

**Virtuális eszköztervezés (GKLM\_AUTM029)**

**Schöffer Fruzsina Zsuzsanna**

**Mérnök Informatikus MSc szak**

**[2023]**

# Tartalomjegyzék

[Tartalomjegyzék 2](#_Toc134294526)

[Bevezetés 5](#_Toc134294527)

[1. Gépi látás 6](#_Toc134294528)

[1.1 Gépi látás szerepe a feladatban 7](#_Toc134294529)

[2. Hardveres és szoftveres környezet 8](#_Toc134294530)

[2.1 MyRIO 8](#_Toc134294531)

[2.1.1 Labview 8](#_Toc134294532)

[2.1.2 Eclipse C/C++ 10](#_Toc134294533)

[2.2 Raspberry Pi 11](#_Toc134294534)

[2.3 Összegzés és kiválasztás 12](#_Toc134294535)

[3. Szoftveres funkciók megtervezése 13](#_Toc134294536)

[4. Módszerek kutatása 17](#_Toc134294537)

[4.1 Jellemző pontok keresése 17](#_Toc134294538)

[4.2 Haar Cascade klasszifikáló 18](#_Toc134294539)

[4.3 Neurális háló betanítása 20](#_Toc134294540)

[4.4 OpenCV képfeldolgozás 21](#_Toc134294541)

[5. Választott módszer implementálása 24](#_Toc134294542)

[5.1 Kockakereső modulok 24](#_Toc134294543)

[5.1.1 Képfeldolgozó modul 25](#_Toc134294544)

[5.1.2 Paraméterszámító függvények 26](#_Toc134294545)

[5.1.3 Segítőszámító modulok 27](#_Toc134294546)

[5.2 Kockára pozicionáló modulok 28](#_Toc134294547)

[5.3 Implementáció bemutatása 31](#_Toc134294548)

[6. Összegzés 33](#_Toc134294549)

[Irodalomjegyzék 34](#_Toc134294550)

Ábrajegyzék

[**1*.* ábra**Pálya 7](#_Toc134294695)

[**2. ábra** Csatlakozás MyRiora 9](#_Toc134294696)

[**3. ábra** Eclipse tool chain editor 10](#_Toc134294697)

[**4. ábra** Folyamatábra a lefutásról 14](#_Toc134294698)

[**5. ábra** Folyamatábra a kockára pozícionálásról 16](#_Toc134294699)

[**6. ábra** Feature matching eredménye szürke képen 17](#_Toc134294700)

[**7. ábra** Haar Cascade modell felismerése 18](#_Toc134294701)

[**8. ábra** Rossz felismerés 19](#_Toc134294702)

[**9. ábra** További felismerések 19](#_Toc134294703)

[**10. ábra** Saját betanított neurális háló felismerés 21](#_Toc134294704)

[**11. ábra** Neurális háló rossz felismerés 21](#_Toc134294705)

[**12. ábra** Kockák felismerése 31](#_Toc134294706)

[**13. ábra** Részleges kék kocka felismerése 32](#_Toc134294707)

[**14. ábra** Rápozícionálás a kockára 32](#_Toc134294708)

Táblázatjegyzék

[**1. táblázat** MyRio és Raspberry Pi 3 összehasonlítása 13](#_Toc134127565)

# Einstein/Zebra puzzle megoldása genetikus algoritmussal

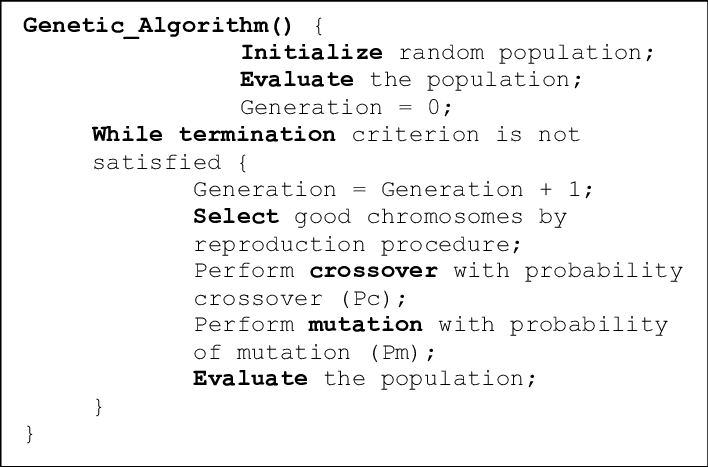
Az Einstein puzzle-t másnéven Zebra puzzle-ként is lehet ismerni, melyben arra a kérdésre keressük a választ, hogy adott feltételek mellett, kié a Zebra.

A puzzlet feltehetőleg Einstein találta ki fiatal korában, és a népesség kis százaléka képes csak megoldani.

A puzzle alapja, hogy van 5 házunk, amikben 1-1 ember él, és személyenként eltérően ők isznak valamilyen italt, szívnak dohányt, tartoznak valamilyen nemzetséghez, illetve más színű a házuk is, és van valamilyen háziállatuk. Ezekkel a paraméterekkel vannak megadva különböző kikötések, például, hogy a brit lakik a piros házban, vagy az, aki Lucky Strike cigarettát szív, az narancslevet iszik.

Ezek alapján kell megtalálni, hogy kié a hal, vagy a zebra (ha a Zebra puzzle-ről van szó).

A feladat megoldására genetikus algoritmust használtam, aminek a pszeudokódja az 1. ábrán látható.



***1. ábra*** Genetikus algoritmus pszeudokódja

*Forrás: https://www.researchgate.net/figure/Standard-pseudocode-of-genetic-algorithm\_fig1\_339802304*

A genetikus algoritmusok az evolúciós algoritmusok alá tartoznak, és az evolúciót mimikálják a működésük során.

Ez azt fogja jelenteni, hogy van egy kezdeti populációnk, ami véletlenszerűen van létrehozva, majd egy iteratív folyamaton keresztül a populációt fogjuk értékelni, és egy részét kiemelni egy fitnesz határérték alapján. Ezután újabb egyedeket hozunk létre a generációnkba keresztezés vagy mutációval, vagy mindkettővel. Ezt a keletkezett populációt újból kiértékeljük, és ha nem érte el az általunk támasztott kilépési feltételt, jelen esetben a jó megoldást, úgy továbbmegy az iterációnk.

A genetikus algoritmust C#-ban valósítottam meg.

Ha kicsit visszatekintünk a közelmúltra, akkor az autóipar egyik legújabb vesszőparipája az önvezető autók. Ezek komplex, biztonságkritikus rendszerek, melyek több szenzor egyidejű felhasználásából állnak. Ilyen része ennek például a sávtartó rendszer is, mely kamera segítségével határozza meg valós időben, hogy jó helyen tartózkodik-e a gépjármű és ez alapján korrigál is.

Itt meg lehet említeni a gépi látást is, mely egy kép alapú adatgyűjtésen alapuló rendszernek mondható, ami a beérkezett adatok alapján vezérlési vagy szabályozási folyamatot indít el. De az is lehet, hogy ez csupán adatgyűjtés céljából történik, és más rendszerek használják ezt fel, és hoznak ez alapján döntést.

Ezeknél a rendszereknél fontos, hogy ne legyen szükséges az emberi beavatkozás, hanem önállóan, a környezetet megfigyelve, és elemezve képesek legyenek döntést hozni. Egy rendszert akkor nevezhetünk autonómnak, ha egy bizonyos fokkal képes feladatokat elvégezni.

A dolgozatban egy általam fejlesztett gépi látást használó autonóm robot működéséről lesz szó, amely a 2020-as WRO World Robot Olympiad versenyre készült. Részletesen ki fogok térni arra, pontosan mi a rendszer felépítése, milyen hardveres és szoftveres megvalósítás történt, és ennek mi volt az oka.

A gépi látás rendszer alapvető célja, hogy feldolgozza egy kamera élő képét, számításokat végezzen ezzel, és ezek alapján döntéseket hozzon.

A robotnak egy pályát kell teljesítenie, mely akadályokkal van felépítve, és ezeknek az akadályoknak, illetve pályának a méreteit ismerjük, viszont azt előre nem tudhatjuk ezek hol vannak elhelyezve.

# Gépi látás

A gépi látás mint olyan manapság igen elterjedt, és nagy szerepet tölt be az élet majdnem minden területén. Legyen szó akár katonaságról, egészségügyről, bármilyen termelésről, a gépi látáson alapuló rendszerek elengedhetetlen részét képezik ezeknek.

A gépi látás manapság igen elterjedt, és egyre nagyobb szerepet kap, de ennek első gyökerei már az 50-es évekre vezethetőek vissza. A 70-es években már a kézi és a gépi írás között tudott különbséget tenni, manapság pedig pillanatok alatt be tudunk olvasni valamit a telefonunkkal és kielemezhetjük ezt. A fejlődés az elmúlt évtizedekben jelentősen felgyorsult, köszönhetően a mesterséges intelligenciának is.

A gépi látást nem feltétlen lehet egy konkrét témakörként leírni, hiszen több kisebb témakört is magába foglal. Ezek lehetnek olyan dolgok is, amik az emberi érzéskelés határain is túlmennek. Az angol szakzsargon például az emberi érzékelés határain túlmenő dolgokat a Machine Vision témakörébe, míg az emberi érzékeléshez köthetőeket a Computer Visionhez sorolja. A magyarban nem igazán teszünk különbséget a kettő között.

Ez azt jelenti, hogy a gépi látásba szenzorként nem csak a kamerát sorolhatjuk, hanem olyan érzékelőket is, mely a környezetünkről tud információkat szolgáltatni.

Egy ilyen rendszert feloszthatunk különböző részekre, például a mintavételezésre, az adatok feldolgozására, majd pedig ezek értelmezésére. Ezek a részek feladatonként különbözhetnek, így nehéz egy olyan rendszert készíteni, ami több feladatnál is megállja a helyét.

[1]

## Gépi látás szerepe a feladatban

Dolgozatomban egy olyan rendszert mutatok be, ahol egy robotnak autonóm módon kell működnie, vagyis elemeznie kell a környezetét, és a feldolgozott információk alapján kell döntést hoznia. A robotnak egy adott feladatot kell teljesítenie. A feladat, hogy több különböző színű kockát, melyeken más színű kis kockák vannak elhelyezve, felismerjen, és a kis kockákat a megfelelő színű nagy kockára pakolja. A feladat elején a robot egy garázsban tartózkodik, ahonnan kijutva egy QR-kód beolvasásával fel kell térképeznie a pályát, és azt, hogy hol vannak a kockák. A robotnak először mindig a kék kockához kell eljutnia, és arról leemelni egy kis kockát, majd pedig elmennie ahhoz a nagy kockához, amilyen színű a leemelt kis kocka.

A feladat bár elsőre egyszerűnek tűnik, a háttérben mégis rengeteg dolognak kell történnie, hogy ez teljesíthető legyen.

A gépi látás rendszer tehát elsősorban a kockák felismeréséért, azok helyzetének meghatározásáért, a színek és méretek megfelelő azonosításáért fog felelni. Az 1. ábrán egy példa pálya látható, amin a robotnak végig kell mennie.

# Hardveres és szoftveres környezet

Ahhoz, hogy a robot teljesítse a pályát, többféle hardveres és szoftveres implementáció jöhet szóba. Fontos, hogy a gépi látás modulra mindenképp szükség lesz, viszont ennek a megvalósítása függ a hardveres környezettől is.

## 2.1 MyRIO

Az első lehetőség a feladat megoldására a myRIO-1900, ami egy jelentős hardveres specifikációkkal bíró SBC, vagyis Single Board Computer, FPGA-val, vagyis field-programmable gate array-vel.

A MyRIO igen népszerű, elég sok helyen van használva akár egyszerű feladatok elkészítésére, és szívesen használják a tanulók is. Szerencsére olyan specifikációkkal rendelkezik, amik miatt komplex feladatokat is végre tud hajtani. A népszerűségéhez hozzájárul az is, hogy támogatja a Labview-t, de lehet C vagy C++ nyelveken is fejleszteni rá.

Mindegyik említett nyelvnek megvannak az előnyei és a hátrányai, így a következőkben ezekről lesz szó.

### 2.1.1 Labview

Az első lehetőség a már előzőleg említett Labview, ami egyszerű grafikus felületet biztosít az ezt használóknak, amivel nem feltétlen kell mélyebb szinten ismerni a különböző algoritmusokat és adatstruktúrákat sem. Természetesen illik itt is ismerni a szekvenciák, vagy a ciklusok működését, hogy tényleg olyan szoftvert lehessen írni, ami meg is tudja oldani a feladatot, de konkrét szintaktikai ismeret nem feltétlen szükséges.

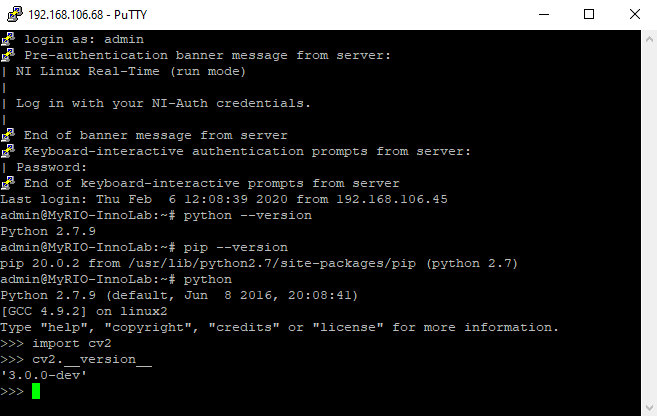
A Labview rengeteg modullal rendelkezik, mint az úgynevezett NI Vision, ami nem ingyenes, vagy az ennek ingyenes, de lebutított verziója, az NI Vision Assistant.

Szerencsére az NI Vision Assistant rendelkezik egy kockafelismerő funkcióval, viszont ez néhány teszt alapján nem feltétlen adott megbízható eredményeket, így ezt a lehetőséget tovább nem vizsgáltam. Sajnos ez az egyik alappillére a feladat teljesítésének, így úgy döntöttem, figyelemmel arra is, hogy az Assistant egy ingyenes verziója csak az NI Visionnek, hogy nem a Labview saját gépi látás modulját fogom használni.

Valószínűleg rengeteg olyan beépített funkciója van, ami általánosságban hasznos tud lenni, viszont a feladat specifikussága miatt lehet több munka az, hogy ezeknek valamilyen kombinációjával létrehozzak egy működő rendszert, minthogy valóba egy sajátot fejlesszek le.

Szerencsére a Labview ugyanakkor támogatja, a Python scriptek integrálását is, így lehetséges implementáció az is, hogy a robot az összes mozgással kapcsolatos funkciója Labviewban kerül lefejlesztésre, és a gépi látással kapcsolatos feladatok pedig Pythonban, majd ezek kerülnek meghívásra, és az értékeikkel pedig dolgozik tovább a robot.

Ehhez először is az előbb említetteket telepíteni kell a Python 3-mat myRIO-ra az opkg-val, ami egy package manager. Ez a művelet látható például a 2. ábrán.



**2. ábra** Csatlakozás MyRiora

Mivel még nem kezdődött el a gépi látás modul implementálása, ami akár magába foglalhat mesterséges intelligenciát is, így első körben arra voltam kíváncsi, hogy a MyRIO számítási kapacitása, hogyan fogja ezt bírni.

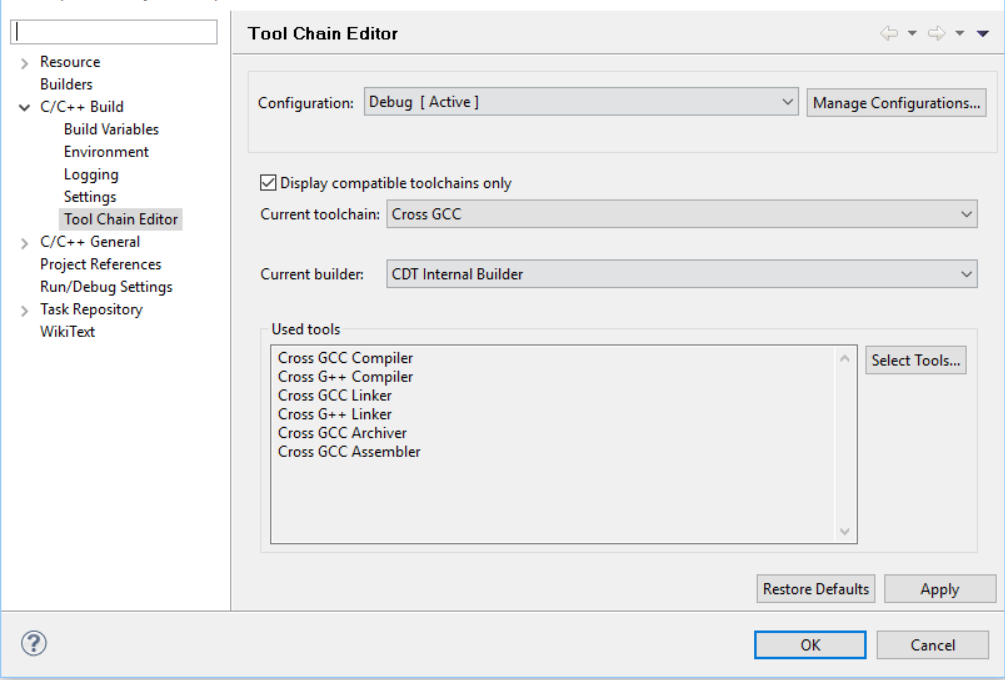
Szerencsére manapság rengeteg olyan implementáció van ezekre, hogy mobiltelefonon, vagy kisebb számítási kapacitással rendelkező eszközön lehessen akár neurális hálókat futtatni (természetesen nem a betanításukat). Pontosan ezért először pár alapvető OpenCV-ben is megtalálható klasszifikációs funkciót is leteszteltem, mint például a Haar Cascade-dal.

A megfelelő memóriafelhasználás alapvetően nem probléma, mivel mind a Python, mind a Labview memóriakezelése implementált, és tekintettel arra, hogy a Python scriptek csak bizonyos részfeladatokat látnának el, de számítást nem végeznének, így a nagy része Labview-ban történne.

### 2.1.2 Eclipse C/C++

A következő lehetőség C vagy akár C++ nyelven elkészíteni az implementációt. Ahhoz, hogy ezt meg lehessen csinálni, a National Instruments által módosított Eclipse-t kell letölteni, és ezt használni a fejlesztéshez. Ennek az első lépése a kód buildelése, majd pedig a bináris fájlnak a mikrokontrollerre való deployolása.

Mivel gépi látás modult szeretnénk használni, ahhoz szükség lesz az OpenCV-re is. OpenCV pedig csakis C++-hoz érhető el, így egyértelműen ezt kell használni C helyett.



**3. ábra** Eclipse tool chain editor

Az Eclipse-ben, hogy MyRio-ra tudjak fordítani, először is meg kellett adni, hogy milyen Toolchain-t használjak. A C/C++ Build beállításokon belül a Tool Chain Editornál meg lehet adni, hogy a toolchain Cross GCC legyen, ami arra fog kelleni, hogy Myrio-ra deployolhassam a programot, ez látható a 3. ábrán is.

A következő, amire szükség van, az az Opencv buildelése, hiszen csakis Visual Studio-hoz lehet prebuilt library-t használni. Ehhez először is telepítettem a Cmake-t, klónoltam az OpenCV egyik stabil verzióját Github-ról, majd a Cmake segítségével elkezdtem elkészíteni a makefile-t. Ez elsőre egyszerűnek tűnt, viszont Windowson folyamatos hibákba ütköztem, hiányzó dependency-k miatt. Többszöri próbálkozás után végül ezt Linux Mint-en, ugyanúgy Cmake segítségével megcsináltam.

A folyamat után rendelkeztem egy Eclipse-hez használható OpenCv-vel, aminek az egyes könyvtárait már csak össze kellett linkelni a futtatandó C++ kóddal. Miután ez is megtörtént, és buildel-ve volt az adott kód, ki lehetett küldeni a MyRio-ra. Szerencsére minden további nélkül sikeresen működtek az OpenCV modulok.

A C++ használata közben az egyik felmerülő aggály a memóriakezelés, a másik pedig az ultrahangszenzorok használata volt, így bár lehetséges a megvalósítás, de jobb alternatíva esetén nem ezt a megoldást választottam.

## 2.2 Raspberry Pi

A következő lehetőség, melyet megvizsgálok, az a Rasberry Pi, amire több nyelven, köztük Pythonban is lehet fejleszteni. Az előzőek alapján a gépi látás végett logikus döntés Python-t használni az implementálásra, így egy Raspberry Pi jó választás tud lenni hardver szempontjából.

Érdemes természetesen előtte összehasonlítani, pontosan milyen különbségek vannak a 2 eszköz között, például memória vagy akár processzor szintjén. Ezt az 1. táblázat mutatja be.

**1. táblázat** MyRio és Raspberry Pi 3 összehasonlítása

|  |  |  |
| --- | --- | --- |
|  | Myrio 1900 | Raspberry Pi 3 Model B |
| USB-portok száma | 1 | 4 |
| Processzor | Xilinx Z-7010 667 MHz | Quad Core Processor 1.2GHz |
| RAM | DDR3 512MB | DDR3 1GB |

A Raspberry Pi alapvetően több szempontból is jobbnak mondható. Az első összehasonlítási alap például az USB-portok száma. Bár ez nem tűnik annyira lényeges specifikációnak, de a látás megvalósításához egy webkamera lesz használva, így fontos ennek lehetősége. Ha pedig bármi olyan másik eszköz kerül beépítésre, amihez USB-port kell, akkor nem árt, ha van több elérhető is.

Az USB-porton kívül látható, hogy RAM és processzor szempontjából is jobban teljesít a Raspberry Pi, ami azt is jelenti, hogy könnyebben tud dolgozni akár nagyobb erőforrásigényű neurális hálókkal.

## 2.3 Összegzés és kiválasztás

Az előzőek alapján tehát 3 lehetőség lett bemutatva, ami szóba jöhet, és ezek közül kell egy olyat kiválasztani, ami megfelelő számomra.

Az első lehetőség a MyRIO, melyet lehet Labview-val használni, ezen belül is csakis egyszerűen a Labview saját moduljaival, mint an NI Vision-nel, vagy pedig Python scriptekkel. A második lehetőség is ugyancsak MyRIO, viszont a Labview helyett C++-t lehetne használni.

A harmadik lehetőség a Raspberry Pi és a Python használata.

Tekintettel arra, hogy Pythonban már többször is programoztam, és jobban ismerem a szintaktikáját, mint a Labview-nak, vagy a C++-nak, úgy döntötten, hogy Pythonban fogom megvalósítani a gépi látás modult.

# Szoftveres funkciók megtervezése

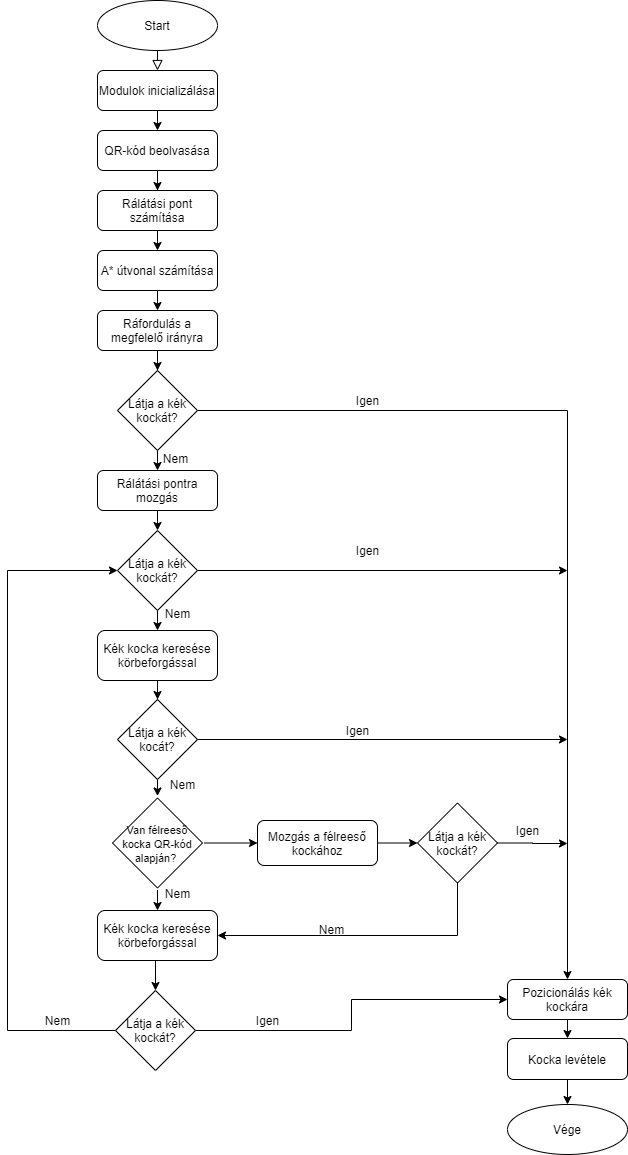
A következő, mielőtt a konkrét implementáció megtörténne, egy pontosabb tervezés arról, milyen funkciók is szükségesek a megfelelő működéshez, vagyis a pálya elvégzéséhez.

A pálya elvégzéséhez egy bizonyos lefutásnak kell megvalósulnia, mivel a pályán először egy QR-kódot kell beolvasni, azt dekódolni, majd pedig megtalálni a kék nagy kockát.

A 4. ábra mutatja be ezt a lefutást, amiből nekünk csakis azok a részek lesznek érdekesek, ahol konkrét gépi látásra van szükség. A robot tehát először beolvassa a QR-kódot, dekódolja azt, kiszámol egy olyan pontot a pályán, ahonnan remélhetőleg az összes kockát látja egyszerre, így nem kell hosszasan feltérképezni azt. Ekkor el kezdi keresni a kék kockát, ha nem találja meg, 360 fokban körbeforog és közben folyamatosan keresi a kockát, ha megtalálja, akkor megáll annak az irányába.

Ha nem találta meg, akkor a kocka valószínűleg ki van takarva, és a QR-kód alapján meghatározza melyik kocka lehet kitakarva, és elindul ehhez. Mindezek közben is folyamatosan figyeli, hogy megtalálja-e a kék kockát.

Ahogy megvan a kék kocka, úgy arra rápozicionál, hogy egyenesen meg tudja közelíteni, majd levenni róla a kis kockát. A közelítés során folyamatosan figyeljük kamerával milyen messze vagyunk, és mennyit kell még haladni a kockáig, és azt is, hogy a robot horizontális éle és a kocka vele szemben levő éle párhuzamosak-e. Erre azért van szükség mert a távolságmérő szenzorok rövid távon tudnak pontosabb értékeket adni.



**4. ábra** Folyamatábra a lefutásról

Az előző lefutás alapján látható tehát, hogy elég sok helyen szükség van a gépi látásra. A következő ábrán ezt kicsit részletesebben is bemutatom.

Amennyiben megtaláltuk a kék kockát, oda kell menni hozzá. Ahhoz, hogy a legkönnyebben tudjuk levenni róla a kis kockát, muszáj párhuzamosnak lennie a robotnak a kockához. Ha volt körbeforgás, akkor lehet, hogy a robotot meghajtó motorok és a pálya egyenetlensége miatt a robot el van csúszva, így muszáj ráfordulni a kockára.

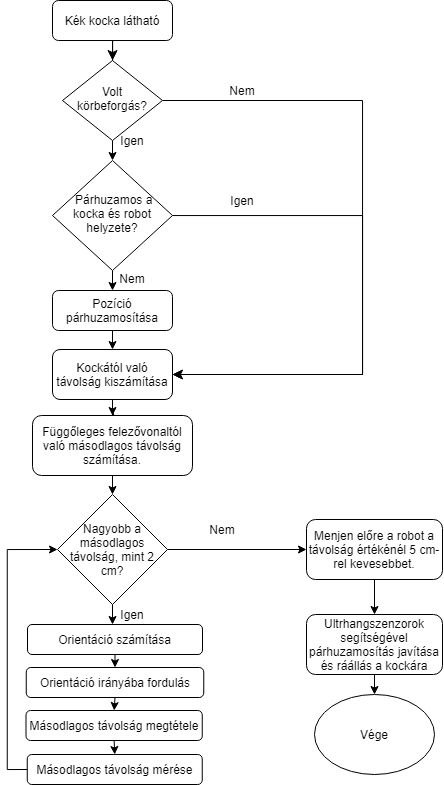
Ha nem volt elfordulás, akkor ez nem szükséges. Az elfordulást egyszerűen a már megtett szöggel lehet kiszámítani, és ez alapján korrigálni. Viszont még ekkor is marad egy fennálló probléma, már pedig ez az, hogy lehet, hogy a robot nem a kockával szemben van, és ha elindulna felé, akkor nem a kocka előtt lenne pontosan, hanem valamivel mellette.

Sajnos az ultrahangszenzorok nem feltétlen megbízhatóak az előbbi esetben, mivel a szenzor el tud mérni mellette, és valami mást észlelni a háttérben, és ezzel pontatlan eredményt adni. Így kamerával meg tudjuk mérni, hogy mennyit jobbra vagy balra mozogni a robotnak, hogy pontosan a kocka elé kerüljön. Ez egy iteratív feladat, ami addig történik, amíg a mért távolság, amit másodlagos távolságnak nevezek a dolgozatban, kisebb mint 2 cm.

Ezek után a robot elkezdi megtenni a távolságot, majd amikor elég közel ért hozzá az ultrahangszenzorokra vált távolságméréshez, és ez alapján közelíti meg a kockát és veszi le a kis kockát.

Ezek után a gépi látás modulnak az előbb leírtakat kell ismételnie, csupán más színekkel.

Összefoglalva tehát a rendszer feladata elég sokrétű. A gépi látás rendszernek fel kell ismernie a kockákat, mint objektumokat, fel kell ismernie ezek színét, meg kell tudni határoznia a képernyőn való poziciójukat, majd pedig a távolságukat is, ki kell tudnia azt is számolni, hogy milyen szögben van elfordulva a robottól, és ami bár nem egy nehéz feladat, de be kell olvasnia egy QR-kódot is.



**5. ábra** Folyamatábra a kockára pozícionálásról

Az előzőekben leírtak alapján tehát körvonalazódott, hogy milyen funkciókat kell tudnia megvalósítania az egyes moduloknak, és az is, hogy azok egymással milyen adatokkal kell, hogy kommunikáljanak. Ebből kiindulva meg lehet tervezni, hogy milyen osztályok, illetve milyen modulok legyenek implementálva, milyen függvényekkel, paraméterekkel illetve változókkal.

# Módszerek kutatása

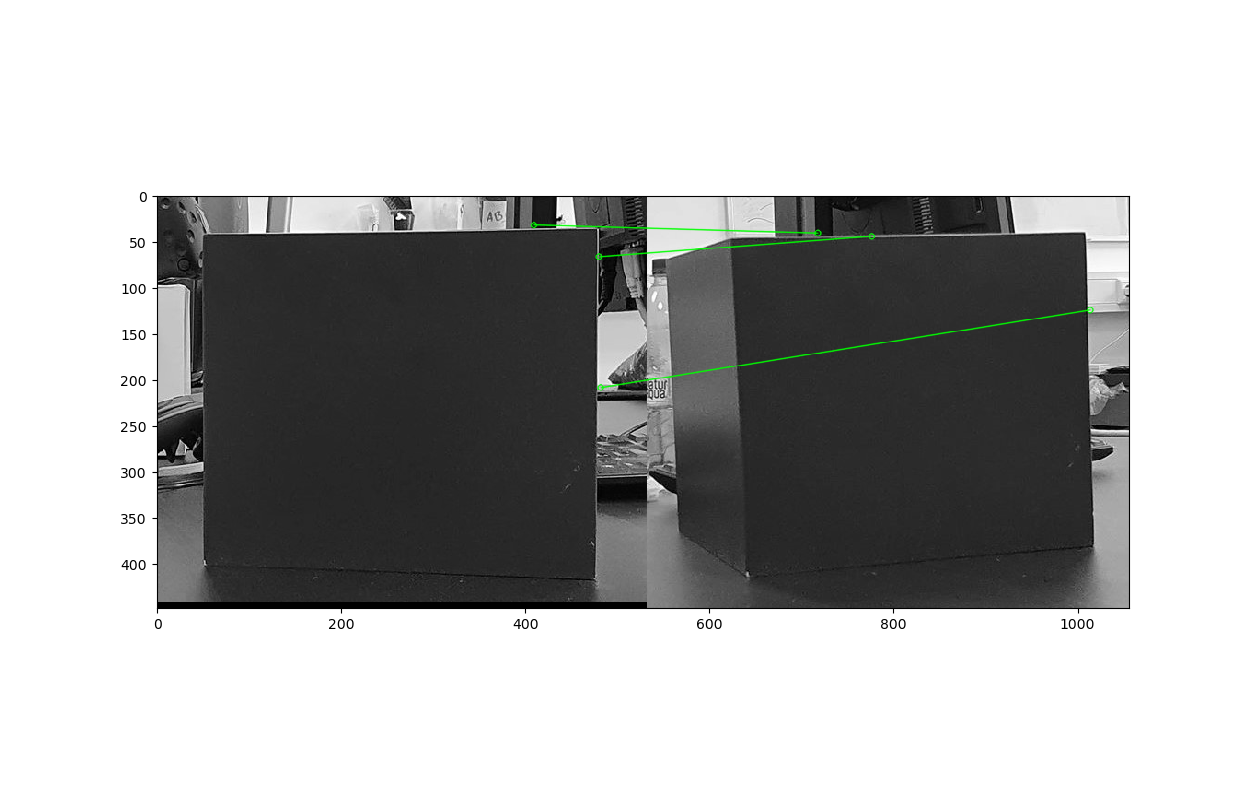
Mielőtt egy konkrét implementáció mellett döntenék, szeretnék minden lehetőséget megvizsgálni. Mivel az alapfeladat, hogy felismerjük a kockákat, és a színeiket, egy eléggé alap kérdése a gépi látás témakörének, így elég sok féle megközelítés lehet erre.

## 4.1 Jellemző pontok keresése

Az első módszer amit kipróbáltam az a jellemző pontok keresése, aminek az a lényege, hogy egy objektumot egy képen felismerve egy másik képen is be tudunk azonosítani. Ehhez a képen úgynevezett érdekességi pontokat határozunk meg, majd pedig ezekhez leíró vektorokat készítünk, ami gyakorlatilag a tulajdonságok leírása. Ez a vektor a 2 pont köré lesz elkészítve, és invariáns lesz az olyan transzformációkra mint az elforgatás, nagyítás stb, éppen ezért ha a keresett objektum mondjuk kicsit el van forgatva a képeken, akkor is meg kellene tudnia találnia az algoritmusnak a vektorok alapján. [2][3]

Létezik a vektorokból globális és lokális is, előbbi rosszabbul teljesít, ha az egész képen történik változás. Mivel a robot folyamatos mozgásban lesz, így a lokális vektor jobb ötlet lehet, mivel az ilyen esetekben nincs befolyásolva.

Több algoritmus is létezik ezekre, mint például a SIFT, SURF, KAZE, AKAZE, amelyből talán a legkedveltebb az AKAZE. [4] Éppen ezért ezzel próbáltam meg először is az összehasonlítást, aminek az eredménye a 3. ábrán látható. Mindösszesen 3 pontot talált az algoritmus, ami elég kevés, bár várható volt, mivel a kocka egyszínűsége nem sok jellemző pontra ad lehetőséget.



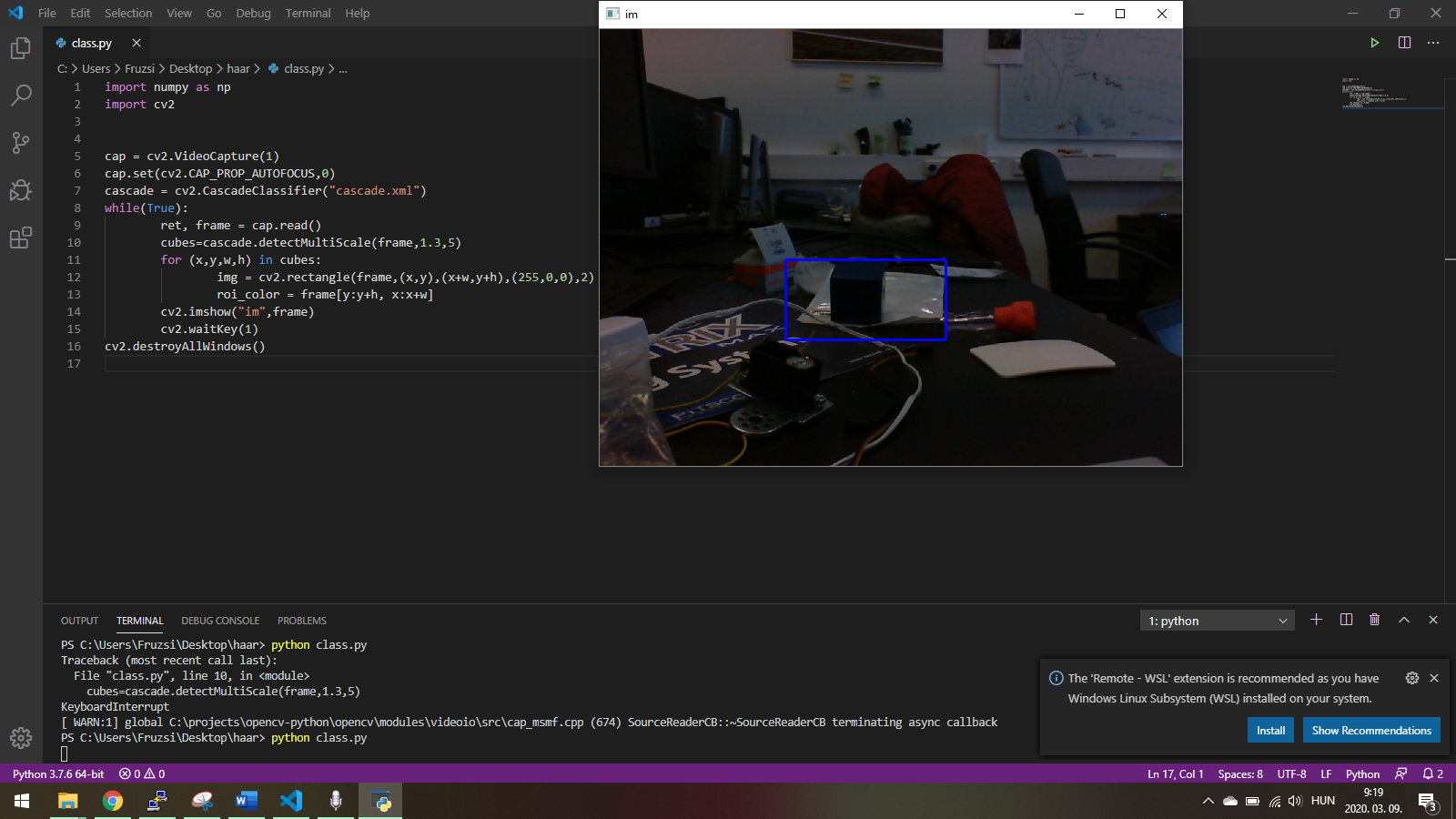
**6. ábra** Feature matching eredménye szürke képen

## 4.2 Haar Cascade klasszifikáló

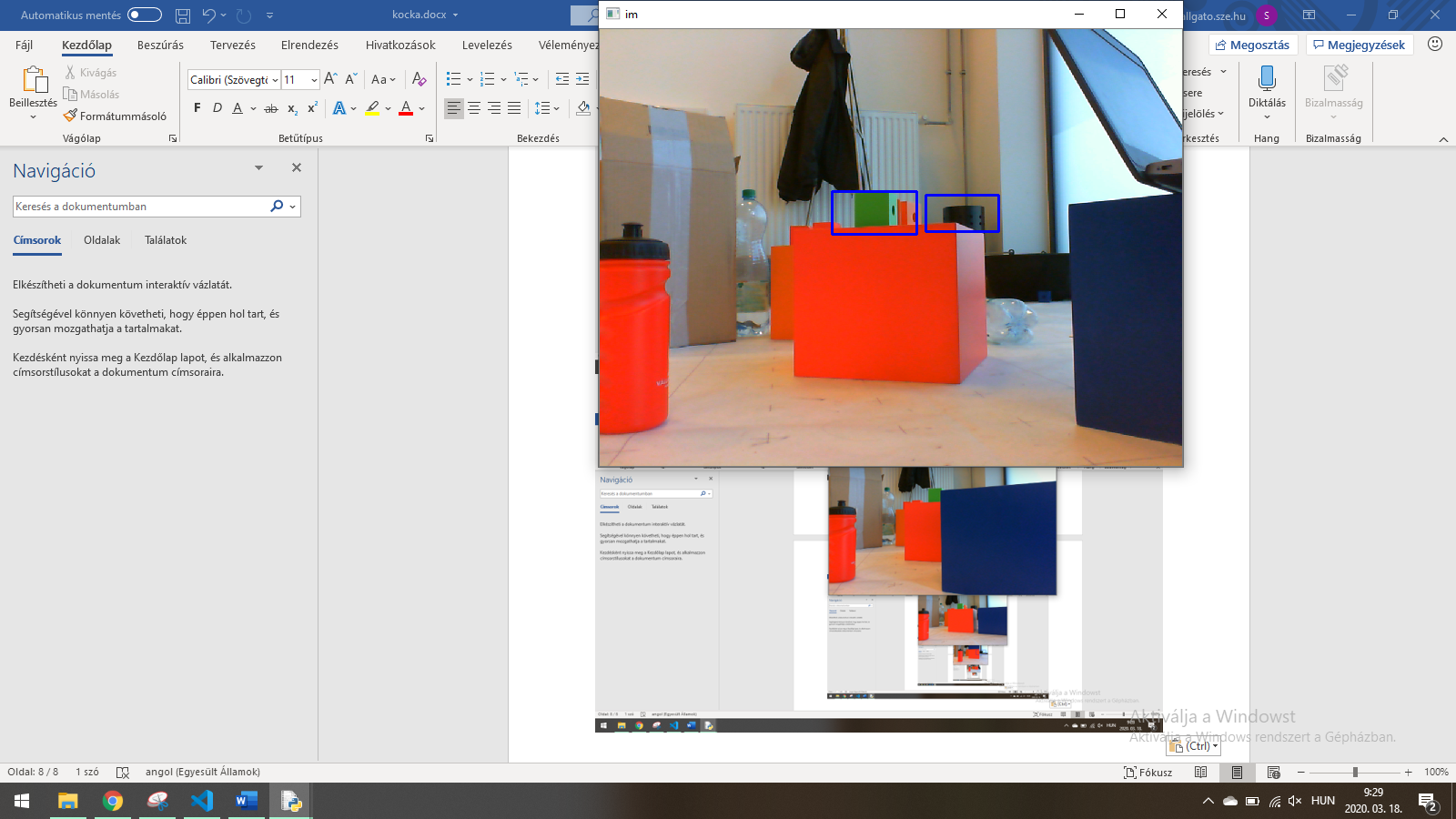
A következő módszer a Haae Cascade klasszifikáció, ami egy már több mint 20 éves gépi tanulási módszer. Ennek az alapja, hogy az algoritmus a pixelintenzitást veszi figyelembe a képeken több kijelölt helyen, és ezek között számolja ki a különbséget.

Az algoritmus a betanításhoz negatív és pozitív képeket használ. A negatív képek azt jelentik, hogy nincs rajtuk az az objektum, amit fel akarunk ismerni, míg a pozitív képeken pedig rajta van. [5]

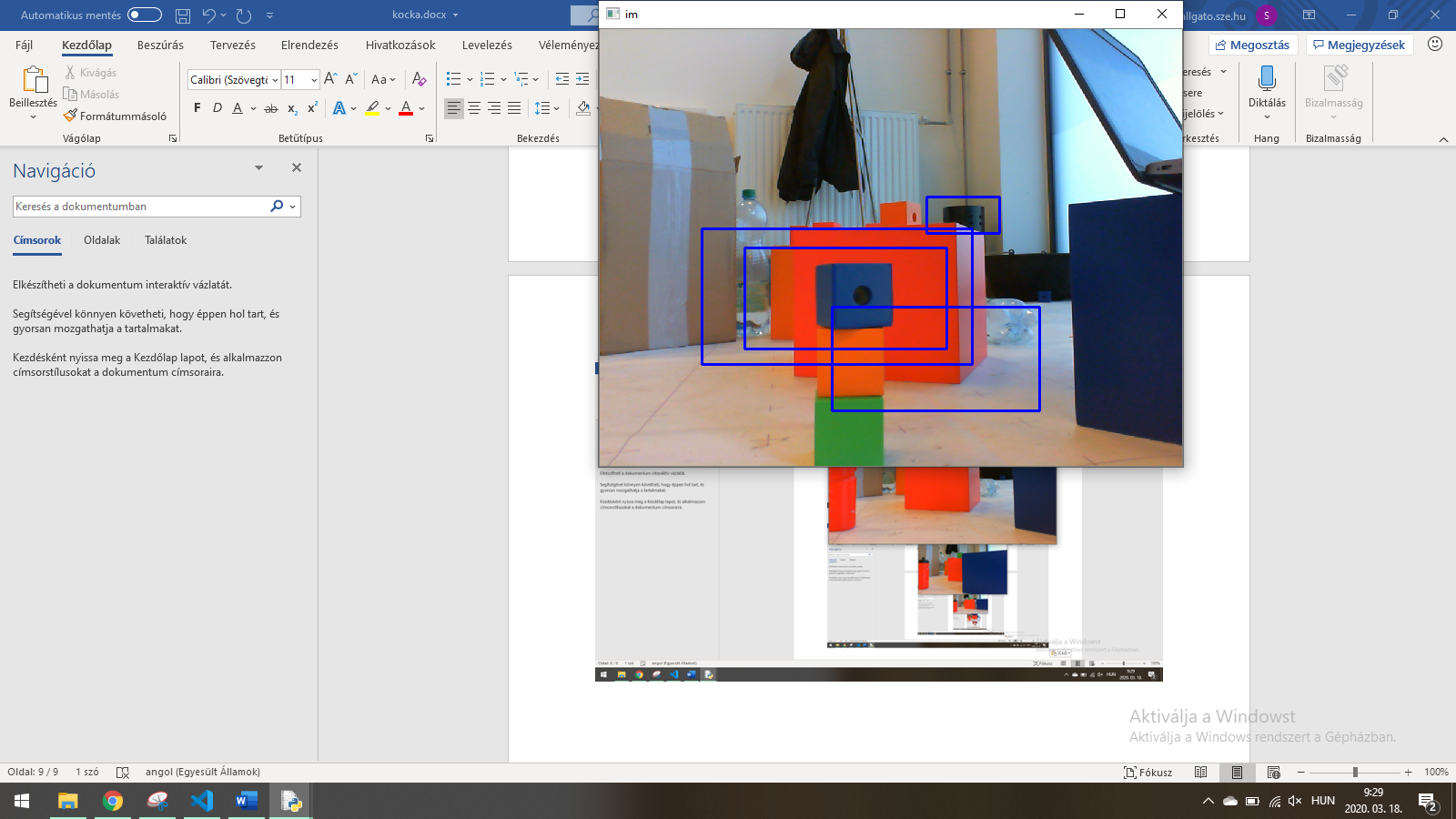
Mivel a feladat teljesen egyéni, így saját adathalmazzal tanítottam be a modellt, amihez több negatív és pozitív képet is csináltam, szám szerint 50 pozitív és 150 negatív képet, majd pedig ezeket különféle módon transzformáltam (elforgatás, kicsinyítés, nagyítás). A tanítás több iterációból áll, és mindegyik végén elkészül egy modell, ami használható. Az más kérdés, hogy ennek a felismerési rátája milyen, viszont ezzel már lehet tesztelni. A 7. ábrán például egy ilyen teszt látható. A környezet és a kocka is igen sötét, ennek ellenére felismeri a modell. Az ezt követő ábrán már sajnos nem csak a kockát, hanem a kukát is felismeri a modell. Ez az eredmény 5 napnyi tanulás eredménye, így valószínűleg ez nem lesz jobb sem.



**7. ábra** Haar Cascade modell felismerése



**8. ábra** Rossz felismerés



**9. ábra** További felismerések

## 4.3 Neurális háló betanítása

A következő módszer amit vizsgáltam az a neurális hálók voltak. Ez egy igen nagy témakör, lehet hálót teljesen a nulláról betanítani, vagy már létező hálót újratanítani. Erre egy jó lehetőséget nyújt a TensorFlow és ennek az Object Detection API-ja. Rengeteg olyan háló található itt, amik a 2017-es Coco adathalmazon tanultak, ami egy általános objektumokat tartalmazó halmaz, például autó, kutya, macska stb. Ez a megközelítés azért is jó, mert a háló véletlenül se fogja felismerni a kockát egy embernek, vagy olyan objektumnak amire be van tanítva. Az újratanításhoz itt is, mint a Haar Cascade-nál is, képekből álló adathalmazt kell használni.

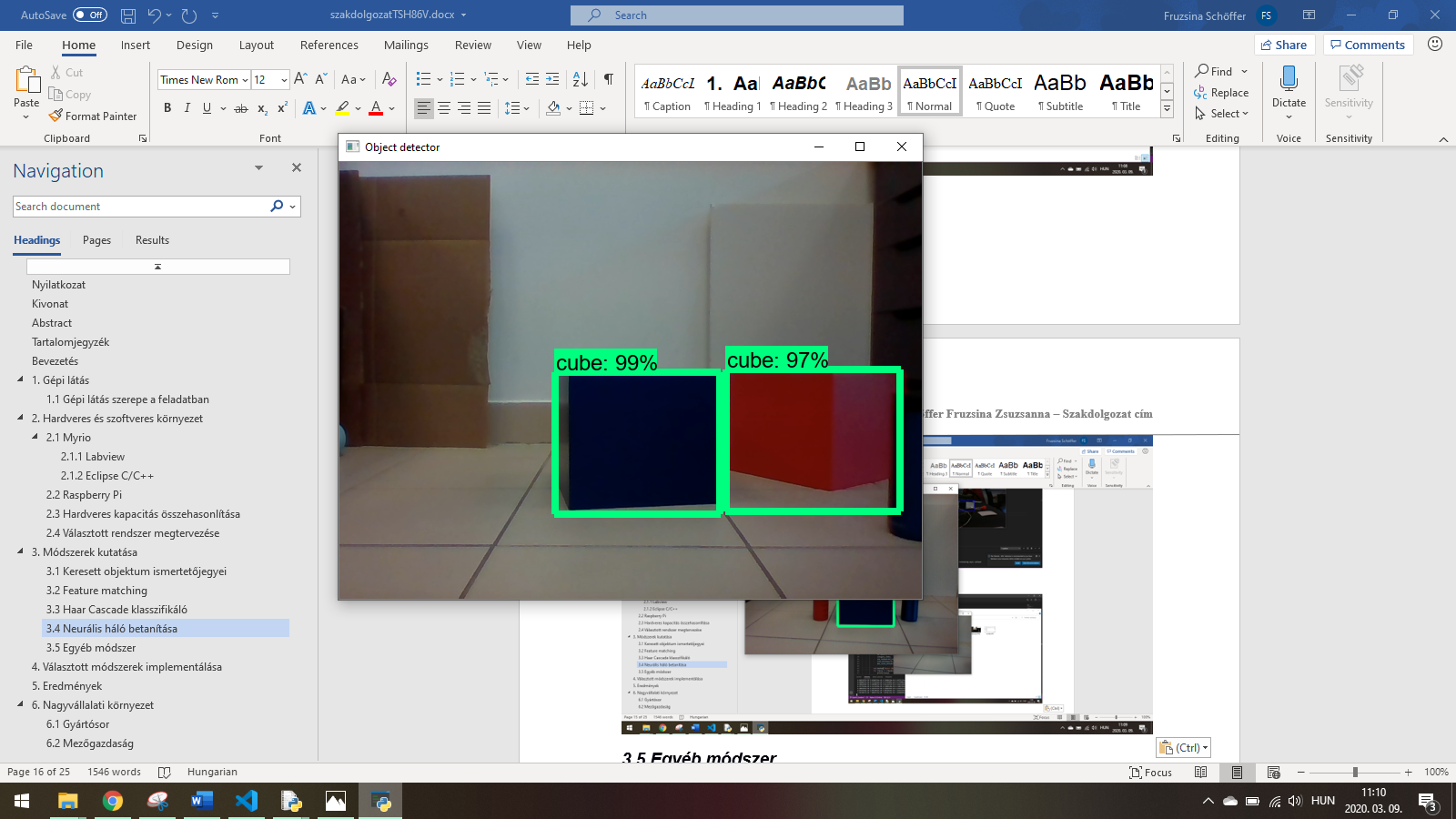
A halmaz elkészítése után az ebben található képeket fel kellett címkézni, ki kellett választani milyen modellt fogunk használni, és ebbe bele kellett kalkulálni azt is, hogy az egész egy Raspberry Pi-n fog futni, tehát nem szabad nagy erőforrásigényűnek lennie. A választásom az SSD Mobilenet v1 COCO volt. Az SSD a Single Shot Detectort jelenti, ami azt jelenti, hogy egyetlen hálóról van szó, ami a felismerést végzi. Éppen ezért ez a modell gyors, bár nem annyira pontos. [6][7]

A következő lépés a neurális háló tanulásának a beállításai. Ilyen beállítás például az úgynevezett batch size, vagyis, hogy egy tanulási iteráció alatt mennyi példán tanuljon a modell. Fontos még az API használata esetén az is, hogy a label map fájlt ki kell egészíteni azokkal az objektumokkal, amiket fel akarunk ismerni, hiszen ezeket a címkéket fogja majd használni.

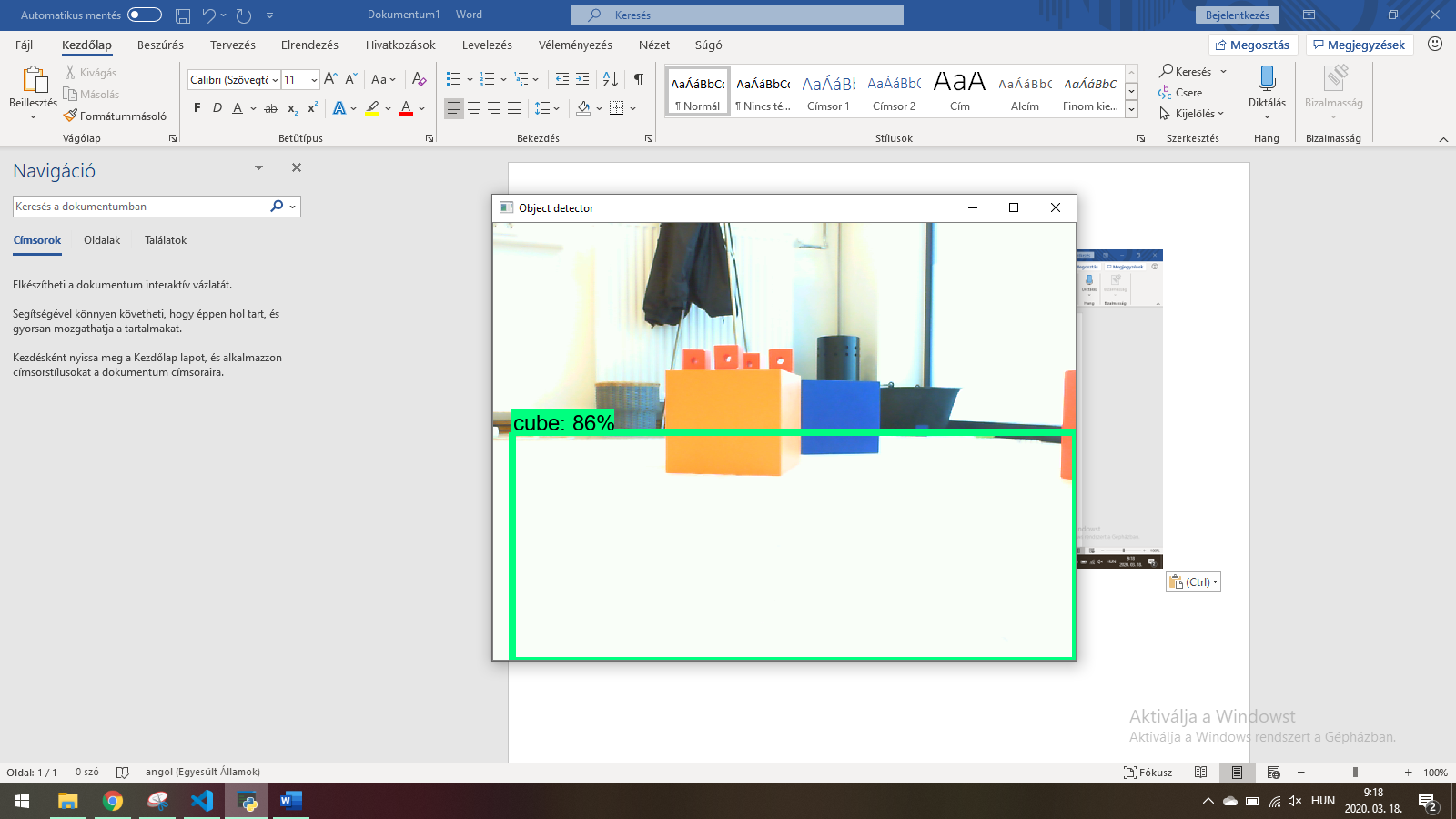
A betanítás leállítását hozzá lehet rendelni különböző határértékekhez, például a veszteség értékéhez is, de ezt nem állítottam be, hanem felügyeltem mikortól nincsenek már jobb értékek. Ez az érték, az angol nevén a loss value, és azt jelzi, hogy mennyire pontos a predikciója a hálónak. Ha egy idő után ez már nem csökken, akkor fölösleges ezt tovább tanítani, mert akkor más problémákba futhatunk. [11]

Az általam végzett betanítás 5 napig tartott, és ez idő alatt folyamatosan csökkent is ez az érték, majd miután egy bizonyos értéken stagnált az érték, leállítottam a betanítást.

A következő ábrán látható például egy felismerés, ahol egész jól felismeri a kockákat, de az ezutáni képen pedig már nem, és itt is hamis pozitív felismerések vannak.



**10. ábra** Saját betanított neurális háló felismerés



**11. ábra** Neurális háló rossz felismerés

## 4.4 OpenCV képfeldolgozás

Az utolsó módszer az feladat implementálása OpenCV osztálykönyvtár segítségével. Ebben a lehetőségben teljesen feladatra tudjuk szabni a dolgokat.

Ebben az esetben először a képeket a konkrét színekre kell szűrni, ami azt fogja eredményezni, hogy a konkrét kockák maradnak már csak a képen. Ez alapján pedig ezekből kiindulva lehet végezni különböző méréseket.

Bár ez egyszerűnek hangzik, az első probléma máris a színek felismerésénél van. A kockák RGB színei ismertek, így logikus ez alapján szűrni, viszont változó fényviszonyoknál (ha például fordul a robot, és esetleg valahol egy lámpa világít be, vagy egy ablak van ott), ez a szűrés már nem megbízható. A képfeldolgozás egyik nagy része a színtér. Ez a színtér gyakorlatilag egy térbeli koordinátarendszer, ahol a színek koordinátákkal vannak kifejezve. Az RGB színtér csakis a vörös, zöld és a kék színeket veszi figyelembe, így a fényesség változását nem tudja ez a színtér ezt figyelembe venni.

Létezik szerencsére több színtér is, például a HSV, ami a kép intenzitását elválasztja a konkrét szín információktól, így változó fényviszonyoknál, vagyis sötétebb és világosabb képeknél is képes a helyes felismerésre.

A HSV a Hue (színárnyalat), Saturation (telítettség) és Value (érték) rövidítése. A Hue 0-360 közötti értéket vesz fel, és az aktuális szín mértékét mutatja. A Saturation a színben található szürkének a száazalékos arányát adja meg, ez egy százalékos érték. A Value fogja megadni a világosság értékét, ami azt mutatja, hogy a vizsgált szín mekkora intenzitással látható.

A leírtak alapján a HSV megoldást választottam, hiszen az a változó fényviszonyokban is jól tud teljesíteni.

Az implementáció során a beolvasott képkockákat át kell alakítani HSV színtérbe, majd pedig ezen végezni a szűrést a konkrét HSV színkódokra.

A szűréshez maszk vagy ablak alapú transzformációra kell gondolni, ahol a képkocka egyes pixeleit, az ezt körülvevő pixelek alapján módosítjuk, vagyis transzformáljuk. Ezt végezhetjük lineáris vagy nemlineáris maszkkal. Az első egyenlet mutatja be ezt. Itt lesz egy input mátrix, egy maszkolási mátrix, vagyis konvolúciós maszk, és egy eredmény mátrix. Az eredmény mátrixon látszik az a pixel, amit transzformáltunk.

A feladathoz morfológiai szűrést használok, és azon belül is a nyitást. Ez azt jelenti, hogy a képkocka először erodálva, majd dilatálva lesz. [8] [9][10]

(1)

Azért fontos, hogy ez legyen használva, mert ennek az első lépése az erodálás, amivel el tudjuk tüntetni a zajokat is, és ezután dilatálunk. A szűrés után megmaradó képen ezután kontúrokat keresünk a Douglas-Peucker algoritmussal, ami a megmaradt objektum kontúrjait adja meg.

Az algoritmus működésének alapja, hogy rekurzívan vizsgálja a pixelen látható pontokat, és ezekből görbéket képez le, mindig kevesebb ponttal. A kapott pontok számát, illetve a kontúrok alapján kapott területet veszem ezek után figyelembe, és minél nagyobb a terület, annál nagyobb a valószínűsége, hogy egy kockát látunk, illetve a kontúrok számának is 4-nek kell lennie.

A minimális érték, ahol egy nagy kockáról beszélhetünk, nem pedig egy kicsiről, valahol egy 3000 körüli értéknél található, amihez egy 150-es értékű eltérést is megengedek.

Mielőtt viszont ezekkel a kontúrokkal továbbdolgoznánk, még vannak olyan vizsgálatok, amiket el tudunk végezni ahhoz, hogy biztosra menjünk abban, hogy valóban kockát lát-e a kamera. Ilyen például a szoliditás, ami a kapott terület arányos értékét mutatja meg a konvex terület értékéhez képest. A következő ilyen érték a képarány, ami a magasság és a szélességét arányát veszi figyelembe, majd pedig a távolság is, mivel a pálya szélessége miatt a kocka és a robot közti távolság nem lehet végtelen nagy.

Ezeknek az értékeknek a határai a (2) egyenletben találhatók.

(2)

Egyiknek sem kell tökéletes értékekkel rendelkeznie, mindegyiknél van egy bizonyos hibahatár, hiszen eltérések lehetnek.

# Választott módszer implementálása

Az előzőekben bemutatásra került az összes lehetséges megoldás. Látható volt, hogy egyik gépi tanulási modell sem teljesített a legjobban, így végső soron a választás az OpenCV-vel való implementálásra esett, hiszen ezt lehet a legjobban feladatra is szabni.

Mielőtt rátérnék a konkrét implementációkra, először szeretném bemutatni a konkrét modulokat és ezeknek az egymással való együttműködését.

Lesz egy belépési pontja a rendszernek, ahonnan a többi modul hívásra fog kerülni, úgy, hogy teljesüljön a feladat. Ebben a fájlban lesz egy szál az ultrahangszenzoroknak, és egy másik szál pedig a kamerának, ahol folyamatosan fogja beolvasni a képeket. Ezen a szálon fog futni a FrameProcessor osztálynak a getFrames() metódusa, ami beolvassa a képet. Lényeges, hogy a beolvasás külön szálon történjen, hiszen így ennek a beolvasása nem fogja blokkolni a fő szálon történő feldolgozást.

Ezután jön a CubeFinder osztály, amiben a kockák felismerését szolgáló metódusok találhatóak meg.

Vannak még úgynevezett segítőmodulok is, amik a kép feldolgozását segítik, például a színtér átalakítást is.

Részletesebben csakis a kockakereséshez fűződő modulokat fogom bemutatni, említés szinten pedig a QR-kód beolvasásához szükséges modult, amiben nincs különösebb logika, csupán a kód beolvasása a pyzbar csomaggal, majd ennek a dekódolása.

## 5.1 Kockakereső modulok

Ebbe a modulba tartoznak azok a metódusok, amik a kocka felismerését és az ebből kinyert paraméterekkel való munkát végzik. Ezen belül található egy további modul, amibe a konkrét képfeldolgozó metódusok vannak, illetve a kocka kereséért felelős osztály.

Ez a két modul együttműködik egymással, és mindezek mellett további modulokból hívnak meg metódusokat, amik például a mintavételezésért, vagy az átlagolásért felel.

Rövid bemutatása a lefutásnak a következő: a **getMorphedFrame()**

Metódust a HelperCalculations modulban levő **getAvgDistance()** függvény hívja meg, és az ebből kapott képkocka a CubeFinder osztály **getParametrizedCubeDistance()** nevű függvényének kerül átadására. Ez az bejövő szín alapján meghívja a **measureCubeDistance()** függvényt, ami visszaadja a kontúrokat. Ezeket kapja meg a **calculateCubeDistance()**, ahol ugyanakkor ellenőrzésre kerül a képarány, szoliditás és a távolság, és ezek után pedig ki lesz számítva a távolság a a **calculateDistanceFromHeight()** függvénnyel.

### 5.1.1 Képfeldolgozó modul

Azok a függvények, amik a képfeldolgozásért felelősek a PictureProcess modulban találhatóak meg. Itt van például a morfológiai műveleteket végző **findContours()** függvény is. Ez a HSV színtérbe alakított képkockát kapja meg, majd egy nyitási művelet után a kontúrok területének a mérete alapján visszaadja a legnagyobb 10-et. Ez lesz továbbadva a CubeFinder osztálynak.

1. ***Kódrészlet*** Függvény a kontúrkeresésre

def findContours(hsv,colorLow,colorHigh):

kernel = np.ones((6,6),np.uint8)

mask = cv2.inRange(hsv,colorLow, colorHigh)

mask=cv2.morphologyEx(mask,cv2.MORPH\_OPEN,kernel)

contours = cv2.findContours(mask.copy(), cv2.RETR\_EXTERNAL,cv2.CHAIN\_APPROX\_SIMPLE)

contours = imutils.grab\_contours(contours)

contours=sorted(contours, key=cv2.contourArea, reverse=True) [:10]

return contours

Ugyanezt a függvényt használja a **getCoordinatesFromContours()** függvény is, ami ebből kiszámolja a robot pozícióját. A függvény végig iterál a kontúrokon poligonokat keresve, majd a talált poligon sarkainak a koordinátáit fogja visszaadni. Ez látható a 2. kódrészletben.

1. ***Kódrészlet*** Koordináták számítása

**def getCoordinatesFromContours(hsv,colorLow,colorHigh):**

**contours=** **findContours(hsv,colorLow,colorHigh)**

**for c in contours:**

**approx=cv2.approxPolyDP(c, 0.05 \* cv2.arcLength(c, True), True)**

**if (cv2.contourArea(c)>=4500):**

**n = approx.ravel()**

**return n**

**else:**

**return 0**

További segítőfüggvények is találhatók még itt, mint a **checkSolidity()**és az **isObjectCube().**

### 5.1.2 Paraméterszámító függvények

Vannak még a paraméterszámító metódusok, amik az ismert tulajdonságaik ismeretében, a kockák olyan értékeit számítják ki mint a távolság, segédtávolság vagy az orientáció. Az orientáció azt mutatja meg, hogy takarja-e valami az adott kockát.

Az egyik legfontosabb paraméter a távolság, amihez a kamera fókusztávolsága is szükséges. Mivel ezt a használt kamerához nem találtam semmilyen útmutatónan, így az alakzatok hasonlóságának tételével lehet ezt meghatározni. Ez azt jelenti, hogy két alakzat nagyítási és egybevásági transzormációval egymásba vihető. Ha ismerjük a pixelméretet, a kamera és a kocka közötti távolságot, és a kocka méretét, akkor megkapjuk a fókusztávolságot. Ezt mutatja be az alábbi kifejezés is. Ezt csakis egyszer kell kiszámolni, és utána már kereshetjük a távolságra hozva a konkrét távolságot.

(3)

A távolságot mindig a kocka magasságából számítom ki, mivel a szélessége esetén lehet arról is szó, hogy kilóg a kocka a képből, de nagyon távol van. Ezzel ellentétben a magasság esetében ha olyan közel kerül a robot a kockához, azt a szenzorok is érzékelik, így nem lehet szerencsére összekeverni rossz méréssel. Ezt a számítást a **calculateDistanceFromHeight()** függvény végzi, ami a 3. kódrészletben látható.

1. ***Kódrészlet*** Távolságszámító függvény

**@staticmethod**

**def calculateDistanceFromHeight(height):**

**return (20\*580)/height**

Az előzőekben szó volt a segédtávolságról is, amit a **calculateLengthToGo()** függvény végez. Ez vár egy színértéket, x és w értéket, illetve távolságot.

A 320-as érték a kapott kép fele, az x a képen látható kocka bal felső pontjának X koordinátáját jelöli, a w pedig a szélességet, vagyis, hogy mekkora a kocka x tengelyen való szélessége. A kép felének értékéből kivonva a szélesség és az X koordináta összegét, majd megszorozva a robot kockától való távolsága és a fókusztávolság hányadosával, egy olyan értéket kapunk, ami közelítőleg jó értéket ad. Mivel a közelítőleg jó érték még mindig nem a legpontosabb, hozzáadva viszont a kocka szélességének a felét, ±0,5 cm-es pontosságú másodlagos távolságot kapunk. A megvalósítást a *4. kódrészlet* mutatja be.

1. ***Kódrészlet*** Másodlagos távolságszámító függvény

**@staticmethod**

**def calculateLengthToGo(color,x,w,distance):**

**togoDistance=((distance/580)\*(320-(x+w)))+10**

A modulban továbbá olyan metódusok vannak, amikkel a kis kockák értékeit lehet még kiszámolni, hasonlóan mint az előzőekben bemutatottak alapján.

### 5.1.3 Segítőszámító modulok

A segítőszámító modulok olyan metódusokat jelentenek, amik az előzőleg is bemutatott értékek átlagolását végzik. Erre azért van szükség, mert bármilyen mintavételezés során megeshet, hogy egy adat korrupt lesz, és ez alapján nem szabadna dönteni.

## 5.2 Kockára pozicionáló modulok

A kockára pozicionáló modulokban olyan függvények kerültek implementálásra, amik a megfelelő ráállást segítik a robotnak. Ilyen például, ha rossz szögből közelíti meg a robot a kockát, vagy esetleg nem pontosan előtte áll, és kamerakép alapján úgy kell ráállnia, hogy az ultrahangszenzor ne mérjen el mellette és pontosítani tudjon.

Itt nem feltétlen megfelelő a HSV színtér, mivel a kockának az oldallapjai nem teljesen különíthetőek el, ami gondot okozhat. Megoldás erre, hogy a kocka legalsó éleit vesszük figyelembe, hiszen azok megkülönböztethetőek a pályától, ami fehér. Ha a fehér színre maszkolunk, akkor annak a kontúrjaiból mérést tudunk végezni. A kontúrnak meg kell találni a legalsó koordinátáját, erre egy egyenest fektetni, majd megvizsgálni, hogy ez az egyenes párhuzamos-e az x tengellyel. Amennyiben nem, úgy feltételezhető, hogy a robot a kockát ferdén, rossz szögből látja.

Ezt valósítja meg a következő pár metódus A

Ennek a megvalósításában az első függvény ami szerepet játszik, az a kontúrok megszerzéséért felelős **getCoordinates()** nevű függvény a FrameProcessor nevű osztályban. A **findBiggestPoint()** nevű függvény megkeresi a legnagyobb értékű Y koordinátákat és eltárolja az ehhez tartozó pontokat is, illetve az előtte való pont koordinátáit is. Ez látható az 5. kódrészletben.

1. ***Kódrészlet*** Legnagyobb koordináta megtalálása

**def findBiggestPoint():**

**points=** **FrameProcessor.getCoordinates()**

**maxcoordY=coordx=coordy=coordbx=coordby=j=0**

**try:**

**if (points.any()!=None):**

**for i in points:**

**if (j%2==1):**

**if points[j]>maxcoordY:**

**coordx=points[j-1]**

**coordy=points[j]**

**coordby=points[j-2]**

**coordbx=points[j-3]**

**maxcoordY=points[j]**

**j=j+1**

**return (coordx,coordy,coordbx,coordby)**

**else:**

**return None**

**except:**

**return None**

Az ezek után megkapott pontokból a **makeVector()** nevű függvény segítségével elkészítjük a 2 pont által kapott vonal vektorát, illetve az x tengellyel párhuzamos vektort is elkészítjük. Ezek után a két vektorból egységvektorokat készítve meghatározhatjuk a kettő által bezárt szöget. Mivel a két vektor egységvektor, így ezek skalárszorzata az ezek által bezárt szög koszinuszát adja meg, aminek a számítását a (4)-es kifejezés, ennek átalakítását pedig a (5)-ös kifejezés valósítja meg.

(4)

(5)

Ahol a és b vektorok, α pedig az általuk bezárt szög. A megvalósítása az előzőekben leírtaknak a 6. kódrészletben látható.

1. ***Kódrészlet*** Vektorok számítása

**def getAngleBetweenVectors(p1, p2, p3, p4):**

**vector1=makeVector(p1,p2)**

**vector2=makeVector(p3,p4)**

**unitVector1 = unitVector(vector1)**

**unitVector2 = unitVector(vector2)**

**return np.arccos(np.clip(np.dot(unitVector1, unitVector2), -1.0, 1.0))**

**def makeVector(p1,p2):**

**return (p1[0]-p2[0],p1[1]-p2[1])**

**def unitVector(vector):**

**return vector / np.linalg.norm(vector)**

Az értékkel ezek után nem dolgozunk tovább, hanem a mozgás során eltárolt rotációk értékéből szűrjük le, hogy mennyit kell korrigálnia a robotnak.

Ezek után számoljuk az orientációt, ami igazából annyit ad meg, hogy a kocka a robot melyik oldalán van, vagyis melyik irányba kell fordulnia a robotnak. Ennek a megvalósítása látható a 7. kódrészletben.

1. ***Kódrészlet*** Pozíciómeghatározás

**def getPositionString(x,y,w,h):**

**position=""**

**imgFourth=int(IM\_WIDTH/4)**

**fullWidth=x+w**

**if x>=0 and x<=imgFourth and fullWidth<=imgFourth\*3:**

**position="left"**

**elif x>=0 and x<=imgFourth and fullWidth>imgFourth and fullWidth<=imgFourth\*2:**

**position="left"**

**elif x>=0 and x<=imgFourth\*2 and fullWidth>imgFourth\*2 and fullWidth<=imgFourth\*3:**

**position="left"**

**elif x>=imgFourth\*2 and fullWidth>imgFourth\*2:**

**position="right"**

**elif x>=imgFourth and x<=imgFourth\*2 and fullWidth>(imgFourth\*3):**

**position="right"**

**else:**

**position="center"**

**return position**

A kapott orientáció segítségével ezek után meghatározzuk azt, hogy melyik irányba kell elfordulni, vagyis mínusz vagy plusz 90°-ot, illetve a másodlagos távolság kiszámításával az elfordulás után az adott távolságnyi értéket fogja megtenni a robot. Ezután az éppen elfordult érték ellentetjét újból visszafordulja, majd újból az előző méréseket kezdi ismételni, és az alapján mozogni. Ez a műveletsor egy while ciklusban játszódik le, aminek a kilépési feltétele, hogy a másodlagos távolság kisebb legyen mint 2 cm. Szerencsére a pontos számítások alapján ez több lefutás közben is általában 2-3 iterációt vett igénybe.

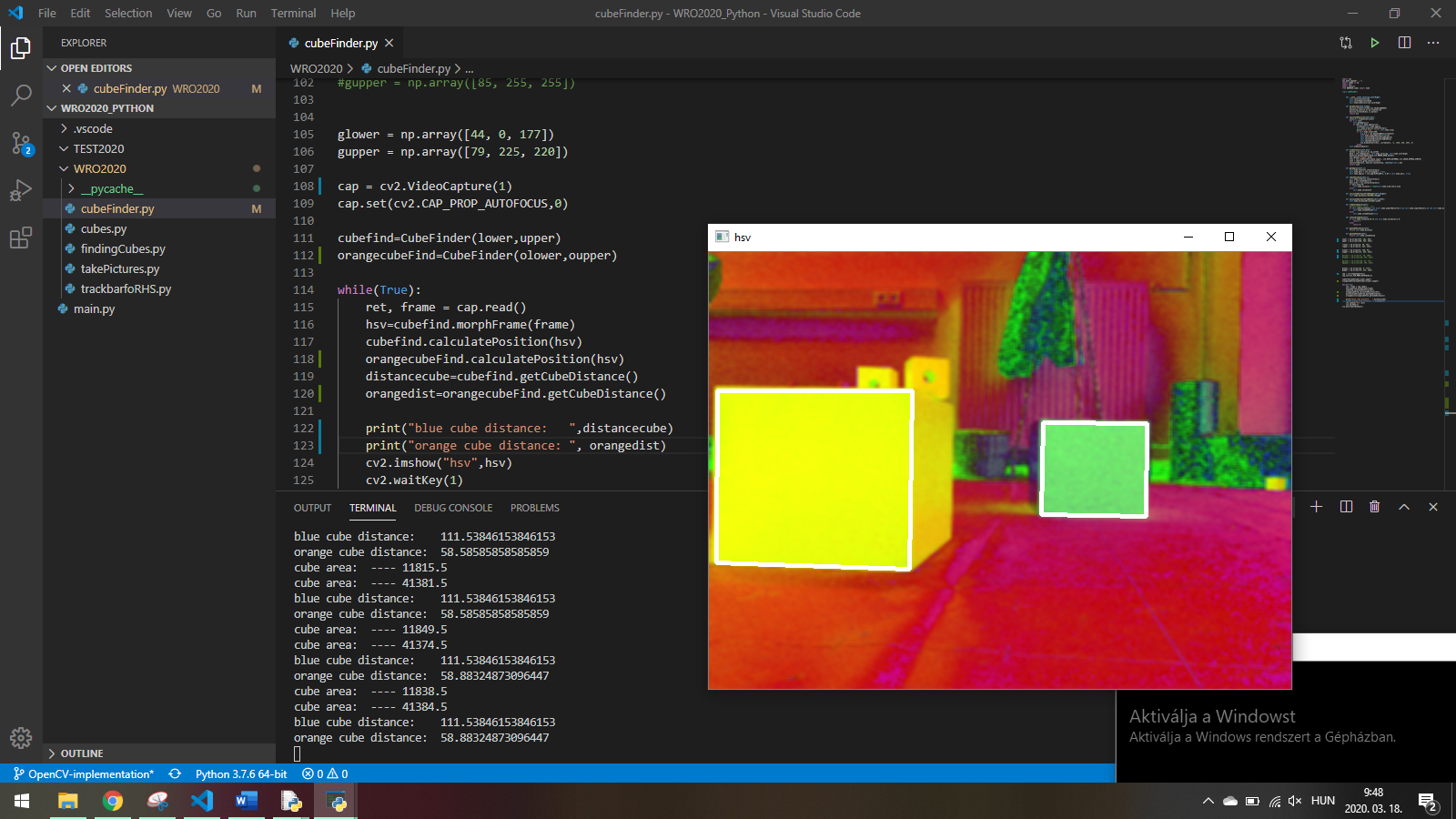
Az előzőben leírt műveletsor elvégzése után, tehát a robot tökéletesen a kocka előtt van párhuzamosan, és innentől tudja használni az ultrahangszenzorokat is.

A fejezetben bemutatott függvények segítik tehát a robot kockának való megközelítését ahonnan ezentúl már csak le kell emelni a kis kockát, és utána újból következhet majdnem ugyanaz a műveletsor, ami eddig is történt.

Miután az összes implementáció elkészült, lehet implementálni a versenykört, ahol a robot lefutásának képe alapján valósul ez meg, a különböző modulok metódusainak meghívásával.

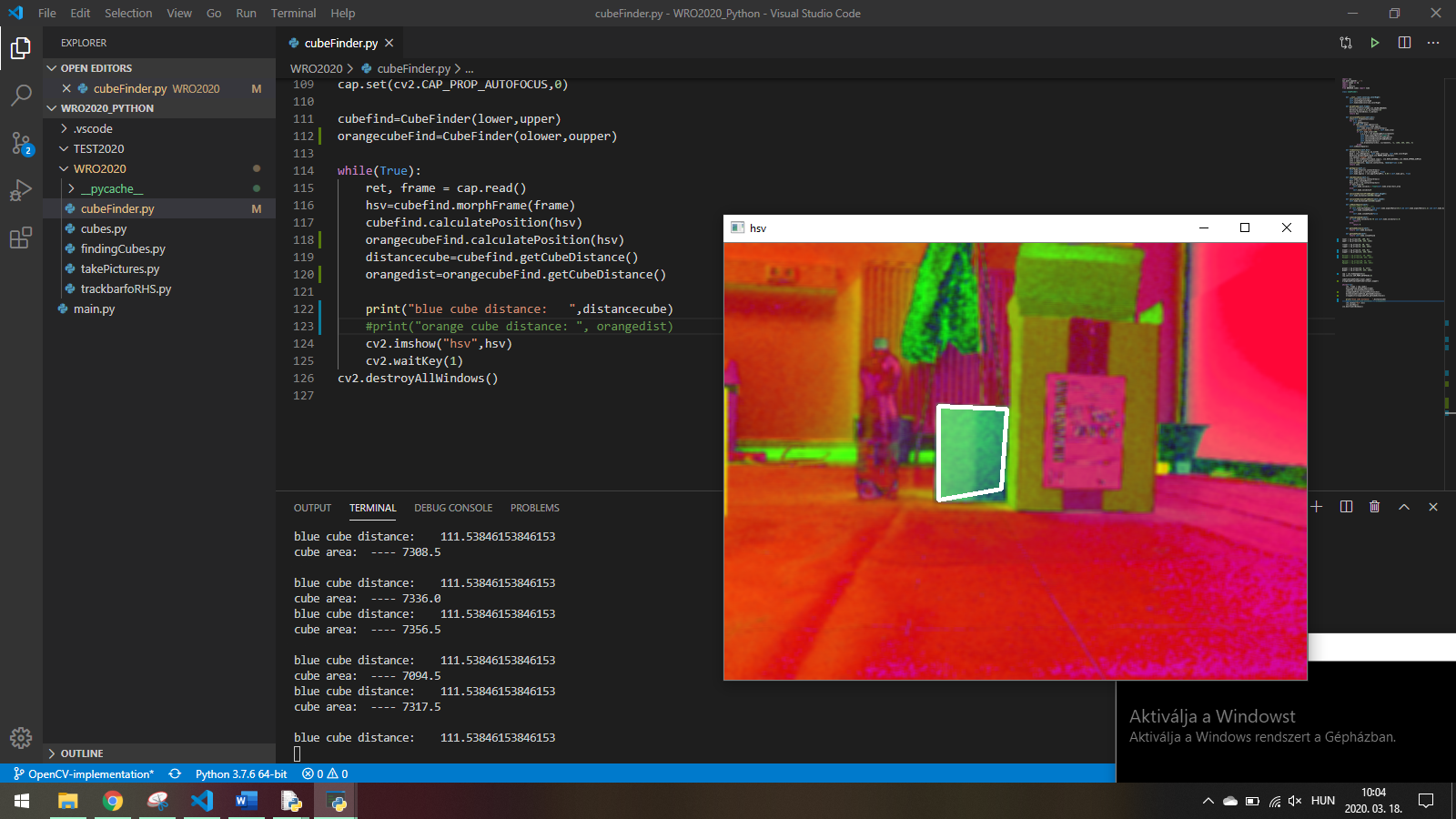
## 5.3 Implementáció bemutatása

A következőkben pár példát hozok arról, hogyan is ismeri fel a modul a kockákat a konkrét pályán. A 12. ábrán egy narancssárga és egy kék kocka látható HSV színtérben, és a bal oldalon pedig ezeknek a távolsága egy tesztmodullal.



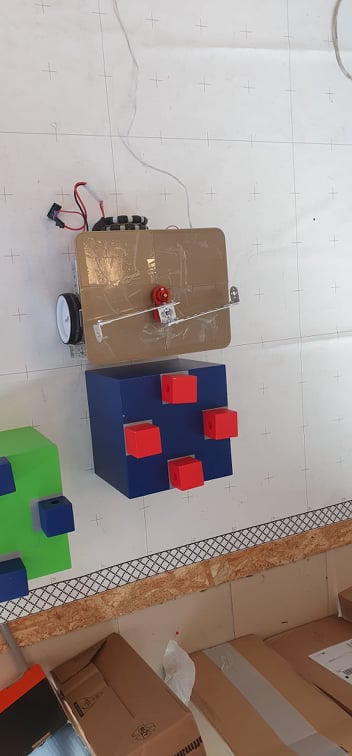
**12. ábra** Kockák felismerése

A 13. ábrán a kék kockát próbáljuk megtalálni, és itt is látni, hogy bár a kocka el van fedve félig, mégis sikerül egy jó távolságot kiszámítania a gépi látás modulnak.



**13. ábra** Részleges kék kocka felismerése

A 14-es ábra azt mutatja be, hogyan áll a robot konkrétan a kocka elé egy konkrét lefutásnál.



**14. ábra** Rápozícionálás a kockára

# Összegzés

Az előzőekben bemutatásra került tehát egy olyan gépi látás modul, ami egy robot autonóm működését segíti.

Rengeteg különböző implementáció létezik az ilyen rendszerekhez, amikhez nyúlni lehet, de egyik se fog tudni olyan jól teljesíteni, mint egy feladatra specializált implementáció. A gépi látás témája rengeteg dolgot rejt magában, és a bemutatottak alapján látható az is, hogy nagyon sok olyan kis dolog van, ami egyes számolásokat eltorzíthat.

Az informatika gyors fejlődése miatt a jövőben valószínűleg még több olyan algoritmussal fogunk találkozni, amik segítik a gépi látást, és nem elhanyagolható a mesterséges intelligencia szerepe sem. Bár a dolgozatban ez nem teljesített jól, ez nem a technológia, hanem inkább a hiányos és nem minőségi betanítási adathalmazhoz köthető.

A jövőben éppen ezért valószínűleg ezt a feladatot is másféle módon is meg lehet majd valósítani, viszont addig is látható, hogy a már létező algoritmusok is jól teljesítenek.

# Irodalomjegyzék

[1] T. S. Huang: *Computer Vision: Evolution and Promise*

CERN School of computing ; 8 ; 21-26, 1996.

[2] Shaharyar Ahmed Khan Tareen (, Zahra Saleem): *A Comparative Analysis of SIFT, SURF, KAZE, AKAZE, ORB, and BRISK  
 2018 International Conference on Computing, Mathematics and Engineering Technologies* (iCoMET), Sukkur, pp. 1-10, 2018.

[3] Lester Kalms (, Khaled Mohamed): *Accelerated Embedded AKAZE Feature Detection Algorithm on FPGA*

*HEART2017: Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies* pp. 1–6, 2017*.*

[4] Awad Ali Ismail (, Hassaballah, M): *Image Feature Detectors and Descriptors; Foundations and Applications*

Springer International Publishing, Verlag, 2016

[5] Adam Schmidt (, Andrzej Kasiński): *The Performance of the Haar Cascade Classifiers Applied to the Face and Eyes Detection*

*Computer Recognition Systems 2 (pp.816-823), Heidelberg, 2007.*

[6] Debojit Biswas (, Hongbo Su): *An automatic traffic density estimation using Single Shot Detection (SSD) and MobileNet-SSD* Physics and Chemistry of the Earth, Parts A/B/C, 110, pp. 176-184, 2019.

[7] James M, (, Derong Liu) *Multilayer Neural Networks and Backpropagation,* IEEE, 2016.

[8] Michael W. Schwarz (, William B. Cowan): *An experimental comparison of RGB, YIQ, LAB, HSV, and opponent color models* ACM Transactions on Graphics, 6, pp. 123-158, 1987.

[9] A. Rebhi (, S. Abid ): *Fabric defect detection using local homogeneity and morphological image processing* International Image Processing, Applications and Systems (IPAS), Hammamet, pp. 1-5, 2016.

[10] Srisha Ravi (, Khan, Am): *Morphological Operations for Image Processing : Understanding and its Applications.* National Conference on VLSI, Signal processing & Communications, Guntur, 2013.

[11] Powers David : *Evaluation: From Precision, Recall and F-Measure to ROC, Informedness, Markedness & Correlation* Journal of Machine Learning Technologies, 2, pp. 37-63, 2011.