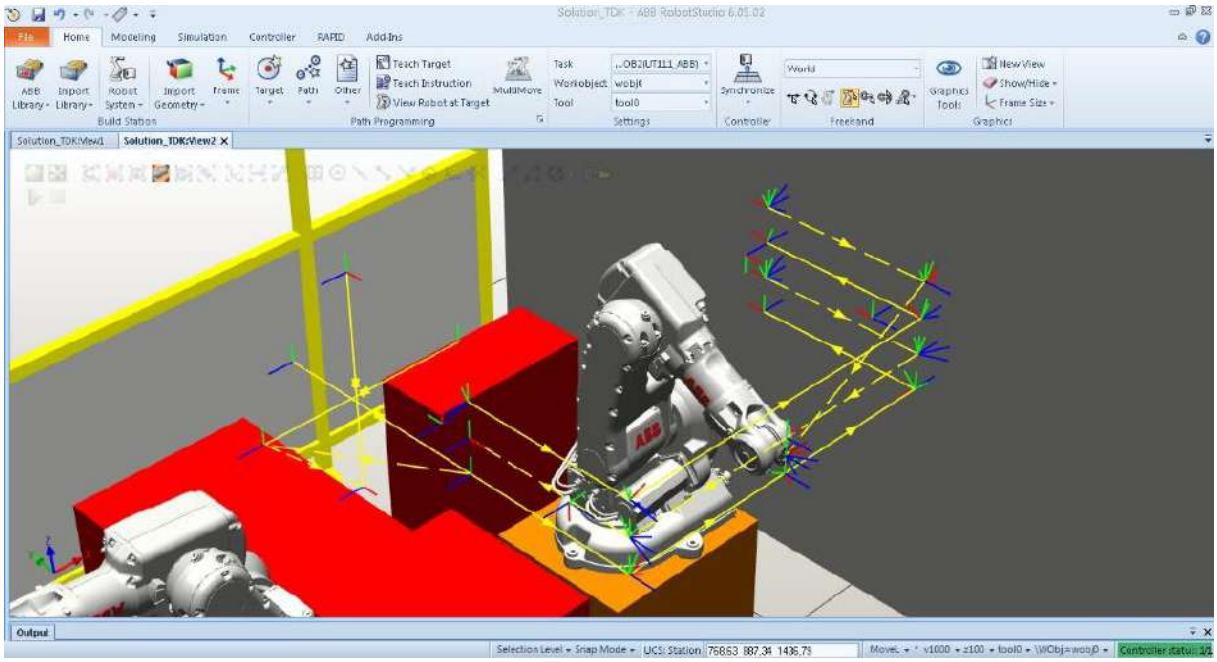
A bejárandó pálya és az orientációs forgatások megszerkesztésére a RobotStudio-t használtam, mely az ABB által biztosított szimulációs program saját robotjaikhoz, robotcelláikhoz. [22] A program lehetőséget biztosít az ütközésvizsgálatra, így a robottér megszerkesztése után biztos lehettem abban, hogy nem okozok kárt a robotcellában a mérések elvégzése során, mert azt a szimulációban előzetesen lefuttattam. Az alábbi *16. ábra* a robot laboratórium robotcelláját ábrázolja a grafikus szoftverben megvalósítva.



16. ábra. *A validációhoz használt robotcella szimulációs tere.*

[21] ipari robot: VDI 2860-as irányelv (1981) szerint: *„Az ipari robot univerzálisan állítható többtengelyű mozgó automata, melynek mozgás-egymásutánisága (utak és szögek) szabadon – mechanikus beavatkozás nélkül – programozható és adott esetben szenzorral vezetett, megfogóval, szerszámmal vagy más gyártó eszközzel felszerelhető, anyagkezelési és technológiai feladatra felhasználható.”*

A mérésekhez csak a 02-es jelzésű ABB robotot használtam. A mérést több pályára osztva zajlott, ahol külön vizsgáltam a BNO szenzor egyes tengelyeit. Az első három szinten a kitüntetett pontokban egytengelyes orientációs méréseket végeztem, majd a negyedik szinten komplex mozgásokat is megvalósítottam. A *17. ábra* mutatja az általam programozott robotpályát, melyben jól elkülöníthető a négy szint. A robot pálya szerkesztése során figyelmet fordítottam arra, hogy minden mérendő tengelyt megmozgassak és átfogó képet kapjak a szenzor képességeiről. Az ipari robot programkódja korábbi TDK dolgozatomban megtalálható [22].



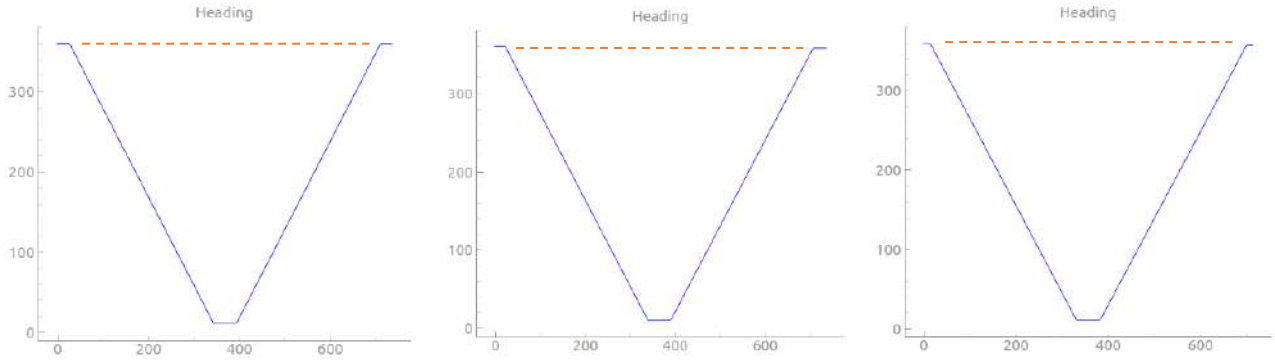
17. ábra. Szimulációs utak a RobotStudio-ban.

A roboton futó program a szimuláció alapján készült el RAPID nyelven [25] melyet részben a szoftver generált, részben magam szerkesztettem. Ennek futtatása a robothoz kapcsolt IRC5-ös vezérlőn keresztül történt. Első alkalommal csak ’Manual’ módban lett lefuttatva a teljes program annak érdekében, hogy a robot lassú mozgással végig követhető legyen. Ebben a módban ugyanis a programozott sebesség csak 10%-a megengedett biztonsági okokból. Miután megbizonyosodtam a program lefutásának tökéletességéről, már elindítható volt a mérés 100%-os sebességgel is. Az elvégzett méréseket valós idejű grafikai megjelenítés segítette, melyet python nyelven készítettem, és a mérendő tengelyek aktuálisan mért értékeit mutatták 40 Hz-es frissítés mellett.

Sikerült olyan adatnaplózást megvalósítani, mely számosított adatbázist hozott létre a mikrokontroller időbélyegeivel és a szenzorértékekkel együtt, de ezenfelül az élő grafikus visszajelzés is lehetőséget adott a mérések értékelésére.

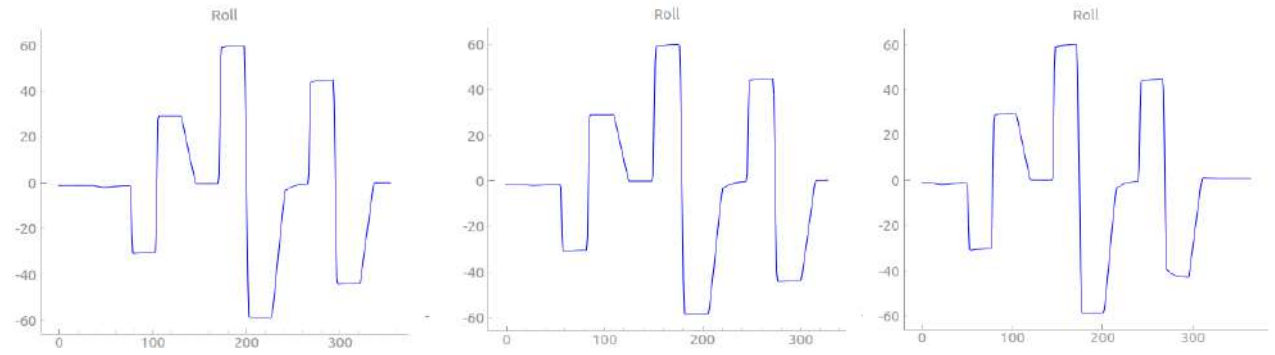
3.5.1 Validációs mérések eredményei

A z-tengely körüli elfordulást a robotkar J1-es tengelyének egyenletes forgatásával tudtam mérni, amit az IRC5-ös vezérlő kézi moduljával tudtam megtenni. [22] Egy teljes körbefordulást követően visszafelé is elvégzésre került a művelet, mely az alábbi grafikonokat (*18. ábra*) eredményezte, és átlagos eltérésre **1,6 °** volt, ami a 360°-os tartományra **0,44%** hibát jelent. Alább jól látható a változás linearitása, ami elvárt volt a robot egyenletes mozgásából fakadóan és az oda-vissza forgatás visszatérésének pontossága.



18. ábra: "Heading" értékek vizsgálatának eredményei.

Az y-tengely körüli elmozdulást a „Roll" értékének vizsgálata mutatta meg, melyben a programozott robotkar pozitív, és negatív irányban egyaránt ugyanakkora mértékben megbillentette a szenzort először 30° majd 60° végül 45° nagyságban. A mérések során **1,19 °** differencia volt tapasztalható. A *19. ábra* jól mutatja az egymást követő változások értékeit a bázisponthoz képest, és azok ellentétes irányú, de azonos amplitúdóval megjelenő görbéit.

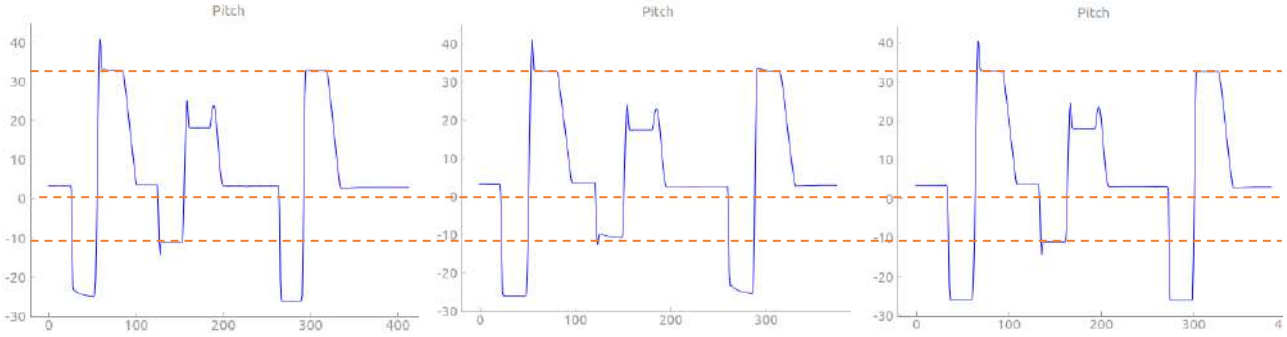


19. ábra: "Roll" értékek vizsgálatának eredményei.

Ebben a mérésben a teljes tartomány ± 180°-ig terjed, mert megkülönböztetett oldalak vannak, pozitív és negatív előjellel. Ennek okán a tartományra vetített hiba **0,661%**.

Az x-tengely mérései során a robotot nem lehetett minden kitüntetett pontban úgy programozni, hogy egy tengelyének mozgásával elérje a kívánt pozíciót, vagy ne kerüljön szingularitási problémába. Ennek magyarázata, hogy a robot vezérlő programja matematikai formulákkal, és mátrix műveletekkel definiált, és képes a „végtelen” kezelésére, míg a valós fizikai környezet nem. Például elképzelhető olyan, kinematikai csukló konfiguráció, melyben a konstans mozgás folytatásához, egy csuklónak végtelen gyorsan át kellene fordulni a másik oldalra. Amíg ez a szimuláció és a matematika során elvégezhető, addig a motorvezérlők nem tudják kezelni az ilyen parancsokat, és a robot program leáll. Ez a probléma merült fel a x-tengelyes mérések során is, így módosítanom kellett a programot, melynek következtében, voltak pontok melyeket a robot csak a mérni kívánt szögérték túllépésével tudott elérni. Alább látható, hogy a *20. ábra* grafikonjai ezt a túllövést tökéletesen visszaadják.

A grafikonokról leolvasható, hogy nullponteltolódási hiba volt tapasztalható [22] a mérés során, ettől eltekintve viszont pontos a szögértékek tartása és a bázisponthoz képest szimmetrikus a grafikon. Egy újabb kalibrációval az nullponteltolódási hiba orvosolható lesz.



20. ábra: "Pitch" értékek vizsgálatának eredményei.

A mérések alapján a szenzor képes lesz a későbbi feladatok pontos kiszolgálására, mert mérési pontatlansága **1%**-on belülre esik. A jelenlegi mért értékeken kívül a szenzor képességeiből adódóan a giroszkóp, és gyorsulási adatokat felhasználva tovább pontosítható lesz a navigációs rendszer. Ennek fuzionális megvalósítása, valamint további szűrők beiktatása az ROS keretrendszeren belül lesz elvégezve. Itt további külsős adatokkal pontosításra kerül majd a navigáció, mint például az odometria és lézerszkenner adatok. Ezen adatokon felül kameraképek feldolgozásából számított pozíció is beillesztésre kerül majd. A kameraképek feldolgozása részben már működőképes, ezt egy konvolúciós neurális hálózat végzi.

3.6 ROS – Node programozás

3.6.1 NANO_reader

Az általam készített „olvasó” ROS csomópont – NANO_reader – elsődleges elvárása az, hogy feldolgozza a soros adatokat a szenzor és a mikrokontroller felől, majd azokat strukturált üzenet típusokba rendezze, és elérhetővé tegye a rendszer többi résztvevőjének számára. Ez a kiszolgálás fontos minden olyan bemeneti szenzor *node* számára, ami meg kívánja osztani információit a rendszeren belül. A feladat megvalósítását egy egyszerű „Publish” csomópont kiegészítéseként készítettem el az inerciamérő szenzor és az odometria számára. Az szenzor adatok feldolgozásáért felelős *node*hoz tartozó programkód a *4. számú mellékletben* található, míg az odometria számítása és a csomópont leírása később kerül említésre. A *NANO_reader* elején kiemelendő, hogy importálásra kerül az *Imu.h és a JointState.h* header (*5.számú melléklet*), melyek tartalmazzák az üzenet definíciókat, és azok felépítését. Importálásra kerül továbbá a soros kommunikációhoz elengedhetetlen *SerialPort.h* header is, ami a soros adattovábbításhoz kapcsolódó függvényeket és definíciókat tartalmazza. A csomópont kódjának elején a publikáció leírása található, melyben az *topic* elnevezések és funkciójuk kerül definiálásra, valamint, hogy milyen üzenet típusokat fognak felhasználni. Meghatározok egy 50 Hz-es elvárt frekvenciát, ami megadja, hogy a csomópontnak ilyen időközökkel kell lefuttatni a feladatát. Példányosítom a két használt üzenet típust, majd belépek egy soros olvasási próba ciklusba. Ez azért fontos, mert ennek segítségével kezelhetők a soros olvasási, vagy port nyitási problémák, anélkül, hogy a *node* összeomoljon. Amennyiben sikeres a port nyitás, elkezdődik a beolvasás. Egy sorból, az összes elem egy vektorba kerül, melyet feldarabolok a szóközök mentén, ezáltal elkülönítem a fogadott értékeket. A vektor egyes elemeire hivatkozva, akárcsak egy mátrix esetén, végig iterálható az eredeti vektor, és minden változónak egyesével értékül adhatom a típuskonverzión átesett beolvasott adatot. Ezt elvégezve a változóim már tartalmazzák az aktuális szenzor adatokat, és közvetlenül beírhatók a definiált üzenetek megfelelő mezőibe. Az *imu* üzenet struktúra elvár egy szenzorral kapcsolatos kovariancia mátrixot is, de az ASSN gyártója ezt nem szolgáltatja, ezért én ezt ismeretlenként kezelve csupa nullával töltöttem fel, az üzenet típus leírásának alapján. A mezők kitöltése után publikálom a két üzenetet, majd a külső ciklusból eldönthető a várakozás mértéke. A korábban meghatározott frekvencia tartásának érdekében a `loop_rate.sleep()` függvény a megfelelő ideig fogja „altatni” a ciklust. Ha már túllépné a műveletsor az 50 Hz-hez tartozó egységidőt, akkor a függvény egyszerűen nullát ad vissza, és rögtön kezdődik előröl a ciklus.

3.6.2 model_odom

Az odometria számítását végző node – *model_odom – (6. számú melléklet*) nagyban eltér az előzőkben leírtaktól. A program elején szintén üzenet definíciók betöltésére van szükség, hogy a beolvasott üzeneteket helyesen dekódolhassa a csomópont, amire csak az üzenet mezőit ismerve van lehetőség. A program elején létrehozok egy „olvasas" struktúrát, majd példányosítok egy *JointState* és egy *Odometry* üzenetet, melyből kiolvasható, illetve írható lesz a tartalmuk. Létrehozom a megfelelő változók az odometria számításához, majd az ROS belső függvényeivel elkészítek két idő típusú változót. Ez azért szükséges, mert az odometria az eltelt idő figyelembevételével számol. A következőkben két függvényt valósítok meg – *jointCallback* - és – *UpdateStep* - melyek a *JointState* üzenetek olvasásáért, és a pozíció frissítéséért felelősek. Elérve a *main()* függvénybe, létrehozom a feliratkozást a *joint_states* topic-ra, valamint az új publikációs topic-ot, az *odometry*-t. Definiálásra kerül a csomópont futásának frekvenciája, melyet ebben az esetben 20 Hz-re állítottam be. A modellautó paramétereinek megadása után nullázom a jelenlegi lokalizációs értékeket, majd belépek a ciklusba. A ciklusban lefut az idő változók aktualizálása és számítása, a pozíció értékek számítása, ami alább látható, majd végül megtörténik az odometria üzenet publikálása.

```
ros::Duration dt;
const ros::Time stamp;
curr_time = ros::Time::now();
dt = curr_time - prev_time;

double linvel = olvasas.velo1*params.wheelradius;
dtheta = linvel/(params.wheelbase)*tan(olvasas.p1);
dx = linvel*cos(theta);
dy = linvel*sin(theta);
UpdateStep(dt.toSec(), stamp);

odom.publish(current_state);
```

A ciklus végén ismét a helyes frekvencia tartásához szükséges időt várok az előzőleg tárgyalt *node*-ban használt függvény segítségével, majd a függvény kiértékelését követően a ciklus újraindul.

3.6.3 UNO_writer

A fenti két csomag elvégzi az összes szenzoradat feldolgozását, valamint az odometria és ezáltal az aktuálisan becsült pozíció kiszámítását, de ezt mind csak abban az esetben tudjuk használni, ha a jármű megmozdul, és a szenzor meg az enkóder is adatot szolgáltat.

A jármű mozgásba hozására egyszerű nyílt hurkú vezérlést kívánok megvalósítani az ROS keretrendszerében. Ennek célja a motorvezérlő PWM[22] jelekkel való kivezérlése, valamint a kormányszervó analóg értékeinek kiküldése USB porton keresztül a mikroszámítógépről a mikrokontroller felé. Az impulzusszéleség-moduláció segítségével a kivezérlés értékét lehet szabályozni nagyfrekvenciás kapcsolásokkal, ami valójában a generált négyszög jelek kitöltési tényezőjét befolyásolja. A nagyfrekvenciás kapcsolásoknak köszönhetően a generált négyszögjelről levehető effektív jel értéke a kitöltési tényezővel arányos.

A soros kommunikációt ebben az esetben a POSIX [26] függvényekkel építettem fel. Ezzel tapasztalatot szereztem egy mélyebb programozási módszerbe, ami később implementálásra fog kerülni a korábban megírt két *node*-ban is. A korábban használt *SerialPort.h* header is ezen függvényeket használja a magjában, de sokkal felhasználóbarátabb módon felépített és egyszerűen megvalósítható megoldással szolgált, mely első alkalommal segítséget nyújtott a fenti csomópontok szerkesztésekor.

Ez a program felépítését tekintve a legegyszerűbb a dolgozatban. Először definiálásra kerül két referencia érték, mely az elküldendő üzenetben fog szerepelni a később leírt struktúra szerint. A *publisher* beállítását követően megadom a működés során elvárt frekvenciát, mellyel a ciklusnak működnie kell, majd kísérletet teszek a port nyitásra. Amennyiben sikerül, úgy a ciklusba belépve a megadott üzenetet sorosan kiírom, majd a kellő időt kivárva, újraindul a ciklus, amennyiben a port nyitás sikertelen, hibakóddal kilép a program A program a *7. számú mellékletben* megtalálható.

A soros írás alkalmával a mikrokontrollernek egy specifikus üzenet kerül kiküldésre, melynek struktúrája: (*#.##,#.##a*) két lebegőpontos két tizedes jegyre kerekített referencia jelet tartalmaz, és egy lezáró ’a’ karaktert. Az ’a’ karakter az üzenetek elválasztásáért, míg a köztes ’,’ a referencia értékek tagolását adja. A soros írás ezután idő alapon ütemezhető, vagy a referencia jelek átadásával egyszerűen vezérelhető külső függvényekből is.

A struktúra használata azért fontos, mert a jelenlegi csapatban működő versenyautó vezérlése is ezt a felépítést követi a programjában, így, ha az ROS rendszer és az „önvezetés” funkciója beépítésre kerül, akkor Jetson TX2 könnyen összehangolható lesz az autóban működő elektronikai vezérlő egységgel.

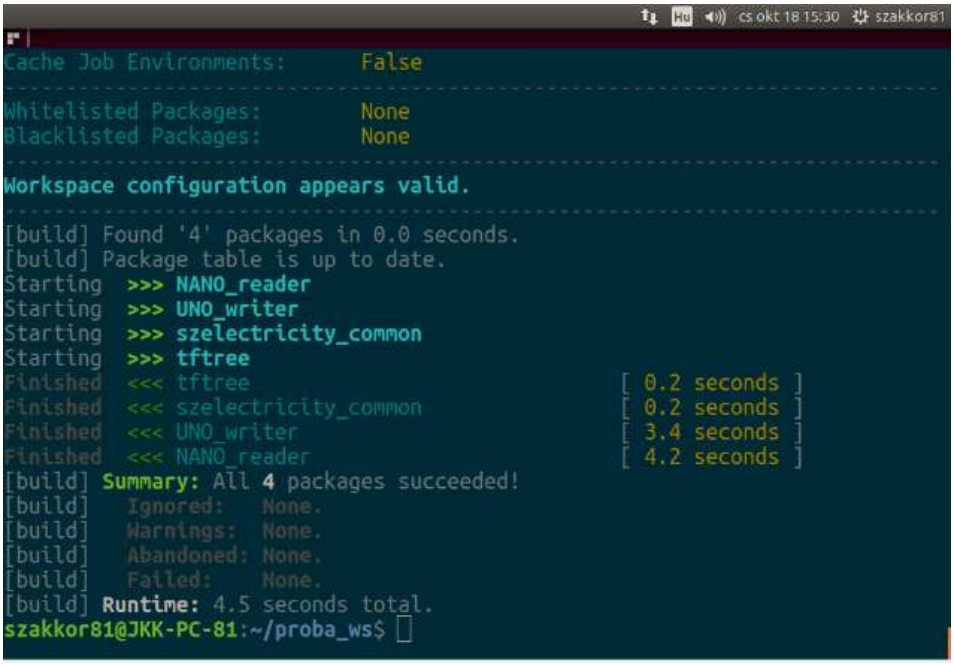
[22] (Pulse-width modulation – impulzusszélesség-moduláció)

3.7 ROS – Node-ok futtatása

A korábban megírt ROS kódokat most kipróbálhattam éles körülmények között a modelljármű fedélzetén. A mikrokontrollereket és a mikroszámítógépet összekötve, az első feladat a soros portok engedélyeinek megadása volt. Ezt Linux rendszerek alatt az alábbi paranccsal lehet megtenni.

```
$ sudo chmod 777 /dev/ttyUSB0
$ sudo chmod 777 /dev/ttyACM0
```

Az elérhetőség garantálását követően a megírt forráskód fordítása következett, mert csak ezt követően lesz valóban működőképes és futtatható a program. A program fordítását a forráskönyvtárban kellett elvégeznem, amire az ROS ajánlása szerint, a *catkin buildert* használtam, melynek eredménye alább látható. (*21. ábra)*



21. ábra: A forráskód építésének eredménye: Sikeres program összeállítás.

Jelenleg még minden részegység egyesével indítandó, de később ez összeintegrálható lesz, annak érdekében, hogy gördülékenyen lehessen méréseket végezni, valamint az önvezető funkciók számára könnyen elérhető legyen a szoftvercsomag összes funkciója. A program működése közben a *NANO_reading* csomag valósítja meg a mikrokontroller olvasását és adatainak rendszerezését, majd megosztását. Ezt követően a *model_odom* csomag elvégzi a szükséges számításokat, és amennyiben megtekintjük az aktuálisan publikált üzenetet abból látható a jármű aktuális pozíció és orientáció értéke X és Y tengely szerint. (*8. számú melléklet*)

4. Navigációs mérések

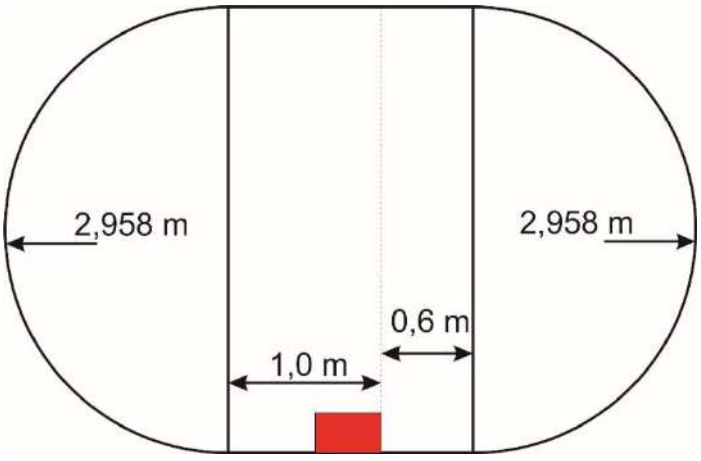
4.1 Vezérlés tesztelése

Mielőtt a navigációs szenzort a modelljárművön alkalmaznám, szükséges volt annak dinamikai problémáit felmérni. [27] Ugyan kinematikailag méretarányosan lett megtervezve, de a modell dinamikai szempontból egyáltalán nem arányos annak eredeti megfelelőjéhez. A hibák feltételezettek, de ezek pontos feltárása szükséges annak érdekében, hogy a későbbi mérések ne csak önmagukban feleljenek a hibákért. Fontos megjegyezni az alábbi rendszer problémákat:

- Nincs Ackermann kormányzás
- Tömegét tekintve a modell nem arányos
- Egy oldalon hajt a jármű
- Analóg kormányszervó visszajelzés egész értékek formájában

A modellautó dinamikájáról többet megtudhattam, ha egy idő alapú programot írva, teszteltem annak viselkedését, egy előre definiált mozgással. Ahogy az az ipari robotoknál is alapfeltétel, és a korábbi dolgozat mérései közben is meghatározó volt, az egyik alapfeladat, hogy egy valamilyen módon definiált útvonal után a robot, vagy jármű visszatérjen a kiinduló pontba. Az ipari robotok bemérése során egy ISO-kockában[23] szokás elvégezni ezen pozíció és ismétlési méréseket [28].

Számomra adottak voltak a szakkörben formálható pályaelemek, amelyekből két egyenesből és két félkörívből álló szakaszt valósítottam meg (*22. ábra*), melyben az autónak vissza kell érnie a kiindulópontjába.



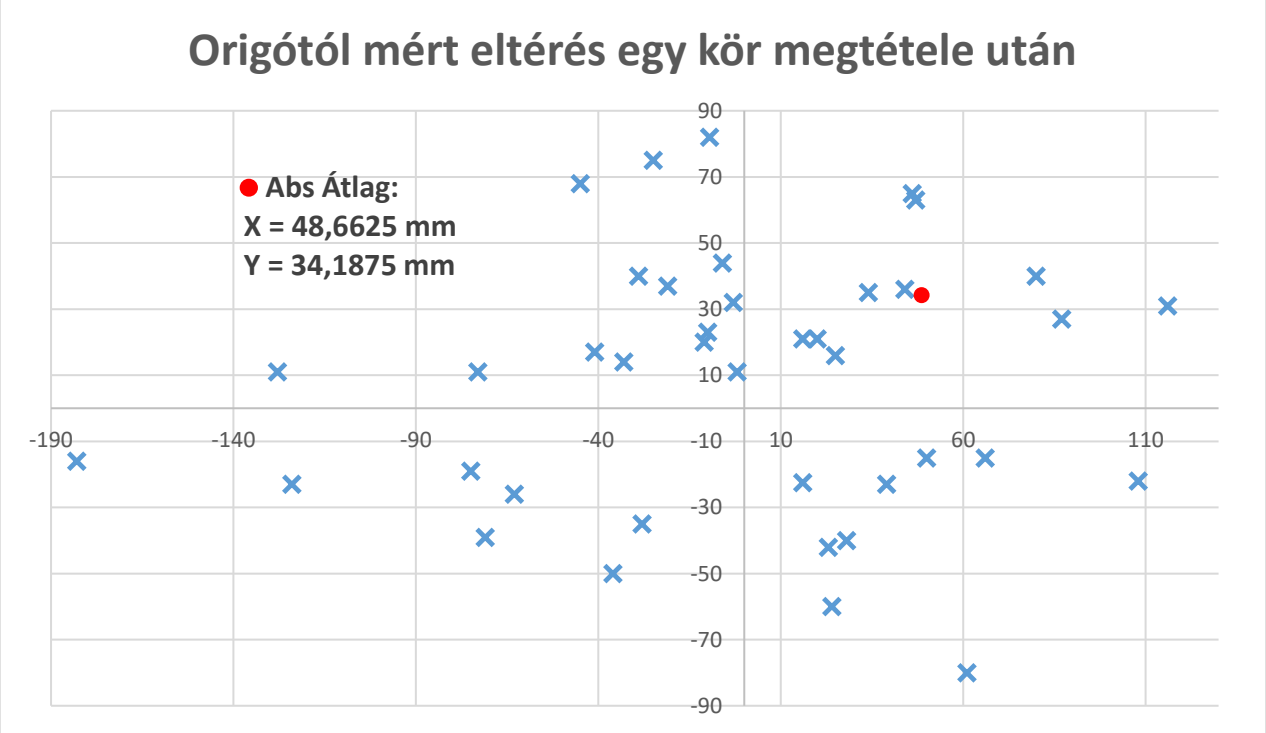
22. ábra. A megtenni kívánt tesztkör.

[23] 1m x 1m x 1m -es szabványos test

A vezérlést Python nyelven írtam, melyben soros kommunikációval küldök, megadott időközönként vezérlő jeleket a motor- és szervovezérlőnek, ezzel megvalósítva a pálya útvonalát. A soros kommunikáció, valamint a várakozásra használt időfüggvény először kimérésre kerültek, melyekből megállapítottam, hogy a függvények alkalmasak egy hiteles mérés kivitelezésére. Alább látható a kód futása során mért rendszeridő, és a különbségekből adódó végrehajtási idő.

```
#3219.429703301   Time before serial_write
#3219.430038246   Time after serial_write, before sleep(1)
#  serial_write = 0.334945 ms
#3220.431092828   Time after sleep(1), before serial_write
#  sleep(1)     = 1001.054582 ms
```

Látható, hogy a soros írás alkalmával kevesebb, mint 1 ezredmásodpercre van szüksége a rendszernek, valamint a várakozás is 1 ezredmásodperc pontosságú másodpercenként. A teljes teszt kód az utolsó 9. *számú mellékletben* található meg. A tesztkódban több vezérlő jel megtalálható egy adott feladatra, ennek oka, hogy az egy oldalon hajtott rendszer már egy ilyen kis tesztkör esetén is nagy oldalcsúszást generál. Így, a párhuzamosságra figyelemmel, kissé módosított értékek szükségesek az elviekben szimmetrikus tesztkör futásához mindkét egyenes szakaszon. A grafikonon (*23. ábra*) látható, átlagosan **0,04866 m** -es eltérés, a nyílt hurkú vezérlésben teljesített tesztkörre vetítve **0,533 %** -os pozíció hibát eredményezett.



23. ábra. Vezérlés mérésének eredményei.

A vezérlő kód módosítása szükséges volt annak érdekében, hogy aktív szabályzás nélkül megtehessen a modell egy egésznek tekinthető „kört”, mely az inerciamérő szenzor vizsgálatához megfelelő adathalmazt ad, és elvárható értékekkel összehasonlítható legyen a szenzor naplózás eredménye. A teszt nagyszámú ismétlése következtében tapasztalható az akkumulátor feszültség szintjének csökkenése. Ennek következtében az alkalmazott egyszerű L298N típusú H-hidas motorvezérlő (*24. ábra*) nem képes állandó feszültséggel táplálni a motort.

24. ábra: L298N motorvezérlő egység. [A9]

Ebből fakadóan az idő alapú vezérlés szét esne, és állandó korrekcióra szorulna. Ezt elkerülendő, egy stabilizált feszültség csökkentő konvertert iktattam az akkumulátor és a motorvezérlő közé, mely ezáltal már képes, fix feszültség szinttel vezérelni a motort, a 40 tesztkör alatt egységesen.

4.2 Szenzor adatok elemzése

Az visszatéréses vizsgálatokat követően, ugyan azon kód használatával, de már soros kommunikációra épített olvasással együtt végeztem el az útvonal méréseket, amelyek során folyamatosan adatnaplózást végeztem. Ennek során rögzítésre került a mikrokontroller rendszerideje, ami időbélyegként szolgált az adatfeldolgozás során. Ezt követően az inerciamérő orientációs adatait és a lineáris gyorsulás értékét mentettem ki. Ezen szenzor értékek mind átestek a Bosch által írt belső feldolgozáson és szenzorfúzión, de további szűrést nem tartalmaznak. Ezt követően kiolvasásra került a kormányszervó állapotának értéke, majd az enkóderből számolt sebesség, fordulatszám, és a megtett út. Az adattípusok feldolgozása során a következő oldalon látható *3. táblázat* szerint kerültek felvételre a különböző adatok.

3. táblázat. Feldolgozott adatok típusa és felbontása.

Név	Típus	Mértékegység	Felbontás
Időbélyeg	Egész	ms	
Orientáció (X-Y-Z)	Tört	deg	4 tizedes pontossággal
Lineáris gyorsulás (X-Y-Z)	Tört	m/s^2	2 tizedes pontossággal
Kormány szög	Egész	deg	[-10° ; 10°]
Motor kimeneti tengely sebesség	Tört	rad/s	2 tizedes pontossággal
Fordulatszám	Egész	rpm	
Megtett távolság	Tört	m	2 tizedes pontossággal

A szenzoros mérések során 12 alkalommal egy-egy kört tettem meg a modellautóval, melyek végén minden alkalommal újra pozícionáltam az autót a helyes kiindulási helyzetbe, mert a vezérlés pontatlanságából adódóan nem volt tökéletes a visszaérkező modellautó pozíciója.

4.2.1 Orientáció – Heading

A szenzoros adatfeldolgozás során az elsődlegesen vizsgált elemem az orientáció volt. Egy kör megtétele 360° változást kell, hogy adjon. A félkörívek elvárt változása 180°. Az orientáció pontatlansága az ismételt mérések együttes összehasonlításával, a *4. táblázatban* látható. Az értékekből jól látható, hogy az egyes műveletek elvégzése még **1 %**-os hibahatáron belül mozog, de a kezdő és végpontok között már **1,538 %-**ra nő. A teljes rendszert vizsgálva az átlagolt eltérés 1,58 fok, ami **0,438 %-**nak felel meg. Amennyiben figyelembe veszem a vezérlés **0,533 %** hibáját, annak tekintetében a BNO055 jól teljesített.

4. táblázat. Összesített orientációs hibák.

Eltérés	Átlag	Minimum	Maximum	Szórás
Egyenes - 1	0,99°	0,63°	1,38°	0,75°
Kanyar -1	2,05°	0,31°	3,69°	3,38°
Egyenes - 2	1,89°	1,44°	2,19°	0,75°
Kanyar - 2	2,49°	0,25°	5,19°	4,94°
Egyenes - 3	0,47°	0,19°	0,75°	0,56°
Kezdő - végpont	5,54°	1,31°	10,00°	8,69°
	Összesített átlag	Min. átlag	Max. átlag	Szórás átlag
	1,58°	0,19°	5,19°	5,00°

A hiba oka visszavezethető mind a vezérlés pontatlanságára, mind pedig a jármű dinamikai anomáliáinak következményeire. Az alábbi táblázatból jól kitűnik, hogy a kanyarodás esetében jóval nagyobb eltéréseket mértem, mely kiemeli a jármű kormányzási elégtelenségét.

4.2.2 Orientáció – Roll – Pitch

Az orientációs vizsgálatok további két paramétere a dőlés és billenés, angol kifejezéssel „roll" és „pitch". Ezen értékek ilyen lassú sebességnél, és a modelljármű által megtett tesztkörben elhanyagolhatók, de fontos megvizsgálni ezen tengelyek hibáját, mert van elvárt értékünk (0°), és ezáltal összehasonlítható a kimenet.

5. táblázat. Dőlés és billenés hibaértékei.

	|Roll|	|Pitch|
Összesített átlag	1,7928°	0,5823°
Minimum átlag	1,1927°	1,4375°
Maximum átlag	2,5885°	0,6042°
Szórás	1,3958°	2,0417°

A fenti, *5. táblázatból* jól látható, hogy azok a tengelyek melyek szándékosan nem kerülnek mozgatásra a mérés során, minimális eltérést mutatnak. Roll esetében átlagosan **0,996 %**-os a hiba, míg Pitch esetén ez csak **0,3235 %**. Ebből leszűrhető, hogy a szenzor **1 %**-os hibahatáron belül pontosan visszaadta a dőlés és billenés értékét.

Az orientációs méréseket sikeresnek ítélem, mert ahogy a korábbi validációs mérések kapcsán látható volt, az 1%-os hibahatárt itt is tartotta a szenzor, amíg műveleti szinten maradt a vizsgálat. A kezdeti és végpontok közti orientációs hiba ugyan már felemelkedett **1,538 %**-ra, de ez nem is kifejezetten a szenzort, valójában a modelljármű teljes rendszerét minősíti, aminek pontatlansága korábban már említésre került. Az akkor a pályára vetített **0,533%**-os hiba az egy százalékos hibahatárhoz hozzáadva jól közelíti a méréseim eredményét.

4.2.3 ASSN - Számított pozíció

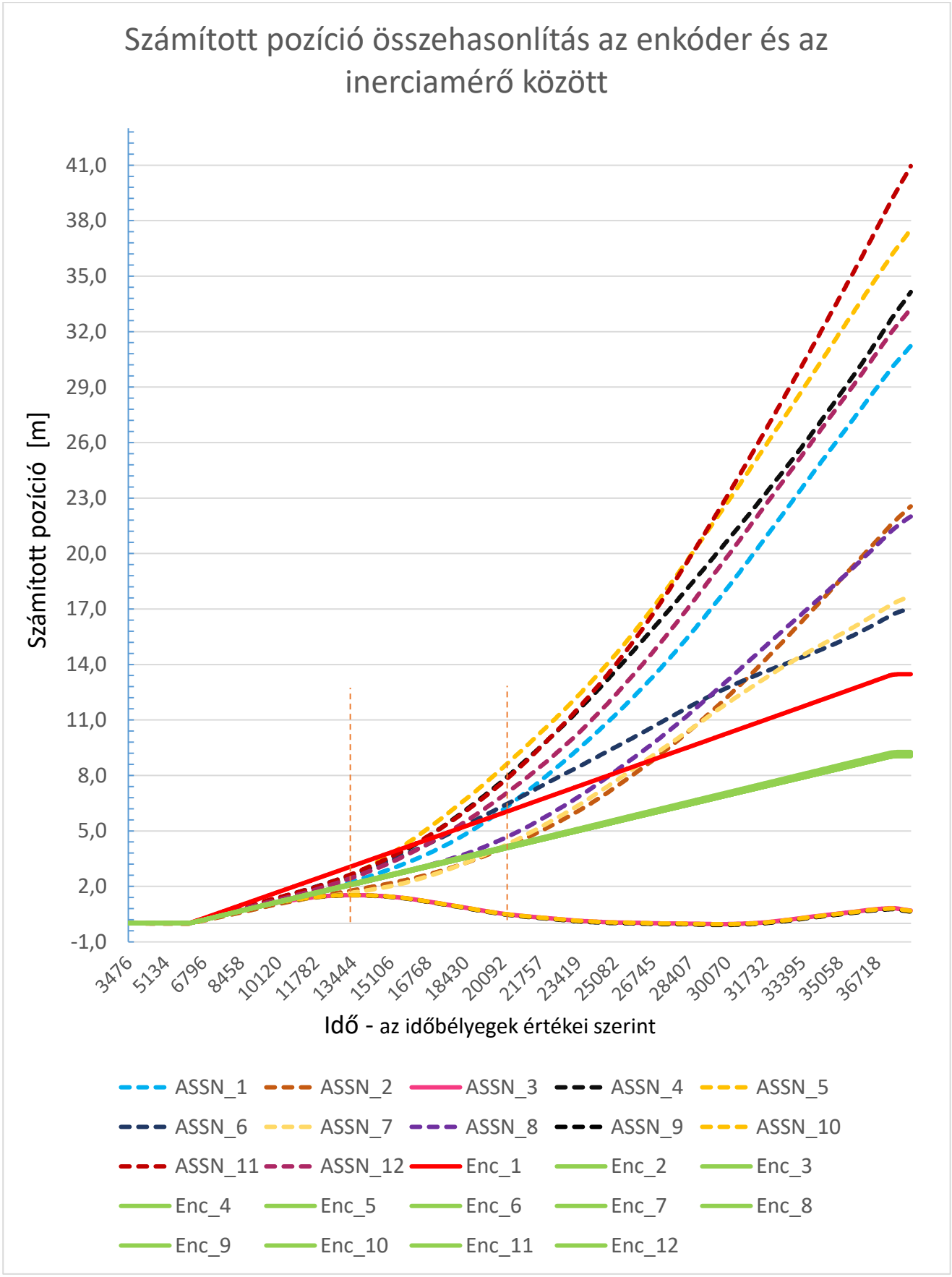
A másodlagos vizsgálati pont, a már jelen lévő gyorsulás adatokból a sebesség majd a pozíció értékek meghatározása. Ennek hibája előre várható volt, de mértéke és szórása kíváncsiságra adott okot, hiszen a BNO055-ös chip belső szenzorfúziójáról nem tudni pontos adatokat, illetve nem ismertek az algoritmusok, amiket használ. A számított távolságot a gyorsulás, majd a sebesség integrálásával számítottam. Ehhez a művelethez a Newton-Cotes formulák közül a trapéz formulát [29] használtam fel, mely a 6.egyenletben látható.

$$\int_a^b f(x)dx \approx \frac{b-a}{2}\left(f(a)+f(b)\right) \tag{9}$$

Az idő teltével a hiba összeadódik, és erőteljes eltérés figyelhető meg a grafikonon. (*25. ábra*) A szenzoros eredményeket összevetve az enkóder stabil jelével, és az abból kiszámolt távolsággal, világosan látszik, hogy a gyorsulásmérőkre támaszkodni önállóan az indulástól számítva körülbelül 8 másodpercig lehet. Az enkóder adataiból számolt út átlagban 9,4915 m, ami 0,3755 m-rel több, mint a teszt kör hossza. Ez a **4,119 %-**os eltérés arra enged következtetni, hogy az egész számú fordulatszám kijelzés pontosítást igényel, és mindenképpen tizedes törtként kell kezelnem ezt is a későbbiekben.

Három olyan mérés volt melyet érdemes volt időben tovább vizsgálni. Ezek 17 másodpercig tudták jól közelíteni az enkóder útszámítását de ekkor már 0,3 m-rel túllőttek azon, amely azért kritikus, mert ennyit az enkóder számítások is átlagban túllőttek. Végértékük szerint ez a három mérés is kritikusnak bizonyult, mert legalább **87,14 %-**kal haladták meg az enkóderből számított megtett távolságot.

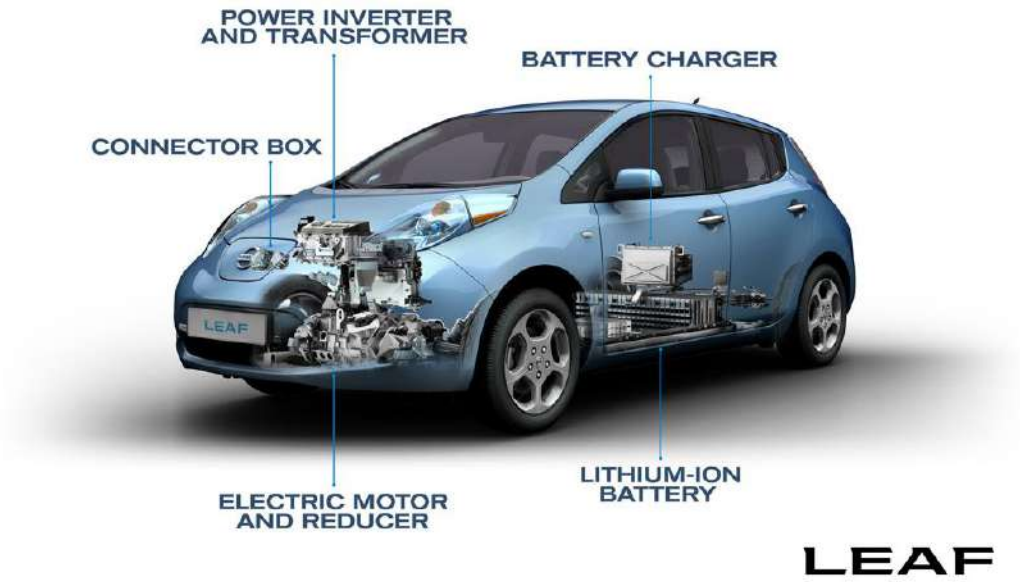
A grafikonon a tizenkét enkóder mérésből egy volt eltérő, ami hibásnak tekinthető ez piros színnel látszik, míg a többi szinte egyként jelenik meg zöld színnel. A mérések közti szórás **0,2222** m volt, eltekintve az egy hibás méréstől, mely **4,03** méteres különbséget adott. Ettől eltérően az integrálást követően az inerciamérőből számított pozíció végső szórása **40,31** m, melyből a negatívba tartó mérések eltávolítása, is csak **23,87** m-re redukálta a szórást. Ezen adatok további szűrést igényelnek, mely után várhatóan jobban közelítik majd az enkóder alapú számításokat.



25. ábra. Távolság számítás összehasonlítása.

5. Tesztelés egy közúti járműben

A fejlesztés következő lépése, hogy az infrastruktúra független navigáció helyet kapjon egy közúti tesztjárműben. Ez a jármű az egyetemen működő Járműipari Kutatóközponthoz tartozik, amivel jelenleg is végeznek önvezető funkció teszteket forgalomtól lezárt területen, illetve a Zalaegerszegen megépített tesztpályán. A jármű alapvetően egy kereskedelmi forgalomban kapható, teljesen elektromos hajtással rendelkező Nissan Leaf (*26. ábra*), amit a cég 2010-ben adott ki, hazánkban pedig 2013-ban kezdődött meg forgalmazása.



26. ábra: A Nissan Leaf technikai áttekintő ábrája. [A10]

A gyártó szerint egy töltéssel körülbelül 175 km-t képes megtenni a 24kWh kapacitású akkumulátorról, így a rövidpályás tesztelésekre kifejezetten alkalmas. [30] Manapság minden gyártó, így a Nissan is CAN hálózatot használ autójában. Ennek a hálózatnak az elérése, olvasása és írása a cél. A tervek szerint használatos -Jetson TX2- fedélzeti számítóegység segítségével kell megvalósítani a feladatot. A korábbi fejlesztések során készült az autóhoz egy egyedi biztonsági rendszer, mely az autó rendszerét, és a fejlesztés alatt álló alsó- és felsőszintű irányítórendszereket elválasztja, melynek alapvető biztonságtechnikai okai vannak. Ez biztosítja, hogyha bármilyen vezérlési parancs helytelen működést okozna, azonnal lekapcsolható legyen az autó egy vészgomb segítségével. Ennek hatására azonnal maximális fékerő kivezérlés történik a fékszenzoroknak, valamit a gázpedál és kormánynyomaték szenzor nulla értékbe áll.

A navigációs próbafutás előtt elsődlegesen a kommunikációs hálózat megfelelő illesztése és tesztelése a feladat. Ennek érdekében a Jetson TX2-t kell a hálózathoz illesztenem.

5.1 Jelenlegi navigációs rendszer összehasonlítása

Az autó jelenleg egy a KVH által gyártott nagypontosságú GPS pontosítású inerciamérőt [31] tartalmaz, mellyel szintén tesztek zajlanak. Az egyik feladat számomra ennek a szenzornak a lecserélése, mert ugyan kutatás-fejlesztésre megfelelő, de katonai minősítése miatt valós piaci szerepe az önvezető autók körében igen kicsi. Ehhez hozzájárul horribilis ára, melyet egy autógyártó cég sem fog megfizetni szériagyártású autóért. Az árakról összehasonlítás a *6. táblázatban* található.

6. táblázat: Említett navigációs eszközök összehasonlítása.

Eszköz neve	Mérete (mm)	Súlya (g)	Ára (Forint)
Adafruit BNO055	20x27x4	3	18900
KVH GEO-FOG 3D	96,4x94x113,1	740	~16 millió

Ennek okán fel kell mérnem, hogy egy olcsó MEMS szenzor képes-e helyettesíteni egy ipari INS[24]-t, és ha igen, milyen pontosság érhető el vele. Fontos szempont, hogy egy moduláris rendszer részeként, könnyen integrálható legyen az általam használt eszköz, és a későbbiekben akár más járművekbe is integrálható legyen, aminek egyik alapfeltétele, hogy piacképes legyen maga az eszköz. Ha sikerülnek a mérések, de az eszköz pontossága nem elegendő, az is megfontolható, hogy két vagy három eszköz kerül elhelyezésre az autó különböző pontjain, melyek adatait fúzionálva, már elérhető lesz a feltételnek szabott pontosság. Az ROS mögöttes futtatása természetesen alapvető szükséglet, de ez a KVH szenzorra (*27. ábra*) is igaz, mert azon adatok is utólagos feldolgozásra szorulnak.

27. ábra: KVH GEO FOG 3D Dual inerciális navigációs eszköz. [A11]

[24] Inertial Navigation System – Inerciális Navigációs Rendszer

5.2 CAN kommunikáció felépítése az autóval

Az tesztautóban jelenleg két CAN hálózat van kiépítve, melyből egy az autó natív hálózat, míg a másik az biztonsági rendszer csatlakozó fedélzeti egységnek van fenntartva. A Jetson illesztése helyileg egyszerű, hiszen a csomagtartóban bőséges hely áll rendelkezésre. A CAN csatlakoztatásához az Nvidia két csatornát készített a Jetson fejlesztő környezetén, melyet elláttak a vezérlőkkel is. Ezeket CAN0 és CAN1 interfésznéven lehet megtalálni, miután a kernelben[25] engedélyezésre került ezek használata. Ha ezt sikerül elvégezni, akkor elkezdhetem a kommunikáció illesztését az interfészekhez. A CAN hálózatok megkövetelik a következő rész egységeket:

- CAN vezérlő
- CAN adó-vevő
- 120 Ohmos lezárások a kommunikációs vonalon

Miután a Jetson TX2 tartalmazza a vezérlőt, adó-vevő beszerzése és illesztése volt a következő feladat. Külföldi tesztek már bizonyították a hardveres támogatottságot, és a CAN-busz sikeres üzembehelyezését egy SN65HVD230 típusú Texas Instruments által gyártott adóvevő chip segítségével. Ez a chip kifejezetten alacsony fogyasztású adó-vevő, ami beleillik a később összeépítésre kerülő rendszerbe, megfelel az ISO 11898-2 standardnak és rendelkezik hővédelemmel. Az eszköz illesztése tűzdelő kábelekkel történt az alábbi *7. táblázat* szerint.

7. táblázat: Jetson TX 2 csatlakozó kapcsolata a CAN adó-vevő modulokkal.

Nvidia Jetson TX2	SN65HVD230 Arduino Modul - 1	SN65HVD230 Arduino Modul - 2
GPIO – 1	3V3	
GPIO – 34	GND	
J26 – 5	Can0 RX	
J26 – 7	Can0 TX	
	CAN High	CAN High
	CAN Low	CAN Low
J26 – 2		3V3
GPIO – 39		GND
J26 – 15		Can1 RX
J26 – 17		Can1 TX

[25] Az operációs rendszer magja, mely a hardver erőforrásaiért felelős.

5.3 Kernel rebuild

A rendszermag újraépítése nem egy mindennapos feladat, Windows esetén ez szóba sem jöhetne, mert nincs hozzáférése az embernek, és a Microsoftnál is csak kevesen nyúlhatnak bele a fejlesztések során. Ennek oka, hogy olyan rendszerjellemzőket változtathatunk meg, amelyek az egész rendszerre kiható kárt, vagy hibát okozhatnak. Egy piacilag stabil cég, milliárdnyi felhasználóval ezt nem teheti meg, ennek okán nem is nyílt forráskódú a Microsoft operációs rendszere. Ettől eltekintve az Ubuntu Linux nyílt forráskódú, és lehetőséget ad a kernel testre szabására. A fejlesztőkörnyezettel kapcsolatosan teljes támogatás van a kernel újraépítését tekintve [32]. Ennek elérése a Github[26]-on egyszerűen megtehető, majd az alábbi parancsok végrehajtásával újraépítettem a rendszermagot.

A megosztott források másolása a saját fejlesztőeszközre, belépés a mappába:

```
$ git clone https://github.com/jetsonhacks/buildJetsonTX2Kernel.git
$ cd buildJetsonTX2Kernel
```

A gyártó honlapjáról elérhető kernel forráskódok letöltése a rendszer /usr/src/kernel mappájába:

```
$ ./getKernelSources.sh
```

Ha a letöltés véget ért, akkor egy grafikus felület nyílik, ahol látható minden jelenlegi modul, illetve hozzáadhatom a plusz modulokat, amik a CAN kommunikációs interfészekhez szükségesek. A modulok ellenőrzését követően a következő parancs segítségével megépítettem a kiválasztott modulokat.

```
$ ./makeKernel.sh
```

Végül a megépített új kernel képet átmásoltam a /boot mappába, ezzel biztosítva, hogy a következő indulás alkalmával a CAN hálózat kezeléséhez kapcsolódó és szükséges modulok, már elérhetők legyenek a rendszer számára.

```
$ ./copyImage.sh
```

Az újraindítás után telepítettem a CAN-hez szükséges fejlesztői programokat az alábbi parancs segítségével, és ezzel készen állt a rendszer a CAN kommunikációra.

```
$ sudo apt-get install can-utils
```

[26] Internetes verzió kezelő szolgáltatás, mely programozók számára biztosít tárhelyet a különböző forráskódok számára

5.4 CAN interfész konfigurálás

A kernel modulok hozzáadásával a rendszer már képes kezelni az eszközöket, de ezek még további konfigurációra szorulnak ahhoz, hogy adatkommunikáció indulhasson a csatornákon. Az engedélyezett modulokat használatba kellett vennem:

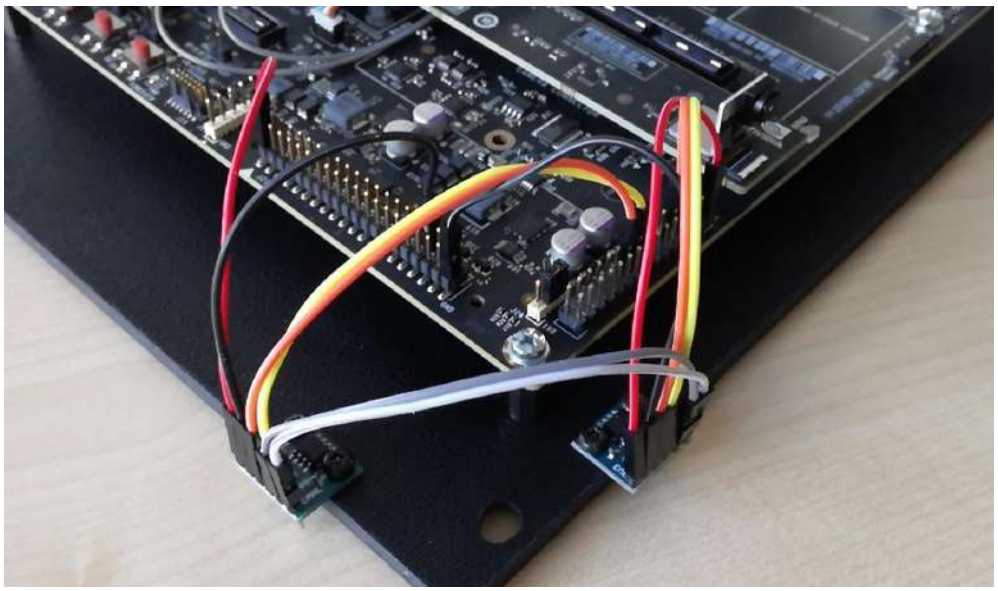
```
$ sudo modprobe can
$ sudo modprobe can-raw
$ sudo modprobe can_dev
$ sudo modprobe mttcan
```

Ezt követően definiálnom kellett a két interfészt, és azok tulajdonságait. Típusként a „can” míg hálózati sebességként az autóban működő korábban kimért CAN hálózat értéke -500000- lett megadva.

```
$ sudo ip link set can0 type can bitrate 500000
$ sudo ip link set can1 type can bitrate 500000
```

Végül, akárcsak egy kapcsolóval, be kellett kapcsolnom az interfészeket, hogy a felcsatlakoztatott CAN adó-vevő modulok kommunikáció képesek legyenek. A csatlakoztatott modulok alább láthatók. (*28. ábra)*

```
$ sudo ip link set up can0
$ sudo ip link set up can1
```



28. ábra: A Jetson TX2 fejlesztőkörnyezethez csatlakoztatott két CAN adó-vevő.

5.4.1 Rendszer szolgáltatás felállítása

Az előzőleg említett beállításokat minden indítás/újraindítás után el kellene végeznem ahhoz, hogy bármilyen tesztet végezhessek a rendszeren a CAN kommunikációval, ezért létrehoztam egy rendszer szolgáltatást [33], mely minden alkalommal elvégzi ezt a műveletsort. Beépítve ezt az Ubuntu ütemezett feladatlistájába a következő indítástól kezdve a fentebb leírt parancsok automatikusan lefutnak. Erre azért van szükség, mert a fedélzeten működő számítási egységnek azonnal tisztában kell lennie a körülötte zajló folyamatokkal és a többi kisebb vezérlő egységgel. Ahogy az a valós járművekben is megvalósított, úgy itt is szükségesnek tartottam a kommunikációs interfészt, már a boot[27]-olás folyamataként „bekapcsolni”.

Az ütemezett Ubuntu rendszer feladatok egy részét elérhetjük a */etc/systemd/system/* mappában. Ehhez volt szükséges hozzáadnom egy egyéni szolgáltatást, ami egy külső fájl meghívásával, mely futtatható állományként már engedélyt kapott, elvégzi a fentebb említett műveleteket. A szolgáltatás leírásának kötött formája van, amit a rendszervezérlő egyértelműen értelmezni tud. Az alábbi kódban olvasható a szolgáltatás definiálása.

```
[Unit]
Description=Can interface initialization
After=jetson_clocks.service

[Service]
ExecStart=/home/nvidia/can_init.sh

[Install]
WantedBy=default.target
```

A definíció során meg kellett adnom a szolgáltatás leírását, és hogy milyen másik szolgáltatás után legyen beütemezve. Továbbá meg kellett adnom azt is, hogy milyen műveletet végezzen el a szolgáltatás. Végül definiálnom kellett, hogy a szolgáltatást „kinek” ütemezze be a rendszer, milyen runlevel[28]-hez legyen hozzárendelve. A szolgáltatás engedélyezése és elindítása ezen felül már csak két parancs kiadását igényelte a parancssorban, melyek alább olvashatók.

```
$ sudo systemctl enable can.service
$ sudo systemctl start can.service
```

[27] Rendszerbetöltés a számítógép bekapcsolását követően.
[28] Futási szint – Megadja, hogy a Linux milyen operációs módban van. Ez határozza megy, hogy konzol vagy grafikus megjelenítő segítségével indul, milyen hálózati interfészeket használhat és hogy egyszerre hány felhasználó léphet be.

5.5 CAN hálózati tesztek

Miután elkészült az automatizált szolgáltatás, és a CAN hálózat igényei kielégítésre kerültek, elkezdhettem a kommunikációs teszteket. Első alkalommal a két felcsatlakoztatott eszköz között próbáltam ki a CAN hálózatra való írást és olvasást. Ezen funkciók kipróbálására szolgál a korábban telepített *can-utils* csomagból a *candump* és a *cangen* parancsok. A

```
$ cangen can0
```

parancs segítségével véletlenszerű tartalommal bíró üzeneteket küldök a CAN0 interfészen keresztül, míg a

```
$ candump can1
```

parancs kiadásával minden, a CAN0 interfészen beérkező üzenetet megjelenítek a parancssorban.

Ezen funkciók az ROS alatt megvalósításra kerültek már, de futtatásuk előzetes fordítást, és más parancsokat igényelnek. Ezen túl viszont előnyökkel szolgál a keretrendszer használata, hiszen így sok az üzenet küldés-fogadásra épülő funkció használható a tesztek során, ami már alapvetően implementálva van az ROS környezetében.

Az ROS alatt használható egyik legjobb csomag a CANOPEN, mely támogatja a szabványos és a nyílt típusú üzenetkezelést. [34] Részegységei közt megtalálható több specifikus csomag, ilyen például a motorvezérlő adatainak értelmezésére, valamint a rendszerszinkronizációra szolgáló gyűjtemény. A csomag használata akár egy OBD-II[29] csatlakozó elérése után is megtehető egy laptopról, melynek segítségével egy közúti jármű adatait akár menetközben is dekódolhatjuk. Ebben az esetben persze, nem érünk el minden CAN hálózatot az autóban, mert ez gyártónként változó, de mindenképp belelátunk a rendszerbe, ha nem is teljes mélységében. Ezen irányú fejlesztés, munkám későbbi terveiben is szerepel.

A legfontosabb csomagrész a *socketcan_bridge*, ami összeköttetést teremt a CAN interfész és az ROS között, így láthatók lesznek a CAN modulok közt küldött üzenetek az ROS keretrendszerén belül. A *10. számú mellékletben* látható egy teszt az írás-olvasás műveletéről, melyben a 10-es ID-val rendelkező üzeneteket dekódolom. A működésben lévő feldolgozást végző programkód a *11. számú mellékletben* található. A feldolgozás egy több esetből álló switch funkció, mely eseteinek magjában hexadecimális üzenetek összefűzése és decimális átalakítása történik meg, olykor egyedi mérőszámokkal felszorozva azokat.

[29] On-board Diagnostics : Fedélzeti diagnosztika csatlakozó, melyet a SAE J1962-es standardja specifikál, és A vagy B típusú verziója, körülbelül a 2000-es évek óta minden személy- és tehergépjárműben megtalálható

6. Konklúzió

A jelen dolgozatomban bemutatott munkát megelőzően elkészítettem egy komplex validációra alkalmas tesztpályát, mely ipari robotok felhasználásával tud méréseket végezni a rögzítésre kerülő szenzorokon. Ezen munka folytatásaként a validált szenzorral megépített modelljármű kezdetleges navigációját vizsgáltam meg. Ehhez egy determinisztikus idő alapú vezérlést kellett megvalósítanom, valamint egy real-time közeli soros kommunikációt és adatnaplózást. Ezen adatok utólagos feldolgozása adta eredményül a vizsgált rendszer, azaz a modelljármű jelenlegi navigációs pontosságát. A korábbi validációs eredményekkel egybevágó eredményekkel tudtam lezárni a jármű dinamikai tesztjeit, mely igazolta, a validációs munka sikerességét és a navigáció elvárt pontosságát egyaránt. A projekt folytatásaként az összetett robotikai rendszerünk egy részletével foglalkozva, megalkottam a szenzorolvasáshoz és mikrovezérlő íráshoz kapcsolódó csomópontokat, majd elkészítettem az odometria számítását végző részegységet. Ebben a feladatban a soros kommunikáció megvalósítása volt a legfontosabb elem. Ezt kétféleképpen is meg tudtam valósítani, ami a biztonságkritikus rendszerünk számára később előnyökkel szolgálhat. A megírt kódok fordítását és beüzemelést követően sikerrel tapasztaltam, hogy a rendszer működőképes a mikroszámítógépen és a további integrációra készen áll. A kommunikáció másik oldala a CAN hálózat felé mutat, mert a járműipar már évtizedek óta ennek a standardnak a nyomvonalán halad. A protokoll kulcsfontosságú elem a fejlesztésben, hiszen a modulárisnak tervezett rendszer, később akár egy könnyen integrálható fedélzeti egységként piacra kerülhet a jelenleg piacon kapható autók számára, hogy azok biztonság technikailag felérjenek az akkor már intelligens autókhoz képest. Ennek okán az autókban jelenlévő CAN hálózat illesztése elsőrendű feladat. A hardveres illesztés megvalósítása piaci elemekkel megoldható volt számomra, melyet a szoftveres adatfeldolgozás szerkesztése követett. Az általam készített program kifejezetten a Nissan Leaf CAN rendszerét képes dekódolni. Egyelőre még csak parancssorban, grafikus felület nélkül működik, de karakteresen megjeleníteni az autó rendszerétől kapott összes információt.

A Jetson TX2-es fejlesztőkörnyezetben végzett munka rendkívül előremutató volt számomra, és arra sarkall, hogy az elektro-mobilitás és jármű fejlesztés további feladataiban részt vegyek. Fiatal mérnök hallgatóként az elvégzett feladatoknak köszönhetően már most sok hasznos tapasztalattal lépek a mesterképzés kapujába, mert lehetőségem nyílt a jelenlegi technika majdnem élvonalában dolgozni. Ennek a munkának és tapasztalatszerzésnek a folytatását remélem a következő képzési időszak alatt is.

7. További fejlesztési lehetőségek

- **µC kód fejlesztés**

A későbbiekben szeretném a mikrokontroller kódját is C++ nyelven átírni, hogy egységes legyen a projekt ezen része, valamint esetleges sebesség növekedést tudjak elérni. Fontosnak tartom, hogy a kalibrációs ofszet adatok ne csak az EEPROM-ból legyenek beolvashatók, hanem a programkódból is, az inicializálás folyamataként. Ez azért lenne fontos, hogy egy esetleges memóriavesztés, sérülés okából fakadóan ne vesszenek el az alapértékeim, mert ez a navigációt alapjaiban megdöntené.

- **Több körös mérések végzése szabályozó algoritmusokkal**

A mérések során felmerülő dinamikai problémák javítása és (vagy) kiküszöbölése szabályozás segítségével, annak érdekében, hogy a szenzort több körös mérésekben is tesztelni lehessen, és szeretnék lézeres mérőeszközöket használni a pozíció ofszet mérések alkalmával Ez a valósághoz köthető távlati tervek miatt elengedhetetlen, és a modelljármű funkcionális fejlesztése során kulcsfontosságú.

- **Validáció más ASSN üzemmódban**

A dolgozatban felhasznált BNO055-ös chip képes több más üzemmódra is, melyben nem működik a belső szenzorfúzió, de jobban paraméterezhetők az egyes szenzorelemek, ami nagyban befolyásolhatja a méréseket, és esetleg pontosíthatja azokat. Ennek vizsgálata a későbbi terveim közt szerepel, mert elképzelhető, hogy a belsőleg működő algoritmusok nem olyan pontosak és hatékonyabb ROS alapú megoldásokat tudok megvalósítani.

- **CAN kommunikáció fejlesztése**

A jelenlegi CAN alkalmazás további fejlesztése is terveim közt szerepel, ahol már grafikus felületen lehet nyomon követni az üzeneteket és azok dinamikus tartalmát, valamint az értékadás is egyszerűen kezelhető. A Jetson fejlesztőkörnyezet mellé, szeretnék kialakítani egy olyan hardveres egységet, meg tartalmazza a felhasznált adóvevőket, és egyszerűen, kompakt módon illeszthető a beültetett csatlakozókhoz. Ez mind a fejlesztés, mind a későbbi modularitás szempontjából fontos.

Köszönetnyilvánítás

Ezúton szeretnék köszönetet mondani témavezetőmnek, Dr. Ballagi Áronnak, aki szakértelmével, szemléletes magyarázataival és hasznos konzultációival segítséget nyújtott dolgozatom elkészítéséhez.
Köszönöm hasznos tanácsait külső konzulensemnek, Kőrös Péternek, aki verseny és konstrukciós tapasztalatával segítséget nyújtott a dolgozat témájában.
Köszönöm a segítséget Szeli Zoltánnak, aki a mérések és a modelljármű előkészítésénél a teljesítményelektronika összeállításában nyújtott segítséget, valamint Hajdu Csabának, aki a C++ programozásában volt nagy segítségemre a Robot Operációs Rendszer kapcsán.
Hálával tartozom továbbá szüleimnek és testvéremnek, hogy tanulmányaim során türelemmel és megértéssel támogattak, és minden helyzetben mellettem álltak.

A szakdolgozat elkészítését a GINOP-2.3.4-15-2016-00003 Felsőoktatási és Ipari Együttműködési Központ a Széchenyi István Egyetemen projekt támogatta.

Irodalomjegyzék

[1] *Shell Eco-marathon információk*:
https://www.shell.com/energy-and-innovation/shell-ecomarathon/about.html
utolsó látogatás: 2018. november 14.

[2] NovAtel Inc.: *Navigation attitude basics*
https://www.novatel.com/solutions/attitude/
utolsó látogatás: 2018. november 14.

[3] NovAtel Inc.: *An Introduction to GNSS*
NovAtel Inc., Calgary, Canada (2015), ISBN: 978-0-9813754-0-3, p. 62-65.

[4] Dr. Budó Ágoston: *Kísérleti Fizika I*
Tankönyvkiadó Vállalat, Budapest (1968), p. 181-186.

[5] Tél Tamás: *A Coriolis-erő és a modern környezetfizika: A lefolyótól a ciklonokig*
Eötvös Loránd Fizikai Társulat: Fizikai Szemle, LVI.évfolyam, 8.szám p.263.

[6] A. D. King: *Inertial Navigation – Forty Years of Evolution*
GEC REVIEW, VOL. 13, NO. 3, (1998), pp. 140-149.

[7] Makan Gergely: *Versenykajak mozgásának mérése és a mért jelalakok analízise*
MSc Diplomamunka: Szegedi Tudományegyetem TTIK Kísérleti Fizika Tanszék, 2012

[8] NovAtel Inc.: *IMU Errors and Their Effects*
NovAtel Inc., Calgary, Canada (2014), ISBN: 978-0-9813754-0-3, p. 62-65.

[9] Almár Iván, Horváth András: *Űrhajózási Lexikon*
Akadémiai Kiadó, Budapest, (1984), ISBN: 9633265444, p. 431.

[10] Tóth Szilvia Ágnes: *Kvaterniók*
BSc Diplomamunka, ELTE, Algebra és Számelmélet Tanszék, 2010

[11] *Apollo 8 Report:*
http://web.mit.edu/digitalapollo/Documents/Chapter6/hoagprogreport.pdf
utolsó látogatás: 2018. november 14.

[12] Lentin Joseph: *ROS Robotics Projects*
Packt Publishing, (2017), ISBN: 978-1-78355-471-3, p 1-20.

[13] *BNO055 Chip Datasheet:*
https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/
BST_BNO055_DS000_14.pdf
utolsó látogatás: 2018. november 14.

[14] Patrik Axelsson: *Evaluation of Six Different Sensor Fusion Methods for an Industrial Robot using Experimental Data*
10th IFAC Symposium on Robot Control, VOL. 45, NO. 22, (2012), p. 126-132.

[15] *Nvidia Jetson információk*:
https://devblogs.nvidia.com/jetson-tx2-delivers-twice-intelligence-edge/
utolsó látogatás: 2018. november 14.

[16] Gyimesi László: *Digitális jelfeldolgozás*
SZIF Universitas Kft., Győr, (1999), p. 5.3 fejezet.

[17] Dr. Fodor Dénes, Dr. Szalay Zsolt: *Autóipari kommunikációs rendszerek*
Pannon Egyetem, (2014), p. 120-121.

[18] Bokor József, Gáspár Péter: *Járműfedélzeti kommunikáció*
Typotex Kiadó (2012), ISBN 978-963-279-612-3, p. 171-173.

[19] Avi Silberschatz, Peter Baer Galvin, Greg Gagne: *Operating System Concepts*
John Wiley & Sons, Inc, (2012), ISBN: 978-1-118-06333-0, p. 283-287; 357-360; 781-827.

[20] *Windows - Linux benchmark:*
http://www.bitsnbites.eu/benchmarking-os-primitives/
utolsó látogatás: 2018. november 14.

[21] *ROS telepítési segédlet:*
http://wiki.ros.org/kinetic/Installation/Ubuntu
utolsó látogatás: 2018. november 14.

[22] Ágoston Máté, Gulyás Péter: *Gyors szenzor validáló módszer fejlesztése inerciális navigációs rendszerhez*
TMDK dolgozat, Széchenyi István Egyetem, Automatizálási Tanszék, 2017/18 tavasz

[23] Dr. Ballagi Áron: *Robotok vezérlése tananyag*
http://www.sze.hu/~ballagi/Robottechnika/RobVez.pdf
utolsó látogatás: 2018. november 14.

[24] *ABB robot datasheet*:
https://search-ext.abb.com/library/Download.aspx?DocumentID=PR10031EN_R15&LanguageCode=en&DocumentPartId=&Action=Launch
utolsó látogatás: 2018. november 14.

[25] *RAPID programozási leírás*:
https://library.e.abb.com/public/688894b98123f87bc1257cc50044e809/Technical%20reference%20manual_RAPID_3HAC16581-1_revJ_en.pdf
utolsó látogatás: 2018. november 14.

[26] Donald Lewine POSIX programers Guide:
O'Reilly & Associates, Inc, Sebastopol, CA, (1991), ISBN: 978-0937175736 p. 147-150

[27] Gulyás Péter: *Inerciális navigáció egy modelljárművön*
TMDK dolgozat, Széchenyi István Egyetem, Automatizálási Tanszék, 2018/19 ősz

[28] Hongyi Liu: *Environment for Industrial Robot Performance Evaluation*
MSc Diplomamunka: KTH Royal Institute of Technology CSC, 2015

[29] Bárczy Barnabás: *Integrálszámítás*
Műszaki Könyvkiadó, Budapest (1969), ISBN: 963-10-9731-5, p. 320.

[30] *Nissan Leaf Manual:*
http://large.stanford.edu/courses/2012/ph240/landreman1/docs/2012-Nissan-LEAF.pdf
utolsó látogatás: 2018. november 14.

[31] *KVH Navigációs szenzor:*
https://www.kvh.com/Commercial-and-OEM/Gyros-and-Inertial-Systems-and-Compasses/Gyros-and-IMUs-and-INS.aspx
utolsó látogatás: 2018. november 14.

[32] *Jetson TX2 kernel rebuild:*
https://www.jetsonhacks.com/2017/03/25/build-kernel-and-modules-nvidia-jetson-tx2/
utolsó látogatás: 2018. november 14.

[33] *LINUX rendszerszolgáltatások a systemd segítségével:*
https://www.linuxvoice.com/issues/010/systemd.pdf
utolsó látogatás: 2018. november 14.

[34] *ROS canopen csomag információk:*
https://github.com/ros-industrial/ros_canopen
utolsó látogatás: 2018. november 14.

Ábrajegyzék

[A1] https://ae-bst.resource.bosch.com/media/_tech/bilder/products/motion/hubsnodes/bno055/pro_BNO055_m.jpg
(utolsó látogatás időpontja: 2018. november 14.)

[A2] https://s.zeptobars.com/mpu6050-HD.jpg
(utolsó látogatás időpontja: 2018. november 14.)

[A3] https://i2.wp.com/www.jetsonhacks.com/wp-content/uploads/2015/04/P02_F09_625.jpg?ssl=1 és https://i0.wp.com/www.jetsonhacks.com/wp-content/uploads/2015/04/P02_F10_625.jpg?ssl=1
(utolsó látogatás időpontja: 2018. november 14.)

[A4] http://web.mit.edu/digitalapollo/Documents/Chapter6/hoagprogreport.pdf
(utolsó látogatás időpontja: 2018. november 14.)

[A5] https://www.gotronic.fr/ori-module-9-dof-ada2472-23896.jpg
(utolsó látogatás időpontja: 2018. november 14.)

[A6] https://ae-bst.resource.bosch.com/media/_tech/media/datasheets/BST_BNO055_DS000_14.pdf
(utolsó látogatás időpontja: 2018. november 14.)

[A7] https://devblogs.nvidia.com/wp-content/uploads/2017/03/Figure1_TX2-e1488772330657-300x206.png
(utolsó látogatás időpontja: 2018. november 14.)

[A8] https://www.raspberrypi.org/app/uploads/2017/05/Raspberry-Pi-3-hero-1-1571x1080.jpg
(utolsó látogatás időpontja: 2018. november 14.)

[A9] https://www.cytron.io/image/cache/catalog/products/MD-L298N/MD-L298N-800x800.jpg
(utolsó látogatás időpontja: 2018. november 14.)

[A10] https://s1.cdn.autoevolution.com/images/gallery/NISSANLeaf-4310_23.jpg
(utolsó látogatás időpontja: 2018. november 14.)

[A11] https://www.kvh.com/~/media/Images/KVH/Product/Gyros%20IMUs/GeoFog3D/prodGeoFog3DDual.jpg?bc=white
(utolsó látogatás időpontja: 2018. november 14.)

Mellékletek

1. melléklet

NVIDIA

Jetson TX2 System-on-Module
Pascal GPU + ARMv8 + 8GB LPDDR4 + 32GB eMMC + WLAN/BT

Description			Jetson TX2 System-on-Module*
Pascal GPU ◊			
256-core GPU | End-to-end lossless compression | Tile Caching | OpenGL® 4.5 | OpenGL® ES 3.2 | Vulkan® 1.0 | CUDA® 8.0 | GPGPU			
Maximum Operating Frequency			1.12GHz
CPU Complex ‡			
ARMv8 (64-bit) heterogeneous multi-processing (HMP) CPU architecture; two CPU clusters (6 processor cores) connected by a high-performance coherent interconnect fabric. NVIDIA Denver 2 (Dual-Core) Processor: L1 Cache: 128KB L1 instruction cache (I-cache) per core; 64KB L1 data cache (D-cache) per core | L2 Unified Cache: 2MB ARM® Cortex® -A57 MPCore (Quad-Core) Processor: L1 Cache: 48KB L1 instruction cache (I-cache) per core; 32KB L1 data cache (D-cache) per core | L2 Unified Cache: 2MB			
Maximum Operating Frequency per Core NVIDIA Denver 2 ARM Cortex-A57			 2.0GHz 2.0GHz
HD Video & JPEG			
Video Decode(Number of Streams Supported): H.265 (†): Main 10, Main 8 H.265 (†): Main 444 H.264 (†): Baseline, Main, High H.264 (†): MVC Stereo (per view) VP9 (†): Profile 0 (8-bit) and 2 (10 and 12-bit) VP8: All MPEG1/2: Main MPEG4: SP/AP VC1: SP/MP/AP			(2x) 2160p60 | (4x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p60 | (2x) 2160p30 | (3x) 1080p60 | (7x) 1080p30 (2x) 2160p60 | (4x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p60 | 2160p30 | 1080p60 | 1080p30 (2x) 2160p60 | (4x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p60 | (2x) 2160p30 | (4x) 1080p60 | (8x) 1080p30 2160p60 | (2x) 2160p30 | (4x) 1080p60 | (8x) 1080p30 (4x) 1080p60 | (8x) 1080p30 (2x) 1080p60 | (4x) 1080p30
Video Encode (Number of Streams Supported): H.265 H.264: Baseline, Main, High WEBM VP9 WEBM VP8			 2160p60 | (3x) 2160p30 | (4x)1080p60 | (8x) 1080P30 2160p60 | (3x) 2160p30 | (7x) 1080p60 | (14x) 1080p30 2160p30 | (3x) 1080p60 | (7x) 1080p30 2160p30 | (3x) 1080p60 | (6x) 1080p30
JPEG (Decode & Encode)			600 MP/sec
Audio Subsystem			
Industry-standard High Definition Audio (HDA) controller provides a multi-channel audio path to the HDMI interface | 6 x I2S | 4 x DMIC | 2 x DSPK | SPDIF | 2 x I and Q baseband data channels | PDM in/out			
Display Controller Subsystem			
Support for DSI, HDMI, DP and eDP | Two multi-mode eDP/DP/HDMI outputs.			
Captive Panel			
MIPI-DSI (1.5Gbps/lane)		Max Resolution	Support for Single x4 or Dual x4 links | 2560x1600 at 60Hz
eDP 1.4 (HBR2 5.4Gbps)		Max Resolution	3840x2160 at 60Hz
External Display			
HDMI 2.0a/b (6Gbps)		Max Resolution	3840x2160 at 60Hz
DP 1.2a (HBR2 5.4 Gbps)		Max Resolution	3840x2160 at 60Hz
Imaging System			
Dedicated RAW to YUV processing engine process up to 1.4Gpix/s | MIPI CSI 2.0 up to 2.5Gbps (per lane) | Support for x4 and x2 configurations (up to 3 x4-lane or 6 x2-lane cameras)			
Clocks			
System clock: 38.4 MHz | Sleep clock: 32.768 KHz | Dynamic clock scaling and clock source selection			
Boot Sources			
Internal eMMC and USB (recovery mode)			

Description	Jetson TX2 System-on-Module*
Security	
Secure memory with video protection region for protection of intermediate results | Configurable secure DRAM regions for code and data protection | Hardware acceleration for AES 128/192/256 encryption and decryption to be used for secure boot and multimedia Digital Rights Management (DRM) | Hardware acceleration for AES CMAC, SHA-1, SHA-256, SHA-384, and SHA-512 algorithms | 2048-bit RSA HW for PKC boot| HW Random number generator (RNG) SP800-90 | TrustZone technology support for DRAM, peripherals | SE/TSEC with side channel counter-measures for AES | RSA-3096 and ECC-512/521 supported via PKA	
Memory ††	
128-bit DRAM interface | Secure External Memory Access Using TrustZone Technology | System MMU	
Memory Type	4ch x 32-bit LPDDR4
Maximum Memory Bus Frequency (up to)	1866MHz
Memory Capacity	8GB
Storage	
eMMC 5.1 Flash Storage	
Bus Width	8-bit
Maximum Bus Frequency	200MHz (HS400)
Storage Capacity	32GB
Connectivity	
WLAN	
Radio type	IEEE 802.11a/b/g/n/ac dual-band 2x2 MIMO
Maximum transfer rate	866.7Mbps
Bluetooth	
Version level	4.1
Maximum transfer rate	3MB/s
Networking	
10/100/1000 BASE-T Ethernet | IEEE 802.3u Media Access Controller (MAC) | Embedded memory	
Peripheral Interfaces Δ	
XHCI host controller with integrated PHY: (up to) 3 x USB 3.0, 3 x USB 2.0 | USB 3.0 device controller with integrated PHY | 5-lane PCIe: two x1 and one x4 controllers | SATA (1 port) | SD/MMC controller (supporting eMMC 5.1, SD 4.0, SDHOST 4.0 and SDIO 3.0) | 5 x UART | 3 x SPI | 8 x I^2C | 2 x CAN | 4 x I2S: support I^2S, RJM, LJM, PCM, TDM (multi-slot mode) | GPIOs	
Temperature Specification	
Storage Temperature Range	-25C – 80C
Power Requirements ♦	
Power Input	5.5V – 19.6V
Applications	
Intelligent Video Analytics, Drones, Robotics, Industrial automation, Gaming, and more.	

* Refer to the software release feature list for current software support.

◊ GPU Maximum Operating Frequency: 1.3GHz supported in boost mode.
Product is based on a published Khronos Specification and is expected to pass the Khronos Conformance Process. Current conformance status can be found at www.khronos.org/conformance.

‡ CPU Maximum Operating Frequency: 1-4 core = up to 2.0GHz; greater than 4 core = up to 1.4GHz

(†) For max supported number of instances: bitrate not to exceed 15 Mbps per HD stream (i.e., 1080p30), overall effective bitrate is less than or equal to 240 Mbps

†† Dependent on board layout. Refer to Interface Design Guide for layout guidelines.

Δ Refer to the Interface Design Guide and *Parker Series SoC Technical Reference Manual* to determine which peripheral interface options can be simultaneously exposed.

♦ Refer to the Product Design Guide and Thermal Design Guide for evaluating product power and thermal solution requirements

2. melléklet

```
//Gulyás Péter; NK-HAFSIW
//DeepPilot Project
//BNO055 ASSN calibration then orientation, encoder, servo read
//Created: 2018-November-14

#include <Wire.h>
#include <Adafruit_Sensor.h>
#include <Adafruit_BNO055.h>
#include <utility/imumaths.h>
#include <EEPROM.h>

/* Connections
             Connect SCL/LRX - pin-12 to analog 5
             Connect SDA/ATX - pin-11 to analog 4
             Connect VCC - pin-20 to 3-5V DC
             Connect GND - pin-19 to common ground
             D2 - Encoder A
             D8 - Encoder B
*/

// Set the delay between fresh samples --> adjusting datarate
#define BNO055_SAMPLERATE_DELAY_MS (13)   // ~40Hz datarate

#define SrvFb A0
#define ENC_A 2
#define ENC_B 8

//Define variables
float QX = 0, QY = 0, QZ = 0, QW = 0;
float GX = 0, GY = 0, GZ = 0;
float LX = 0, LY = 0, LZ = 0;
String mystring;

struct MotFb
{
  double velo;
  double dist;
  int rpm;
};
typedef struct MotFb Motor_FB;

int AngFbInt;               //---------------------------------
String VeloFeedback;                    //FEEDBACK//
String RpmFeedback;
String DistFeedback;
//String str;
String buf1;
String buf2;
String buf3;            //-------------------------------------

const int interrupt0 = 0; // (PIN 2)  //ENCODER READING//

long sum_pulse_num = 0;
int pulse_num;                    //Calculations//
int rpm = 0;
```

```
// FUNCTION LIST ----------------------------------------------START------

void displaySensorDetails(void);
void displaySensorStatus(void);
void displayCalStatus(void);
void displaySensorOffsets(const adafruit_bno055_offsets_t &calibData);
void Enc_Calc();
void Pulse_counter();
int ServFb();

// FUNCTION LIST ----------------------------------------------END-------

// Identify sensor with special ID
Adafruit_BNO055 bno = Adafruit_BNO055(55);

// Setup function to check calibration.--------------------START-------
// If needed: doing calibration, and saving to EEPROM -----------------

void setup(void)
{
  Serial.begin(115200);
  delay(1000);
  Serial.println("BNO055 Orientation sensor "); Serial.println("");

  // Sensor initialization
  if (!bno.begin())
  {
    // Error message
    Serial.print("BNO055 not detected ... Check your I2C ADDR and wiring!");
    while (1);
  }

  displaySensorDetails();
  delay(1500);

  int eeAddress = 0;
  long bnoID;
  EEPROM.get(eeAddress, bnoID);

  adafruit_bno055_offsets_t calibrationData;
  sensor_t sensor;

  //  Look for unique sensor ID in EEPROM.
  bno.getSensor(&sensor);
  if (bnoID != sensor.sensor_id)
  {
    Serial.println("Calibration data not found in EEPROM. Calibration nedded!");
    delay(500);

    bno.setExtCrystalUse(false);
    sensors_event_t event;
    bno.getEvent(&event);
    Serial.println("Calibrating Sensor: ");
    while (!bno.isFullyCalibrated())
    {
      displayCalStatus();
      Serial.println("");
      delay(BNO055_SAMPLERATE_DELAY_MS);
    }
```

```
    Serial.println("Calibration completed!");
    Serial.println("Calibration Results: ");
    adafruit_bno055_offsets_t newCal;
    bno.getSensorOffsets(newCal);
    displaySensorOffsets(newCal);
    Serial.println("---------------------");
    Serial.println("Storing calibration data to EEPROM.");
    eeAddress = 0;
    bno.getSensor(&sensor);
    bnoID = sensor.sensor_id;
    EEPROM.put(eeAddress, bnoID);
    eeAddress += sizeof(long);
    EEPROM.put(eeAddress, newCal);
    Serial.println("Calibration array stored to EEPROM.");
    delay(500);

  }
  else
  {
    //Serial.println("Calibration data found in EEPROM. Using it as Offset array.");
    eeAddress += sizeof(long);
    EEPROM.get(eeAddress, calibrationData);
    //displaySensorOffsets(calibrationData);
    bno.setSensorOffsets(calibrationData);

  }

  //displaySensorStatus();

  /*Serial.println("Orientation data transfer starts in 3 sec");
  delay(1000);
  Serial.println("Orientation data transfer starts in 2 sec");
  delay(1000);*/
  //Serial.println("Orientation data transfer starts in 1 sec");
  delay(1000);

}

// Setup function to check calibration.---------------------END---------

// Serial orientation data transfer to host --------------START--------
void loop()
{
  // Get new data reading from BNO055 chip
  sensors_event_t event;
  bno.getEvent(&event);

  //Serial data transfer - Start -
  // Timestamp
  //Serial.print(millis());
  //Serial.print(",");
  // Display Heading-Roll-Pitch as orientation
  /*Serial.print(event.orientation.x, 2);
  Serial.print(",");
  Serial.print(event.orientation.y, 2);
  Serial.print(",");
  Serial.print(event.orientation.z, 2);*/
```

```
/* Display Quaternion data */
imu::Quaternion quat = bno.getQuat();
QX = (quat.x());
QY = (quat.y());
QZ = (quat.z());
QW = (quat.w());
mystring = String(QX, 6);
Serial.print(mystring);
Serial.print(" ");
mystring = String(QY, 6);
Serial.print(mystring);
Serial.print(" ");
mystring = String(QZ, 6);
Serial.print(mystring);
Serial.print(" ");
mystring = String(QW, 6);
Serial.print(mystring);
Serial.print(" ");

/* Display the Gyroscope data */
imu::Vector<3> eulergyro = bno.getVector(Adafruit_BNO055::VECTOR_GYROSCOPE);
GX = eulergyro.x();
GY = eulergyro.y();
GZ = eulergyro.z();
mystring = String(GX, 3);
Serial.print(mystring);
Serial.print(" ");
mystring = String(GY, 3);
Serial.print(mystring);
Serial.print(" ");
mystring = String(GZ, 3);
Serial.print(mystring);
Serial.print(" ");

/* Display the Linearaccelerometer data */
imu::Vector<3> eulerlin = bno.getVector(Adafruit_BNO055::VECTOR_LINEARACCEL);
LX = eulerlin.x();
LY = eulerlin.y();
LZ = eulerlin.z();
mystring = String(LX, 3);
Serial.print(mystring);
Serial.print(" ");
mystring = String(LY, 3);
Serial.print(mystring);
Serial.print(" ");
mystring = String(LZ, 3);
Serial.print(mystring);
Serial.print(" ");

/* Display the Encoder and servo data */
Enc_Calc();
AngFbInt = ServFb();
Serial.print(AngFbInt);
Serial.print(" ");
Serial.print(buf1);
Serial.print(" ");
Serial.print(buf2);
Serial.print(" ");
Serial.print(buf3);
```

```
  // New line for next sample
  Serial.println("");
  // Serial data transfer - End -

  // Wait for specified time.
  delay(BNO055_SAMPLERATE_DELAY_MS);

}
// Serial orientation data transfer to host --------------END----------

// Functions used by the program ------------------------START---------

// Display basic information about the sensor -------------------------
void displaySensorDetails(void)
{
  sensor_t sensor;
  bno.getSensor(&sensor);
  Serial.println("------------------------------------");
  Serial.print("Sensor:       "); Serial.println(sensor.name);
  Serial.print("Driver Ver:   "); Serial.println(sensor.version);
  Serial.print("Unique ID:    "); Serial.println(sensor.sensor_id);
  Serial.print("Max Value:    "); Serial.print(sensor.max_value); Serial.println("
xxx");
  Serial.print("Min Value:    "); Serial.print(sensor.min_value); Serial.println("
xxx");
  Serial.print("Resolution:   "); Serial.print(sensor.resolution); Serial.println("
xxx");
  Serial.println("------------------------------------");
  Serial.println("");
  delay(500);
}

// Display sensor status ----------------------------------------------
void displaySensorStatus(void)
{
  // Get the system status values
  uint8_t system_status, self_test_results, system_error;
  system_status = self_test_results = system_error = 0;
  bno.getSystemStatus(&system_status, &self_test_results, &system_error);

  Serial.println("------------------------------------");
  Serial.print("Sys Status: 0x");
  Serial.println(system_status, HEX);
  Serial.print("Self Test:     0x");
  Serial.println(self_test_results, HEX);
  Serial.print("Sys Error:  0x");
  Serial.println(system_error, HEX);
  Serial.println("------------------------------------");
  delay(500);
}

// Display BNO055 calib status ----------------------------------------
void displayCalStatus(void)
{
  // Get calibration status (0 to 3) 0:Not calibrated; 3:Fully caibrated
  uint8_t system, gyro, accel, mag;
  system = gyro = accel = mag = 0;
  bno.getCalibration(&system, &gyro, &accel, &mag);
```

```
  // The data ignored until the system calibration is > 0
  Serial.print("\t");
  if (!system)
  {
    Serial.print("!");
  }
  //Display the individual values
  Serial.print("Sys:");
  Serial.print(system, DEC);
  Serial.print(" Gyro:");
  Serial.print(gyro, DEC);
  Serial.print(" Acc:");
  Serial.print(accel, DEC);
  Serial.print(" Mag:");
  Serial.print(mag, DEC);
}

// Display the raw calibration offset and radius data ----------------
void displaySensorOffsets(const adafruit_bno055_offsets_t &calibData)
{
  Serial.print("Accelerometer: ");
  Serial.print(calibData.accel_offset_x); Serial.print(" ");
  Serial.print(calibData.accel_offset_y); Serial.print(" ");
  Serial.print(calibData.accel_offset_z); Serial.print(" ");

  Serial.print("\nGyro: ");
  Serial.print(calibData.gyro_offset_x); Serial.print(" ");
  Serial.print(calibData.gyro_offset_y); Serial.print(" ");
  Serial.print(calibData.gyro_offset_z); Serial.print(" ");

  Serial.print("\nMag: ");
  Serial.print(calibData.mag_offset_x); Serial.print(" ");
  Serial.print(calibData.mag_offset_y); Serial.print(" ");
  Serial.print(calibData.mag_offset_z); Serial.print(" ");

  Serial.print("\nAccel Radius: ");
  Serial.print(calibData.accel_radius);

  Serial.print("\nMag Radius: ");
  Serial.print(calibData.mag_radius);
}

// Calculating dinamic values from encoder reading -----------------
void Enc_Calc(){
  Motor_FB FB_1;                // 48 Encoder pulse = 1 round
  rpm = pulse_num * 60 / 24;    // RPM calculating ratio for 1 ch
  //Serial.print(rpm);
  //Serial.print("\t");
  FB_1.rpm = rpm;
  FB_1.velo = rpm / 9.5492965964254;
  //Serial.print("Speed = ");
  //Serial.print(FB_1.velo, 4);
  //Serial.print(" rad/s");
  //Serial.print("\t");
  sum_pulse_num = sum_pulse_num + pulse_num;
  FB_1.dist = sum_pulse_num / 1185.36 * 0.282743;
  //Serial.print("Distance: ");
  //Serial.print(FB_1.dist, 4);
  //Serial.println(" m");
```

```
  //Serial.print("\t");
  VeloFeedback = String(FB_1.velo);
  RpmFeedback = String(FB_1.rpm);
  DistFeedback = String(FB_1.dist);
  buf1 = VeloFeedback;
  buf2 = RpmFeedback;
  buf3 = DistFeedback;
  pulse_num = 0x00;
}

// Counting encoder pulses - Only falling edge is considerd -----------------
void Pulse_counter()
{
 if(digitalRead(ENC_B) == HIGH) pulse_num++;
 else pulse_num--;
}

// Servo feedback and mapping, from analog readig -----------------
int ServFb(){
  int val = analogRead(SrvFb);              //Analog value reading to know the position
  //Serial.print(val);
  int valMap = map(val, 132, 413, -10, 10); //Remap the analog interval to get the
angle of the car
  return valMap;
}

// Functions used by the program ------------------END-----------------
```

3. melléklet

sensor_msgs/Imu Message

File: sensor_msgs/Imu.msg

Raw Message Definition

```
# This is a message to hold data from an IMU (Inertial Measurement Unit)
#
# Accelerations should be in m/s^2 (not in g's), and rotational velocity should be in rad/sec
#
# If the covariance of the measurement is known, it should be filled in (if all you know is the
# variance of each measurement, e.g. from the datasheet, just put those along the diagonal)
# A covariance matrix of all zeros will be interpreted as "covariance unknown", and to use the
# data a covariance will have to be assumed or gotten from some other source
#
# If you have no estimate for one of the data elements (e.g. your IMU doesn't produce an orientation
# estimate), please set element 0 of the associated covariance matrix to -1
# If you are interpreting this message, please check for a value of -1 in the first element of each
# covariance matrix, and disregard the associated estimate.

Header header

geometry_msgs/Quaternion orientation
float64[9] orientation_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance # Row major about x, y, z axes

geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance # Row major x, y z
```

Compact Message Definition

```
std_msgs/Header header
geometry_msgs/Quaternion orientation
float64[9] orientation_covariance
geometry_msgs/Vector3 angular_velocity
float64[9] angular_velocity_covariance
geometry_msgs/Vector3 linear_acceleration
float64[9] linear_acceleration_covariance
```

autogenerated on Fri, 09 Nov 2018 03:18:54

4. melléklet

```
#include "ros/ros.h"
#include <stdio.h>
#include "SerialPort.h"
#include <algorithm>
#include <stdlib.h>
#include <math.h>
#include <iostream>
#include <string>
#include <unistd.h>
#include <cstdlib>
#include <sensor_msgs/JointState.h>
#include <sensor_msgs/Imu.h>
#include <sstream>
#include <iomanip>

#define _USE_MATH_DEFINES

const double Conv = 1.000000;
float cov_base = 0.00;

int main(int argc, char **argv)
{
	ros::init(argc, argv, "NANO_reader");
	ros::NodeHandle n;
	ros::Publisher joint = n.advertise <sensor_msgs::JointState> ("joint_states",
1000);
	ros::Publisher assn = n.advertise <sensor_msgs::Imu> ("assn", 1000);
	ros::Rate loop_rate(50);

	// ROS message init, and message fill up.
	sensor_msgs::Imu assn_pub;
	assn_pub.header.frame_id = "imu";

	sensor_msgs::JointState joint_pub;

	joint_pub.name.resize(3);
	joint_pub.position.resize(3);
	joint_pub.velocity.resize(3);

	joint_pub.name[0] = "front_axis_to_left_front_steer";
	joint_pub.name[1] = "front_axis_to_right_front_steer";
	joint_pub.name[2] = "rear_axis_to_left_rear_wheel";

	try
	{
		//Open serial port with defined parameters
		ROS_INFO("Opening serial connection.");
		SerialPort serial_port ("/dev/ttyUSB0");
		serial_port.Open(SerialPort::BAUD_115200,
					SerialPort::CHAR_SIZE_8,
					SerialPort::PARITY_NONE,
					SerialPort::STOP_BITS_1,
					SerialPort::FLOW_CONTROL_HARD);
```

```
            // Enter the main loop
            while (ros::ok())
            {
                  // Wait until the data is available
                  if(serial_port.IsDataAvailable())
                {
                        // Define string to read in data
                        std::string ss;

                        // Reading in from serial port
                        ss = serial_port.ReadLine();
                        std::istringstream iss(ss);

                        // Create a vector of strings separated by "whitespace"
                        std::vector<std::string> results(std::istream_iterator
                        <std::string>{iss},
                        std::istream_iterator<std::string>());

                        //Print all IMU data to console as a string
                        float QX = Conv*atof(results[0].c_str());
                        //std::cout << QX  << '\t';
                        float QY = Conv*atof(results[1].c_str());
                        //std::cout << QY  << '\t';
                        float QZ = Conv*atof(results[2].c_str());
                        //std::cout << QZ  << '\t';
                        float QW = Conv*atof(results[3].c_str());
                        //std::cout << QW  << '\t';
                        float GX = Conv*atof(results[4].c_str());
                        float GY = Conv*atof(results[5].c_str());
                        float GZ = Conv*atof(results[6].c_str());
                        float LX = Conv*atof(results[7].c_str());
                        float LY = Conv*atof(results[8].c_str());
                        float LZ = Conv*atof(results[9].c_str());
                        int angWheel = Conv*atoi(results[10].c_str());
                        float velo = Conv*atof(results[11].c_str());
                        int rpm = Conv*atoi(results[12].c_str());
                        float dist = Conv*atof(results[13].c_str());

                        assn_pub.header.stamp = ros::Time::now();
                        joint_pub.header.stamp = ros::Time::now();

// Orientation-Quaterninon data passed to message   // IMU MESSAGE START
                        assn_pub.orientation.x = QX;
                        assn_pub.orientation.y = QY;
                        assn_pub.orientation.z = QZ;
                        assn_pub.orientation.w = QW;

                        // Covariance matrix filled up with zeros
                        std::fill( assn_pub.orientation_covariance.begin(),
                        assn_pub.orientation_covariance.end(), cov_base );

                        // 3 axis Gyroscope data passed to message
                        assn_pub.angular_velocity.x = GX;
                        assn_pub.angular_velocity.y = GY;
                        assn_pub.angular_velocity.z = GZ;
```

```
                        // Covariance matrix filled up with zeros
                        std::fill( assn_pub.angular_velocity_covariance.begin(),
                        assn_pub.angular_velocity_covariance.end(),  cov_base );

                        // 3 axis Linear acceleration data passed to message
                        assn_pub.linear_acceleration.x = LX;
                        assn_pub.linear_acceleration.y = LY;
                        assn_pub.linear_acceleration.z = LZ;

                        // Covariance matrix filled up with zeros
                        std::fill( assn_pub.linear_acceleration_covariance
                        .begin(),assn_pub.linear_acceleration_covariance.
                        end(),  cov_base );

// Fill the JointState message  IMU MESSAGE END   JOINTSTATE MESSAGE START
                        joint_pub.position[0] = (angWheel * M_PI) / 180;
                        joint_pub.position[1] = (angWheel * M_PI) / 180;
                        joint_pub.position[2] = dist;
                        joint_pub.velocity[0] = (rpm / 9.55);
                        joint_pub.velocity[1] = (rpm / 9.55);
                        joint_pub.velocity[2] = (rpm / 9.55);


// Publish messages                                    JOINTSTATE MESSAGE END
                        assn.publish(assn_pub);
                        joint.publish(joint_pub);

                  }

                  ros::spinOnce();

                  loop_rate.sleep();

            }

            // Close connection
            if(serial_port.IsOpen())
            {
                  ROS_INFO("Closing serial connection.");
                  serial_port.Close();
            }

            ROS_INFO("Serial node closed.");
            return 0;
      }
      catch (SerialPort::OpenFailed exception)
      {
            ROS_ERROR("Failed to open the port");
      }

      catch (SerialPort::AlreadyOpen exception)
      {
            ROS_ERROR("Port is already open");
}
}
```

5. melléklet

sensor_msgs/JointState Message

File: sensor_msgs/JointState.msg

Raw Message Definition

```
# This is a message that holds data to describe the state of a set of torque controlled joints.
#
# The state of each joint (revolute or prismatic) is defined by:
#  * the position of the joint (rad or m),
#  * the velocity of the joint (rad/s or m/s) and
#  * the effort that is applied in the joint (Nm or N).
#
# Each joint is uniquely identified by its name
# The header specifies the time at which the joint states were recorded. All the joint states
# in one message have to be recorded at the same time.
#
# This message consists of a multiple arrays, one for each part of the joint state.
# The goal is to make each of the fields optional. When e.g. your joints have no
# effort associated with them, you can leave the effort array empty.
#
# All arrays in this message should have the same size, or be empty.
# This is the only way to uniquely associate the joint name with the correct
# states.


Header header

string[] name
float64[] position
float64[] velocity
float64[] effort
```

Compact Message Definition

```
std_msgs/Header header
string[] name
float64[] position
float64[] velocity
float64[] effort
```

autogenerated on Fri, 09 Nov 2018 03:18:54

6. melléklet

```
#include <ros/ros.h>
#include <std_msgs/String.h>
#include <sensor_msgs/JointState.h>
#include <nav_msgs/Odometry.h>
#include <std_msgs/Float64.h>
#include <szelectricity_common/VehicleParameters.hpp>
#include <szelectricity_common/MathGeometry.hpp>

const std::string DEFAULT_BASE_FRAME = "base_footprint";
const std::string DEFAULT_ODOM_FRAME = "odom";

struct joints {
  double p1;
  double p2;
  double velo1;
} olvasas;

sensor_msgs::JointState msg_joint;

nav_msgs::Odometry current_state;         // Current odometry state (pose, twist,
covariance)
double theta = 0.0;                       // Current yaw related to COG
double dtheta = 0.0;                      // Change in theta
double dx = 0.0;                          // Change in x
double dy = 0.0;                          // Change in y

ros::Time prev_time, curr_time;

void jointCallback(const sensor_msgs::JointState::ConstPtr& msg)
{
  //std::cout << *msg << std::endl;
  msg_joint.position = msg->position;
  msg_joint.velocity = msg->velocity;
  //std::cout << msg_joint.position[0] << std::endl;
  //std::cout << msg_joint.velocity[2] << std::endl;
  olvasas.p1 = msg_joint.position[0];
  //std::cout << olvasas.p1 << std::endl;
  olvasas.p2 = msg_joint.position[1];
  olvasas.velo1 = msg_joint.velocity[2];

}

void UpdateStep(double dt, const ros::Time stamp)
    {

        current_state.header.stamp = stamp;
        // Update the pose estimation
        theta = theta+dtheta*dt;
        current_state.twist.twist.linear.x = dx;
        current_state.twist.twist.linear.y = dy;
        // Angular speed is defined as RPY, only YAW is considered
        current_state.twist.twist.angular.z = dtheta*dt;
        // Pose update
        current_state.pose.pose.position.x += dx*dt;
        current_state.pose.pose.position.y += dy*dt;
```

```
        current_state.pose.pose.orientation = szenergy::YawToQuaternion(theta);
    }

int main(int argc, char **argv)
{

    ros::init(argc, argv, "joint_listener");
    ros::NodeHandle n;
    ros::Subscriber sub = n.subscribe("/joint_states", 1000, jointCallback);
    ros::Publisher odom = n.advertise <nav_msgs::Odometry> ("odometry", 1000);
    ros::Rate loop_rate(20);

    szenergy::VehicleParameters params("modelcar",
      0.045,
      0.267,
      0.188,
      0.147);

    current_state.pose.pose.position.z = 0.0;
    current_state.twist.twist.linear.z = 0.0;
    current_state.pose.pose.orientation.w = 1.0;

    current_state.twist.twist.angular.x = 0.0;
    current_state.twist.twist.angular.y = 0.0;

    current_state.header.frame_id = "odom";
    current_state.child_frame_id = "base_link";

    while(ros::ok())

    {
        olvasas.p1;
        olvasas.p2;
        olvasas.velo1;
//std::cout << olvasas.p1 << '\t' << olvasas.p2 << '\t' << olvasas.velo1 << std::endl;

        ros::Duration dt;
        const ros::Time stamp;
        curr_time = ros::Time::now();

        dt = curr_time - prev_time;

        double linvel = olvasas.velo1*params.wheelradius;
        //std::cout << linvel << '\t';
        dtheta = linvel/(params.wheelbase)*tan(olvasas.p1);
        //std::cout << dtheta << '\t';
        dx = linvel*cos(theta);
        //std::cout << dx << '\t';
        dy = linvel*sin(theta);
        //std::cout << dt << std::endl;
        UpdateStep(dt.toSec(), stamp);

        odom.publish(current_state);

        prev_time = curr_time;
        ros::spinOnce();
        loop_rate.sleep();
    }
}
```

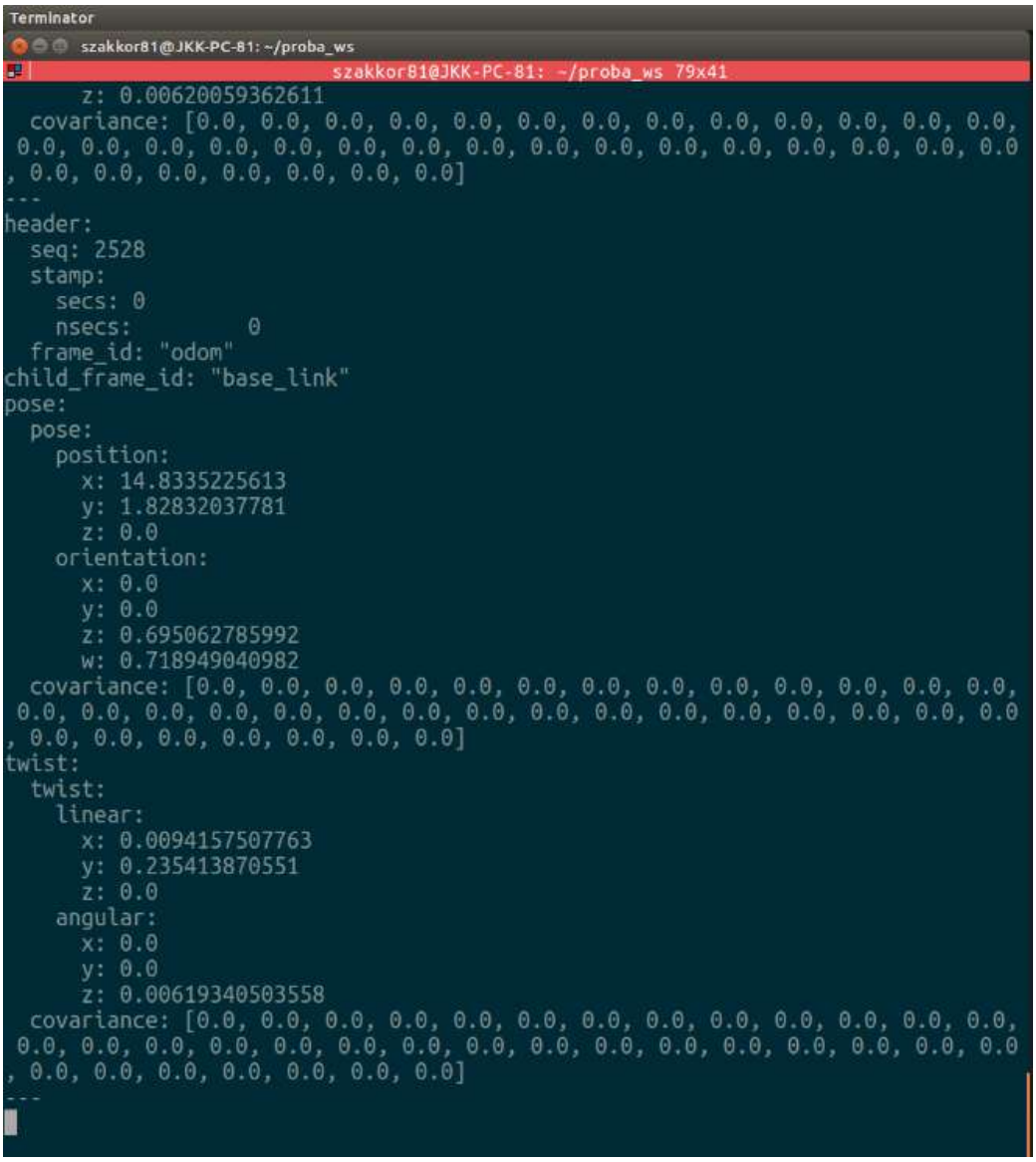
7. melléklet

```
#include "ros/ros.h"
#include <stdio.h>
#include <algorithm>
#include <stdlib.h>
#include <iostream>
#include <string>
#include <unistd.h>
#include <cstdlib>
#include <termios.h>
#include <fcntl.h>
#include "std_msgs/String.h"
#include <sstream>
#include <iomanip>

int open_port(void);

// Declare file descriptor for the port.
int fd;

int len = 10;
std::string send ("");
float angRef = 1.80;
float torqueRef = 0.30;
int wsuc = 0;

int main(int argc, char **argv)
{
	ros::init(argc, argv, "UNO_reader");
	ros::NodeHandle n;
	ros::Publisher uno = n.advertise<std_msgs::String> ("uno", 1000);
	// Under 40 Hz : system has much latency --> SET >= 40
	ros::Rate loop_rate(40);

	try
	{
		// Open serial port with defined parameters
		ROS_INFO("Opening serial connection.");
		open_port();
		//std::cout << fd << std::endl;
		ros::Duration(2).sleep();

		// Enter the main loop
		while (ros::ok())
		{
			// -------WRITING SEQUENCE ----START--------

			std::stringstream stream;
			stream << std::fixed << std::setprecision(2) << torqueRef << "," <<
			angRef << "a";

			send = stream.str();
			wsuc = write(fd, send.c_str(), len);
			//std::cout << send << std::endl;
			// --------WRITING SEQUENCE -------END-------
			ros::spinOnce();
			loop_rate.sleep();
		}
```

```
        }
        catch (std::exception)
        {
                ROS_ERROR("Failed to open the port");
        }

close( fd );

}

// ------------PORT OPENING FUNCTION ------------------------
int open_port()
{
        // Open port
        fd = open("/dev/ttyACM0", O_RDWR | O_NOCTTY | O_NDELAY);
        if (fd == -1)
    {
                // ERROR: Could not open port!
                std::cout << "open_port: Unable to open /dev/ttyACM0" << std::endl;
    }
        else
        {
        fcntl(fd, F_SETFL, FNDELAY);
                // Port opened successfully!
                std::cout << "In Open port fd = " << fd << std::endl;
        }
        // Read the configureation of the port.
        struct termios options_2;
        tcgetattr( fd, &options_2 );
        // Set I/O speed of the port.
        cfsetispeed( &options_2, B115200 );
        cfsetospeed( &options_2, B115200 );

        // Enable the receiver and set local mode.
        options_2.c_cflag |= ( CLOCAL | CREAD );
        // Set the new options for the port.
        tcsetattr(fd, TCSAFLUSH, &options_2);

        // Set the Charactor size: Mask the character size bits
        options_2.c_cflag &= ~CSIZE;
        // Select 8 data bits
        options_2.c_cflag |= CS8;

        // Set parity - No Parity (8N1)
        options_2.c_cflag &= ~PARENB;
        options_2.c_cflag &= ~CSTOPB;
        options_2.c_cflag &= ~CSIZE;
        options_2.c_cflag |= CS8;

        // Disable Software Flow control
        options_2.c_iflag &= ~(IXON | IXOFF | IXANY);

        // Chose raw (not processed) output
        options_2.c_oflag &= ~OPOST;
        if ( tcsetattr( fd, TCSANOW, &options_2 ) == -1 ){
                // ERROR: set tcsetattr not successfull!
                std::cout << "Error with setting attributions." << std::endl;
        }
```

```
        else{
                // tcsetattr OK!
                std::cout << "Attributes settings succeeded" << std::endl;
        }
        usleep(1500);
        tcflush(fd,TCIOFLUSH);          // Flush I/O queue.

        return(fd);
}
// ----PORT OPENING FUNCTION -------------END------------
```

8. melléklet

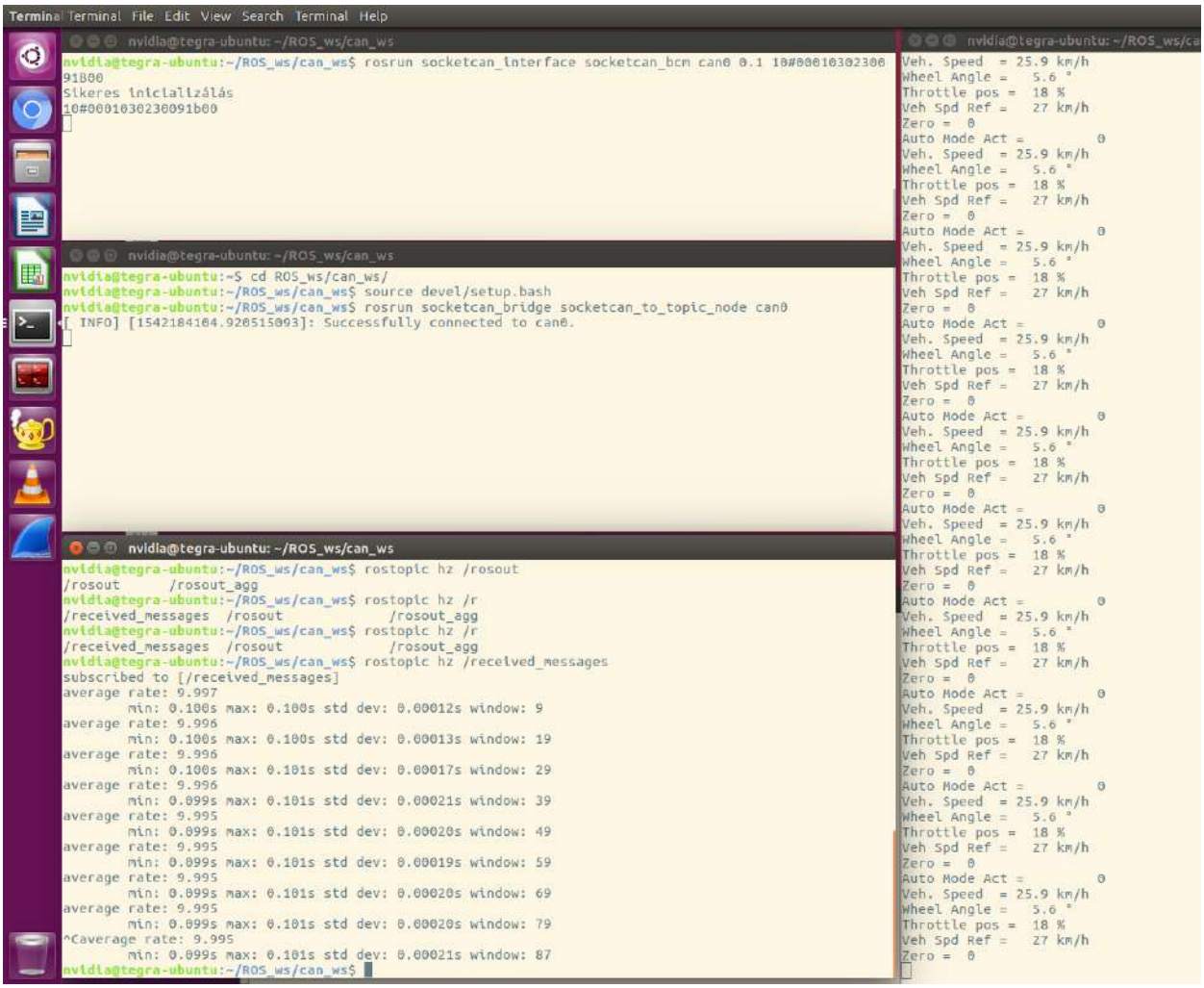

Terminator
szakkor81@JKK-PC-81: ~/proba_ws
szakkor81@JKK-PC-81: ~/proba_ws 79x41
z: 0.00620059362611
covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---
header:
seq: 2528
stamp:
secs: 0
nsecs: 0
frame_id: "odom"
child_frame_id: "base_link"
pose:
pose:
position:
x: 14.8335225613
y: 1.82832037781
z: 0.0
orientation:
x: 0.0
y: 0.0
z: 0.695062785992
w: 0.718949040982
covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
twist:
twist:
linear:
x: 0.0094157507763
y: 0.235413870551
z: 0.0
angular:
x: 0.0
y: 0.0
z: 0.00619340503558
covariance: [0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
---

9. melléklet

```
#!/usr/bin/python
# -*- coding: utf-8 -*-
"""
Created on Mon Oct 4 11:27:10 2018
@author: Potex (github.com/Potex)
Python version: 3.4.2
Control script for Szenergy-model.
"""
# Raspberry Pi script which controls the motor and servo
# also creats log files to save sensor data

import serial
import time
import threading
import csv

t = time.localtime()
timestamp = time.strftime('%Y%m%d%H%M%S', t)
ser=serial.Serial("/dev/ttyACM0", 115200)
ser2=serial.Serial("/dev/ttyUSB0", 115200)
time.sleep(1)

sens_reading = True
log = []

with open('log_sensor'+timestamp+'.csv', 'w') as csvfile:
    Sens_log = csv.writer(csvfile, delimiter=' ', quotechar='|',
quoting=csv.QUOTE_MINIMAL)
    Sens_log.writerow(['Sensor values form Encoder, Servo and ASSN'])
print('Sensor CSV Done. Start logging')

def sensor_read():
    while sens_reading:
        log1 = ser2.readline()
        data = str(log1)
        data2 = data[2:-5]
        with open('log_sensor'+timestamp+'.csv', 'a') as csvfile:
            Sens_log = csv.writer(csvfile, delimiter=' ', quotechar='|',
quoting=csv.QUOTE_MINIMAL)
            Sens_log.writerow(data2)

s = threading.Thread(target=sensor_read)
s.start()
time.sleep(4)

nullStr  =  "1.0,1.0a"
forwStr  =  "0.0,0.95a"
rightStr =  "0.0,2.0a"
right2Str =  "0.0,1.88a"
leftStr =  "0.2,0.0a"
forw2Str = "0.0,0.93a"

with open('log_jarmumodell'+timestamp+'.csv', 'w') as csvfile:
    Modell_log = csv.writer(csvfile, delimiter=' ', quotechar='|',
quoting=csv.QUOTE_MINIMAL)
    Modell_log.writerow(['Systime between performed tasks'])
print('Modell CSV Done. Start logging')
```

```
##3219.429703301   Time before serial_write
##3219.430038246   Time after serial_write, before sleep(1)
##  serial_write = 0.334945 ms
##3220.431092828   Time after sleep(1), before serial_write
##  sleep(1)     = 1001.054582 ms
##3220.431360638   Time after serial_write, before sleep(2)
##  serial_write = 0.267810 ms
##3222.433491833   Time after sleep(2), before serial_write
##  sleep(2)     = 2002.125453 ms
##3222.433730789   Time after serial_write, before sleep(11)
##  serial_write = 0.238956 ms

##-------------1. Lap ---------------
#log.append(time.perf_counter())
ser.write(nullStr.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(1.5)
#log.append(time.perf_counter())
ser.write(forw2Str.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(3.5)
#log.append(time.perf_counter())
ser.write(rightStr.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(10.2)
#log.append(time.perf_counter())
ser.write(forwStr.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(5.6)
#log.append(time.perf_counter())
ser.write(right2Str.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(10.2)
#log.append(time.perf_counter())
ser.write(forwStr.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(1.65)
#log.append(time.perf_counter())
ser.write(nullStr.encode('ascii'))
#log.append(time.perf_counter())
time.sleep(2)
#log.append(time.perf_counter())
##-------------1. Lap ----END--------
sens_reading = False
time.sleep(1)
##---------Write to log file---------
with open('log_jarmumodell'+timestamp+'.csv', 'a') as csvfile:
    Modell_log = csv.writer(csvfile, delimiter=' ', quotechar='|',
quoting=csv.QUOTE_MINIMAL)
    Modell_log.writerow(log)
print('Log written.')
##------Write to log file---END------

print('Sensros logging thread stopped.')
ser.close()
ser2.close()
print("Test round comlpeted. Serial connection closed, motor stopped, steering set to
middle position. ")
```

10. melléklet



11. melléklet

```
#include <iostream>
#include <memory>
#include <unordered_set>

#include <boost/exception/diagnostic_information.hpp>
#include <class_loader/class_loader.hpp>
#include <socketcan_interface/socketcan.h>

using namespace can;

#include <iostream>

int rtflag=0;
int statecommand=0;
int act_state=0;
int auto_mode_active=0;
int zero =0;
int throttle_pos=0;
int act_gear=0;
int temp;

float veh_speed=0.0;
float wheel_ang=0.0;
float veh_spd_ref=0.0;
float serv_comm=0.0;
float serv_sens=0.0;
float steer_ang=0.0;
float nissan_veh_speed=0.0;
float WSpd_FR=0.0;
float WSpd_FL=0.0;
float WSpd_RR=0.0;
float WSpd_RL=0.0;

void print_error(const State & s);
void print_frame(const Frame &f){

    switch((int) f.id){
      case 0x2 :
            temp = f.data[0];
            steer_ang = temp*256;
            temp = f.data[1];
            steer_ang += temp;
            steer_ang = steer_ang/10/4684.8;
            std::cout << "Steering angle = " << steer_ang << "°" << std::endl;
            break;

      case 0x11 :
            rtflag = f.data[0];
            statecommand = f.data[1];
            std::cout << "RT flag = " << "\t" << rtflag << std::endl;
            std::cout << "State command = " << "\t" << statecommand << std::endl;
            break;

      case 0x12 :
            act_state = f.data[1];
            std::cout << "Actual state = " << "\t" << act_state << std::endl;
            break;
```

```
case 0x176 :
       temp = f.data[0];
       nissan_veh_speed = temp*256;
       temp = f.data[1];
       nissan_veh_speed += temp;
       nissan_veh_speed = nissan_veh_speed/8.570637;
       std::cout << "N. Veh. spd = " << nissan_veh_speed << " km/h" << std::endl;
       break;

case 0x180 :
       throttle_pos =  0,5 * (f.data[5]);
       std::cout << "Th. pos = " << "\t" << throttle_pos << " %" <<  std::endl;
       break;

case 0x284 :
       temp = f.data[0];
       WSpd_FR = temp*16777216;
       //std::cout << serv_comm << std::endl;
       temp = f.data[1];
       WSpd_FR += (temp*65536);
       temp = f.data[2];
       WSpd_FR += (temp*256);
       temp = f.data[3];
       WSpd_FR += temp;
       temp = f.data[4];
       WSpd_FL = temp*16777216;
       temp = f.data[5];
       WSpd_FL += (temp*65536);
       temp = f.data[6];
       WSpd_FL += (temp*256);
       temp = f.data[7];
       WSpd_FL += temp;
       std::cout << "WSpeed Fr-Right = " << "\t" << WSpd_FR << std::endl;
       std::cout << "WSpeed Fr-Left = " << "\t" << WSpd_FL << std::endl;
       break;

case 0x285 :
       temp = f.data[0];
       WSpd_RR = temp*16777216;
       //std::cout << serv_comm << std::endl;
       temp = f.data[1];
       WSpd_RR += (temp*65536);
       temp = f.data[2];
       WSpd_RR += (temp*256);
       temp = f.data[3];
       WSpd_RR += temp;
       temp = f.data[4];
       WSpd_RL = temp*16777216;
       temp = f.data[5];
       WSpd_RL += (temp*65536);
       temp = f.data[6];
       WSpd_RL += (temp*256);
       temp = f.data[7];
       WSpd_RL += temp;
       std::cout << "WSpeed Re-Right = " << "\t" << WSpd_RR << std::endl;
       std::cout << "WSpeed Re-Left = " << "\t" << WSpd_RL << std::endl;
       break;
```

```
        case 0x421 :
              act_gear = f.data[0];
              std::cout << "Actual Gear = " << "\t" << act_gear << std::endl;
              break;
        default:
              //std::cout << "Not defined CAN message recieved!" << std::endl;
              ;
        }
}
std::shared_ptr<class_loader::ClassLoader> g_loader;
DriverInterfaceSharedPtr g_driver;

void print_error(const State & s){
    std::string err;
    g_driver->translateError(s.internal_error,err);
    std::cout << "ERROR: state=" << s.driver_state << " internal_error=" <<
s.internal_error << "('" << err << "') asio: " << s.error_code << std::endl;
}

int main(int argc, char *argv[]){

    if(argc != 2 && argc != 4){
        std::cout << "usage: "<< argv[0] << " DEVICE [PLUGIN_PATH PLUGIN_NAME]" <<
std::endl;
        return 1;
    }

    if(argc == 4 ){
        try
        {
            g_loader = std::make_shared<class_loader::ClassLoader>(argv[2]);
            g_driver = g_loader->createUniqueInstance<DriverInterface>(argv[3]);
        }

        catch(std::exception& ex)
        {
            std::cerr << boost::diagnostic_information(ex) << std::endl;;
            return 1;
        }
    }else{
        g_driver = std::make_shared<SocketCANInterface>();
    }

    FrameListenerConstSharedPtr frame_printer = g_driver-
>createMsgListener(print_frame);
    StateListenerConstSharedPtr error_printer = g_driver-
>createStateListener(print_error);

    if(!g_driver->init(argv[1], false)){
        print_error(g_driver->getState());
        return 1;
    }

    g_driver->run();
    g_driver->shutdown();
    g_driver.reset();
    g_loader.reset();

    return 0;}
```