Sávdetektáló algoritmus teljesítményének összehasonlítása C++ és Python nyelven

Doba Dániel*. Fehér Árpád**
Aradi Szilárd***

**Budapesti Műszaki és Gazdaságtudományi Egyetem, MSc hallgató,*
(e-mail: doba.daniel@outlook.com)
*** Budapesti Műszaki és Gazdaságtudományi Egyetem, egyetemi adjunktus,*
(e-mail: feher.arpad@mail.bme.hu)
**** Budapesti Műszaki és Gazdaságtudományi Egyetem, egyetemi adjunktus,*
(e-mail: aradi.szilard@mail.bme.hu)

Abstract: A Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedés- és Járműirányítási Tanszékén futó hallgatói projekt keretében egy elektromos hajtású kísérleti járművet fejlesztünk. A jármű beavatkozó rendszerének elkészülte után lehetőséget biztosít különböző autonóm járműves tudományos kísérletek elvégzésére. Első feladatként egy kamera alapú sávtartó rendszer fejlesztését tűztük ki célul, ami kapcsán – az ADAS rendszerekben használatos kamerákhoz hasonlóan – egy költséghatékony sávdetektáló kamerát fejlesztünk. A célunk egy komplett szenzoregység elkészítése, amit felszerelve a járműre magasszintű sávinformációkat tud szolgáltatni. A Raspberry Pi hardveren történt a fejlesztés, ami később két irányba ágazott el. Elkészült egy Python nyelven írt szoftververzió, amiből számos hasznos tapasztalatot gyűjtve az algoritmus tisztán C++ kódban történő implementálására helyeződött a hangsúly. Ugyanazon probléma megoldása két különböző programozási nyelven rendkívül jó összehasonlítási alapot biztosított.

1. BEVEZETÉS ÉS MOTIVÁCIÓ

Napjainkban tisztán látszik, hogy a fejlett vezetéstámogató és aktív biztonsági rendszerekkel szerelt gépjárművektől az önvezető autós fejlesztésekig, egyre gyakoribb az olcsó CMOS-rendszerű kamera alapú megoldások alkalmazása. Ez a trend a korszerű ADAS (Advance Driver Assistance Systems) rendszerek elterjedésével kezdődött, amik eleinte csak a csúcskategóriás autókban jelentek meg, de a hatályba lépő jogszabályváltozások miatt gyorsan megjelentek a középkategóriás és a belépő szintű járművekben is.

Az autógyárók több különböző helyre szerelnek kamerákat (Dabral *és mtsai.*, 2014). A leggyakoribb az előre néző kamera, melyet rendszerint a szélvédő felső részére a visszapillantó tükör környékére helyeznek el. Ez a kamera alapszenzora lehet például egy táblafelismerő vagy egy sávtartó ADAS rendszernek, de számos hasznos információ kinyerhető belőle egyéb asszisztens vagy biztonsági rendszer részére. Ezek a kamerák információkat szolgáltathatnak például a látószögön belül lévő járművek kategóriájáról, távolságáról és sebességéről. Mindezek mellett a gyalogosdetektálás fontos szenzora is lehet. A legtöbb szériában kiadott rendszer természetesen nem csak egy szenzorra épül, hanem legtöbbször a hasznos információ egyéb szenzorok (pl.: radar, ultrahang) bevonásával, szenzorfúzió után kerül meghatározásra.

A Budapesti Műszaki és Gazdaságtudományi Egyetem Közlekedés- és Járműirányítási Tanszékén futó hallgatói projekt keretében egy belsőégésű motorral szerelt gokartot alakítottunk át teljesen elektromossá és szereltük fel a szükséges beavatkozó aktuátorokkal és feldolgozó egységekkel. A kormányzást egy léptető motor felszerelésével, míg a fékezést oldalanként egy-egy hidraulikus kerékpárfékkel és modellszervókkal oldottuk meg. Így kaptunk egy járműplatformot, ami lehetőséget biztosít különböző autonóm járműves témájú tudományos kísérletek elvégzésére. A járműre került egy erős ipari PC, aminek központi adatfeldolgozó és irányító szerepet szántunk.

Miután a gokart vezetéknélküli manuális irányítása és a távoli biztonsági vészleállító rendszer elkészült, első feladatként egy sávtartó asszisztens funkciót terveztünk elkészíteni. A feladat során a cél, hogy a gokart tisztán kamera alapon autonóm módon kövessen egy tesztpályán kialakított sávot. A feladat első lépéseként a sávok detektálását kell megoldani. Erre a feladatra kerestünk egy hardvert, ami – az ADAS rendszerekben használatos kamerákhoz hasonlóan - önmagában képes megoldani a detekciót és már magas szintű sávinformációkat tudjon szolgáltatni az ipari PC felé. A feladat során egy igen erőforrásigényes komplex képfeldolgozási problémát kell megoldani. A költséghatékonyság is fontos szempont volt, ezért a hagyományos beágyazott rendszerek nem jöhettek szóba. A követelményeket figyelembe véve a jelenleg igen népszerű és nagy támogatással rendelkező Raspberry Pi eszközre esett a választásunk, aminek gyári tartozéka egy CMOS-rendszerű kamera.

A Raspberry PI hardveren egy Debian Linux alapú operációs rendszeren történt a fejlesztés, ami később több irányba ágazott el. Elsőként elkészült a sávdetektáló algoritmus Python programnyelven írt változata, ami a nyelv tervezési filozófiájából adódóan rendkívül jó tanulási lehetőséget biztosított, viszont hamar megtapasztaltuk a korlátait. Ezután az algoritmus tisztán C++ kódban történő implementálására

helyeződött a hangsúly. Ugyanazon probléma megoldása két különböző programozási nyelven rendkívül jó összehasonlítási alapot biztosított.

A cikk 2. fejezetében bemutatásra kerülnek a klasszikus sávdetektáló algoritmusok. Ezután a 3. fejezetben ismertetve lesz az alkalmazott hardver. A 4. fejezetben a szoftver felépítéséről és a szoftveres megvalósításról lesz szó. Az 5. fejezetben ismertetésre kerülnek a teszteredmények.

2. SÁVDETEKTÁLÁS KLASSZIKUS MÓDSZEREKKEL

A sávdetektálás célja a járműpozícióját meghatározni a forgalmi sávon belül egy előretekintő kamera képe alapján, valamint a forgalmi sáv paramétereit (szélesség, görbület). A sávdetektálás során az elsődleges markerek az út felületén felfestett vonalak, de az út széle mentén fellépő textúra változás is kiinduló alapja lehet az algoritmusoknak.

A sávdetektáló algoritmusok széles szakirodalommal rendelkeznek (M Kumar és Simon, 2015), de azok teljesítménye a probléma komplexitásából adódóan még nem elégséges, hogy minden körülmények között megbízhatóak legyenek. Ezek részben a számítástechnika fejlődésével orvosolhatóak, de az algoritmusok robusztusságára a mérnöki tudománynak kell választ adnia.

A sávdetektáló algoritmusok általában 2 fő részre bonthatóak: az első az előfeldolgozás, mely alatt elsődlegesen képfeldolgozási eljárások összeségét kell érteni; második főrész pedig a sáv detektálására és követésére, a sávgeometria kiszámítására koncentrál

2.1 Előfeldolgozás

Az előfeldolgozó algoritmusok elsődlegesen szürkeárnyalatos képekre fókuszálnak, de találni színes képes megoldásokat is. A színes képes megoldások esetében főleg szegmentálási eljárásokhoz vagy színspecikfikus leírókat alkalmaznak, melyekhez elengedhetetlen, hogy rendelkezésre álljanak a színinformációk. A szürkeárnyalatos képek esetében általában valamilyen éldetektáló szűrőt alkalmaznak, főleg Sobel szűrőt, mely gyakorlatilag egy speciális 3x3-as konvolúciós mátrixot használ a szűréshez. Az egyszerűsége és a relatíve kicsi számításigénye miatt elég népszerű, de találhatunk Canny éldetektáló megoldásokat is, mely a folyamat során iteratív módon kiszűri a rövid és gyenge éleket. A Canny szűrő alkalmazásának előnye, hogy csak a fontos élek maradnak meg, de a nagy számításigénye miatt az alkalmazhatósága behatárolt. A képfeldolgozási folyamatokhoz hozzátartozik, hogy egyes alkalmazásokban a perspektív ábrázolásból adódó hibákat azzal küszöbölik ki, hogy inverz perspektív transzformációt hajtanak végre a képen, mely a gyakorlatban az út felülnézeti képének előállítását jelenti. Ezt a transzformációt a képfeldolgozási folyamat végén és elején is el lehet végezni. Ha képfeldolgozás elején történik a transzformáció, akkor csak a vizsgált területen lesz végre hajtva a képfeldolgozás folyamata, de a járműtől távolabbi részén a kép jelentősen torzul, míg a képfeldolgozás után elvégezve a kapott eredmény információ tartalma kevésbé torzul. Hátránya, hogy jelentősen növelik a számítási igényt.

2.2 Sávdetektálás és -követés

A detektálási és követési folyamatra is különböző megoldásokat láthatunk az egyszerű szabály alapú megoldásoktól a bonyolult RANSAC és Hough transzformációig. A szabály alapú megoldás esetében a kép fel van osztva különböző régiókban és aszerint, hogy a képfeldolgozási folyamat során kiszűrt sáv pontjai hova esnek, becsüli a sáv geometriáját. Ebben az esetben viszont megoldandó feladat a sávközépvonalának meghatározása és képesnek kell lennie kezelnie azt, ha a jármű nem párhuzamosan/érintő irányban áll a sávhoz képest, ezen adatok birtokában egy újabb transzformáció elvégzése után van lehetőség a geometria meghatározására.

Esetleges megoldás lehet másod- és harmadfokú görbék illesztése, ami alapján a sáv geometriája kalkulálható melyek a fent említett hibák kompenzálása ugyan szükséges, de azok elvégezhetők a kép transzformációja nélkül is egyszerű számítással. További algoritmusok is használatosak ugyanakkor ezek számításigénye sokkal nagyobb. A képpontokra adott matematikai modell illesztéséhez a Hough transzformáció (Ozgunalp és Dahnoun, 2014) is alkalmazható, mely során a különböző releváns képpontokat a Hough térbe transzformálva a releváns modell paraméterei kiemelkednek. A RANSAC módszer (Guo, Wei és Miao, 2015) esetében egy meglévő matematikai modell segítségével kiszűrhető a sávhoz nem tartozó pontok és kiszámolható a sávgeometriája.

Egy kezdeti detekció esetén részecskeszűrő (Berriel *és mtsai.*, 2015) alkalmazása is lehetséges, mely a igyekszik megbecsülni a vonal következő helyét a véletlenszerűen generált részecskék segítségével, majd a mérés alkalmával súlyozza azokat és kiválogatja a megfelelőket, végül a maradék átlagolásával egyetlen megoldás marad. A részecskeszűrőhöz hasonlóan a Kalman-szűrő(Lim *és mtsai.*, 2009) is igyekszik optimális becslést adni a vonalak paramétereiről azzal a különbséggel, hogy ebben az esetben egyetlen a vonal paramétereit becsüli folyamatosan.

Ezen a területen is hasznosíthatónak mutatkoznak a különböző neurális hálós megoldások, melyekkel egyszerű a sávdetektálási feladat megoldása, de esetenként az előfeldolgozást is ezekkel oldják meg. Általánosan elmondható a legtöbb esetben az eredmények egy nagy teljesítményű eszközön lettek fejlesztve és validálva, melyek az eddigi autóipari trendektől erősen eltérnek, ahol cél a költség és energia hatékonyság. Azonban a legújabb fejlesztések során a gyártók már egyre inkább számolnak a gépi tanulás lehetőségeivel és az ehhez szükséges célhardverek alkalmazásával.

2.3 Követelmények

Az algoritmussal szemben támasztott követelmények között van, hogy lehetőség szerint nappali időjárási viszonyokra érzéketlen, vagy csak egy szűk, jól behatárolt körülmény együttes esetén a rendszer hiba rátája egy tolerálható érték fölé emelkedik. Ezért az algoritmusnak nagy biztonsággal kell tudnia kezelni a töredezett útfelületet, a különböző fényerősség- és kontraszt viszonyokat pl.: árnyékos útviszonyok. Az algoritmusnak képesnek kell lennie információt szolgáltatni a sáv geometriájáról és a jármű pozíciójáról a sávon belül.

Az algoritmusnak továbbá elég gyorsnak kell lennie, hogy az valós időben is stabilan működjön, ezért 100 ms-os ciklusidő lett kitűzve, mely a gokarton elvégzendő kísérletekhez elegendő.

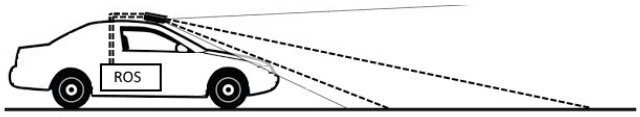
3. HARDVERARCHITEKTÚRA ÉS IMPLEMENTÁCIÓS LEHETŐSÉGEK

A sávdetektáló egységnek a megfizethető és rendkívül jó támogatottsággal rendelkező Raspberry Pi hardvert választottuk. Ezzel a járműipari kamerákhoz hasonlóan egy egységben meg lehet valósítani a teljes sávdetektálást a képfeldolgozástól egyészen a magas szintű sávinformációk kiküldéséig.

3.1 Raspberry Pi

A Raspberry Pi 3B+ verzióját használtuk, aminek központi eleme egy négymagos ARMv8 alapú SoC (System on Chip) 1.4 GHz -es processzor órajellel. Az eszköz rendelkezik számos elterjedt vezetékes és vezeték nélküli interfésszel és az alkalmazási céljainknak kedvező méretei vannak. A szabványos CSI (Camera Serial Interface) csatlakozó felületen keresztül egy 5 megapixeles natív felbontásra képes, fix fókuszú lencsével szerelt CMOS kamerát illesztettünk a rendszerünkhöz.

A Raspberry Pi eszközt használó kamera egység feladata a sávdetektálás, ami alapján a kísérleti tesztjárművön elhelyezett ipari PC valósítja meg a sávtartást az 1. látható módon. A számítógépen egy Linux rendszer alatt lévő ROS (Robot Operating System) bemenete lesz a magas szintű



sávinformáció és kimenete pedig a szükséges vezérlőjel.

1. ábra: A rendszer sematikus vázlata

3.2 Robot Operating System

A Robot Operating System (ROS) egy Linuxra épülő, nyílt-forráskódú meta-operációs rendszer, amely elsősorban önjáró robotok fejlesztésére készült.

Legfőbb előnye, hogy egységes, átlátható és egyszerűen integrálható kommunikációs keretrendszert biztosít az egyes funkciókat megvalósító szoftvermodulok (node-ok) számára. Az egyes node-ok akár azonos, akár önálló hardveren is futhatnak, a kommunikáció alapja TCP/IP, a sávszélességet csak a fizikai réteg korlátozza.

Emellett a rendszer számtalan szoftvercsomaggal rendelkezik a robotok környezetérzékelésének, lokalizációjának és irányításának megvalósításához. Továbbá több hatékony szoftvereszközt is találunk a robotok és a környezet fizikai szimulációjához, a szenzoradatok rögzítéséhez és megjelenítéshez, valamint a felépített ROS hálózat menedzseléséhez és felügyeletéhez.

Népszerűsége és széleskörű közösségi támogatása különösen alkalmassá teszi kutatási feladatokban való alkalmazásra.

3.3 Python

A Raspberry Pi számos implementációs lehetőséget biztosít. A fejlesztés első szakaszában Python nyelven készült a logika. A Python egy jelenleg rendkívül jól támogatott általános célú, nagyon magas szintű programozási nyelv. A fő tervezési filozófiája előtérbe helyezi az olvashatóságot és a gyors fejleszthetőséget a futás sebességgel szemben. A megírt programok önálló futásra alkalmatlanok, futtatáshoz egy interpreter kell, ezért a futtatáshoz nincs szükség hosszas fordításra.

3.4 C++

A fejlesztés második fázisában a rendszer gyorsítása érdekében C++ nyelven is elkezdődött a feladat implementálása. A C++ egy magas szintű, általános célú programozási nyelv, mely futtatásához fordítás szükséges. A fejlesztés nehezebb és időigényesebb folyamat, de a fordítás utáni kód jóval hatékonyabb és gyorsabban futó programot eredményez. A Python sebessége sokat fejlődött az évek alatt, de a jócskán alulmarad a C++ -hoz képest.

4. SZOFTVERARCHITEKTÚRA

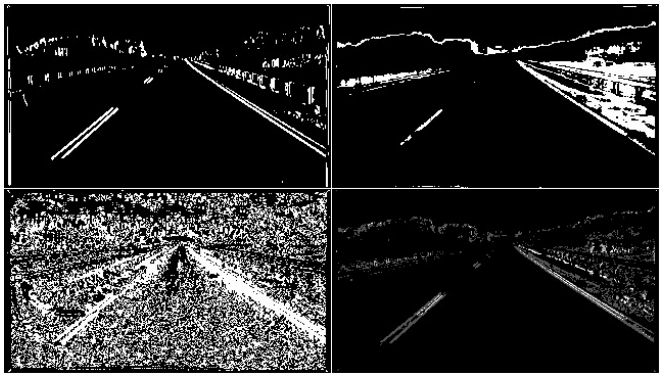
A fejlesztéshez több felvétel is készült a Raspberry Camerával, melyek színes, 320x180 felbontású videók, a rögzítés 15 fps sebességgel történt. A felvételek BGR színkódolási színtartományban készült.

A program 4 fő részből tevődik össze: képfeldolgozás, sávszélhez tartozó pontok kiválasztása, sávgeometria számítása és a végeredmény megjelenítése.

Az algoritmus fejlesztése során az OpenCV 4.1 (*OpenCV*, 2019) képfeldolgozó függvényei lettek felhasználva. Az OpenCV egy multiplatform, objektum orientált könyvtár, mely speciálisan a számítógépes látással kapcsolatos módszerekhez tartalmaz könnyen és egyszerűen használható, optimalizált függvényeket.

A képfeldolgozás 1. lépése, hogy a kamera belső hibájából adódó torzulásokat kiszűrjük vagy lehetőség szerint

minimalizáljuk. Ezek a hibák elsődlegesen a kamera belső felépítéséből, lencsék torzítóhatásából adódnak.



2. ábra: A képfeldolgozás által használt bináris képek: Sobel filter vízszintes irányba (bal felül), Pixelek él irányultsága alapján (bal alul), HSL színtér Fényesség információja alapján (jobb felül), valamint ezek súlyozott összege (jobb alul)

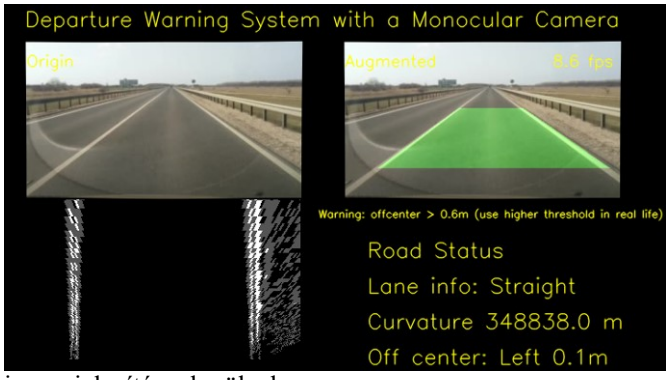
A képfeldolgozás 2. lépése a kép megszűrése, mely kétféleképpen történt. Egyrészt a kép HLS színtérbe történő konverziója utána a színtér L, azaz Lightness - Fényesség -, értékeire történt a szűrés, melyek alsó és felső korlátai úgy lettek beállítva, hogy a fényforrások és a sötét foltok kiszűrhetők legyenek. Másrészt a kép szürkeárnyalatos változatán Sobel-szűrők lettek alkalmazva x és y irányban, hogy az adott pixelek él jellege detektálható legyen. A szűrt képek segítségével további küszöb értékekkel való szűrése, valamint a kép pontok Sobel-szűrős képekből kalkulálható irányultságból további információk nyerhetők ki. A szűrési folyamat 3 db bináris értékekből álló képet eredményez.

A 3. lépés a kapott logikai értékekből álló képek kombinálása, mely során egyes kombinációkat különböző súllyal lettek összeadva, ezek eredménye a 2. képen látható. A 4. lépés a súlyozott kép inverz perspektív transzformáció, mely során az út felül nézeti képé konvertálódik át. Ez a lépés segít eliminálni a perspektív nézetből származó esetleges hibákat a későbbi sávgeometria számításnál.

A vonalak detektálásához és követéséhez 50 pixel széles vizsgáló sáv segítségével történt, hogy ezzel is csökkenthető legyen a szükséges számítások mennyisége. A nem nulla pontok kigyűjtésével a kapott pontokra másodfokú görbék illeszthetőek. A követés során az előző képkockán felismert vonalak mentén halad végig a képen és gyűjti össze a megfelelő pontokat a görbeillesztéshez. A görbeillesztési folyamat utána az algoritmus a kapott görbéket az utolsó 3 képkocka eredményével összeveti és kiátlagolja, ezzel szűrve a kapott eredményt. Az átlagolt görbék lesznek a detektálási és követési folyamat a kimenetei.

Sávgeometria számítása során a másodfokú polinomok átskálázásával valós méretű adatok nyerhetők ki, melyek segítségével számolható a sávgeometriája és a jármű a sávon belüli laterális helyzetét.

Végezetül az eredmény vizualizációja történik, ahogy az a 3. képen is látható. A vizualizáció során az eredeti kép mellett a detektált sáv kerül kiemelésre, valamint a képfeldolgozás végeredményeként keletkező szürkeárnyalatos kép is meg van jelenítve egy átskálázás után úgy, hogy az vizuálisan is értelmezhető legyen. A végeredményen további információk



is megjelenítésre kerülnek.

3. ábra: A szoftver vizuális eredménye

5. TESZTEREDMÉNYEK

Az algoritmus implementálása Python és C++ környezetben történt. A programmal támasztott minőségi kritériumok tesztelése során hamar nyilvánvalóvá vált, hogy a pythonos implementáció valós idejű futtatására csak korlátozott keretek között van lehetőség, a megcélzott 100 ms-os ciklusidő csak nehezen tartható. A lenti táblázat tartalmazza különböző szoftver részeinek futási idejét különböző gyorsítási megoldásokkal. A teljes ciklus idő alatt a képkocka beolvasása és a sávgeometria kiszámítása között eltelt idő van értve. Az 1. táblázatban található adatok 1127 képkocka hosszúságú videó anyag átlagát tartalmazza.

1. táblázat: Az egyes folyamatokhoz szükséges idő ezred másodpercben

	Python	Python + Numba	Python + SWIG	C++
Képfeldolgozás	89,6	98,1	69,9	69,4
Sávdetektálás és geometria számítás	23,0	27,7	14,2	N/A

A tisztán pythonos implementáció a fordító működéséből adódóan számításigényesé vált. A képfeldolgozás jelentősen nagyobb mennyiségű időt vett el, mint a detekció és a sávgeometria számolása, valamint a ciklusidő 80ms-os sávban szórt, ezért szükségesé vált egy olyan megoldás alkalmazása, mely gyorsabb működést biztosít lehetőleg a Python keretein belül.

Első körben szükséges áttekinteni, hogy az OpenCV C++-ban implementált függvények egy csatolón keresztül hívhatók meg. A képek és tömbök kezelésére az OpenCV-nek egy Mat nevű objektum szolgál, mely kifejezetten OpenCV specifikus, ezért konverzióra van szükség. A Pythonban erre a feladatra az általánosan használt numpy nevezetű könyvtár használható, mely a Python keretein belül képes a tömb változókat hatékony formában kezelni, valamint a tudományos számításokat leegyszerűsíteni.

5.1 Gyorsítási kísérletek

Az első megoldás a Numba (*Numba: A High Performance Python Compiler*, 2019) nevezetű könyvtár, mely a program fordítása során a kód egyrészét egy C nyelvhez közelebbi formába fordítja ezzel jelentős mennyiségű teljesítmény takarítható meg. A fő probléma a Numba esetében az volt, hogy az nem támogatja az OpenCV-t a numpy támogatás ellenére, így annak képességei erősen korlátozott módon volt kihasználható. A feldolgozás sebesség hasonló nagyságrendbe esik a sima pythonos megoldással bár működése során az ciklusidők egy 70 ms-os sávban szórtak.

A következő lépés tisztán Python-os megoldás helyett a nagy számítási kapacitás igénylő részek kiszervezése hardverközelibb program nyelvre ezzel sebesség növekedést elérve. A C++ környezet egy logikus választás az OpenCV miatt. Az implementálás és meghívható DLL-é történő fordítás után csak az adott függvényt kell kicserélni. Ugyanakkor ez további problémákat is magával hoz.

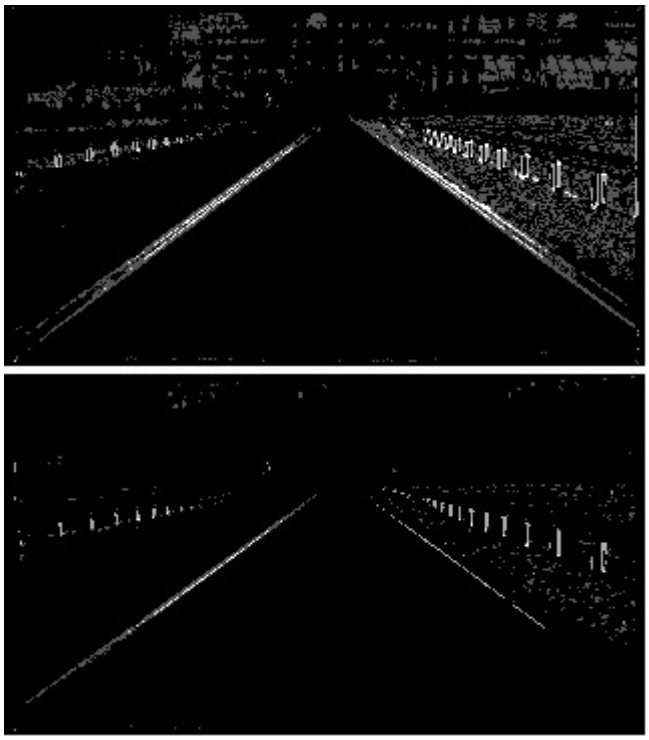
Python:
```
hls = cv2.cvtColor(img, COLOR_BGR2HLS)
s_channel = hls[:,:,2]
s_binary = np.zeros_like(s_channel)
s_binyary[(s_channel >= minHLS_value) & (s_channel <=
maxHLS_value)] = 1
```

C++ :
```
Mat lookUpTable1 = Mat::zeros(1, 256, CV_8U);
uchar* p1 = lookUpTable1.data;
for (int i = 0; i < 256; ++i)
{
   if ((i >= (int)minHLS_value)  & (i <= (int)maxHLS_value))
                        p3[i] = 1;
            else
                        p3[i] = 0;
}
cvtColor(temp, hls, COLOR_BGR2HLS);
split(hls, planes);
LUT(planes[2], lookUpTable1, hls_thres);
```

4. ábra: Python és C++ implementációja a HLS széntér fényesség értékeinek szűrésére

Egyrészt a C++-ban történő kódolás jelentősen szigorúbb, nagyobb körültekintést igényel a fejlesztő részéről, mint a Python. Másrészt az OpenCV Mat objektuma nehezen kezelhető, főleg, ami a képek egyes pixeleihez történő hozzáférést illeti (4. ábra).

Jelentős bonyodalmat okoz az is, hogy egyes függvények különböző eredményeket szolgáltatnak Python és C++ környezetben, így a képfeldolgozási eljárás végén az eredmények nehezen megfeleltethetőek egymásnak, ahogy az a 5. képen is látható.



5. ábra: A Python(felül) és a C++(alul) képfeldolgozás eredménye a bináris képek súlyozott összegzése után

Továbbá az OpenCV dokumentációja hiányos, melyet részben az internetes közösség képes pótolni.

A C++ kód meghívásához 2 megoldás is lett tesztelve. Először a Python beépített megoldása, a ctypes nevű könyvtár, mely a CDLL függvény segítségével meghívta a lefordított algoritmust. Ennek alkalmazása memóriakezelési problémákat hozott magával, mely mögött feltételezhetően az OpenCV áll.

Másodsorban a SWIG nevezetű wrapper(*Simplified Wrapper and Interface Generator*, 2019) segítségével történt, mely automatizált módon hozzáadja az összes szükséges kódot a megfelelő portoláshoz. Használata bonyolultabb és a létrejött kód is terjedelmes, de a folyamat végén egy DLL-t és Python algoritmust kapunk, melyet egyszerűen lehet használni a továbbiakban. A SWIG használatával sikerült az elvárható sebességre gyorsítani, ahogy ez a táblázatból is leolvasható. Továbbá a ciklus idő szórása is 40 ms alatt maradt, így az alkalmazás közel stabilan tudta tartani a 100 ms-os ciklus idő célt. De az eredményt fenntartással kell kezelni, hisz a képfeldolgozási folyamat után jelentősen kevesebb pont található képen, amire a másodfokú görbék illeszthetőek lehetnének, így ennek a programrésznek a javulása a ciklusidőt illetően nem várt volt. A képfeldolgozás során

használt szűrők újrakalibrálása szükséges, valamint a teljes algoritmus C++ nyelvre történő átültetése javasolt.

Egy esetleges platform váltás esetén az Intel Distribution for Python (*Intel® Distribution for Python* | Intel® Software*, 2019) nevű python interpreterével lehetne kísérletezni, mely x86-os mikroarchitectúrára lett optimalizálva, pl. Intel Edison, bár ennek a teljesítménye is lényegesen kisebb, mint a Raspberry Pi3-é. További lehetőség, hogy az OpenCV támogatja CUDA grafikai processzor architektúrát, mely nagyfokú párhuzamosíthatóságot tesz lehetővé képfeldolgozási műveletek esetében, ezzel felgyorsítva az alkalmazást. Ennek a kihasználásához az pl.: Nvidia Jetson nevezetű fejlesztői eszköze szolgálhatna alapjául, mely felár ellenében dedikált grafikus processzorral rendelkezik.

6. KONKLÚZÓ

Összegzésként elmondható, hogy habár a Python egy jól kezelhető, könnyen tanulható és használható fejlesztői eszköz, addig teljesítménykritikus alkalmazások esetén annak használata nem javasolt, vagy csak erős megkötések mellett. Az OpenCV alkalmazása sokat könnyít az ilyen jellegű feladatok megvalósításában, ugyanakkor nagy szakértelmet és tapasztalatot igényel. Valamint célszerű lenne egy olyan hardvereszköz alkalmazása, melynek grafikus teljesítménye és az általános számításokra alkalmas processzorok teljesítménye kiegyensúlyozottabb.

További feladatok az algoritmus teljes C++ környezetbe való átültetése és a valós körülmények között történő hangolása és tesztelése, továbbá ha szükséges, a használt szűrők redukálása.

7. KÖSZÖNETNYILVÁNÍTÁS

EFOP-3.6.3-VEKOP-16-2017-00001: Tehetséggondozás és kutatói utánpótlás fejlesztése autonóm járműirányítási technológiák területén - A projekt a Magyar Állam és az Európai Unió támogatásával, az Európai Szociális Alap társfinanszírozásával valósul meg.

HIVATKOZÁSOK

Berriel, R. F. *és mtsai.* (2015) „A Particle Filter-Based Lane Marker Tracking Approach Using a Cubic Spline Model”, in *2015 28th SIBGRAPI Conference on Graphics, Patterns and Images*. IEEE, o. 149–156. doi: 10.1109/SIBGRAPI.2015.15.

Dabral, S. *és mtsai.* (2014) „Trends in camera based Automotive Driver Assistance Systems (ADAS)”, in *2014 IEEE 57th International Midwest Symposium on Circuits and Systems (MWSCAS)*. IEEE, o. 1110–1115. doi: 10.1109/MWSCAS.2014.6908613.

Guo, J., Wei, Z. és Miao, D. (2015) „Lane Detection Method Based on Improved RANSAC Algorithm”, in *2015 IEEE Twelfth International Symposium on Autonomous Decentralized Systems*. IEEE, o. 285–288. doi: 10.1109/ISADS.2015.24.

Intel® Distribution for Python | Intel® Software* (2019). Elérhető: https://software.intel.com/en-us/distribution-for-python (Elérés: 2019. augusztus 9.).

Lim, K. H. *és mtsai.* (2009) „Lane Detection and Kalman-Based Linear-Parabolic Lane Tracking”, in *2009 International Conference on Intelligent Human-Machine Systems and Cybernetics*. IEEE, o. 351–354. doi: 10.1109/IHMSC.2009.211.

M Kumar, A. és Simon, P. (2015) „Review of Lane Detection and Tracking Algorithms in Advanced Driver Assistance System”, *International Journal of Computer Science and Information Technology*, 7(4), o. 65–78. doi: 10.5121/ijcsit.2015.7406.

Numba: A High Performance Python Compiler (2019). Elérhető: https://numba.pydata.org/ (Elérés: 2019. augusztus 12.).

OpenCV (2019). Elérhető: https://opencv.org/ (Elérés: 2019. augusztus 12.).

Ozgunalp, U. és Dahnoun, N. (2014) „Robust lane detection & tracking based on novel feature extraction and lane categorization”, in *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, o. 8129–8133. doi: 10.1109/ICASSP.2014.6855185.

Simplified Wrapper and Interface Generator (2019). Elérhető: http://www.swig.org/ (Elérés: 2019. augusztus 12.).