GÉPI LÁTÁS DOKUMENTÁCIÓ

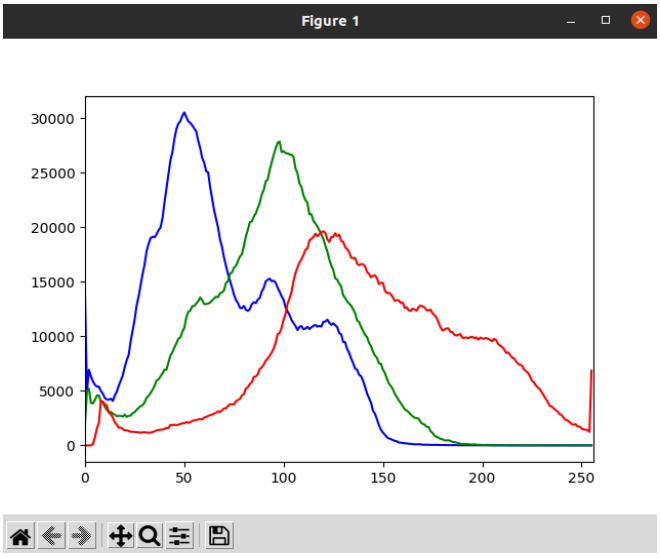
BEVEZETÉS

A biztonsági kamerák használata elterjedt, mind az iparban, mind a hétköznapi életben. A különböző használati esetekben a közös pont az, hogy a biztonság növelhető. Az esetleges lopások, betörések, illetéktelen belépések és sok más büntethető, vagy éppen nem kívánatos eset felvételre vehető, vagy éppen elhárítható. A videófelvételen a mozgás felismerése kritikus szempont. A választott feladatban éppen ezzel a problémával foglalkozom, a felfedezett mozgás nyomon követesével kiegészítve. Az eredeti feladatban nem volt meghatározva a mozgó objektumok száma, emiatt a fejlesztendő program egyszerre több mozgó objektumot is tud detektálni. A választott programozási nyelv a Python volt, ebből is a 3.8.10-es verzió. A használt külső könyvtárok pedig a következőek voltak: OpenCV 4.2.0, numpy 1.21.4. A fejlesztést pedig az Ubuntu 20.04-es verzión végeztem. A tesztelésnél használt videóanyagok legnagyobb részét saját magam vettem fel. Ide azt is hozzá kell tenni, hogy az alább bemutatott algoritmus csak fixált pozíciójú kamera felvételek esetén fog elvárt módon működni, nincs felkészítve a kamera mozgására, erre majd a tesztesetekben fogok is megerősítő példákat bemutatni.

MEGOLDÁSHOZ SZÜKSÉGES ELMÉLET

Digitális kép felépítése

A látható fénytartomány adja meg az emberi szem számára a színskálát. Ezt a színskálát valahogyan értelmezni kellett a számítógépek világában is. Erre az egyik módszer az RGB színcsatornák használata, természetesen ezen kívül van más megközelítés is. (HSV) Ahhoz, hogy megértsük hogyan kezel egy színes képet a számítógép azt kell tudnunk, hogy egy digitális kép, mint minden információ az informatika világában kis adategységekből épülnek fel. Ezt az informatikai adategységet nevezzük bitnek, 8 bit pedig egy bájt. Egy bájt pedig 2^8 (256) féle számot tárolhat. A színek reprezentálása az RGB színskála szerint van értelmezve ebben a probléma megoldásban. Az egyes színek értéke a benne található piros (red), zöld (green), kék (blue) alapszínek mennyiségétől függ. Egy-egy szín egy színcsatornát jelent. Az egyes színcsatornák értéke pedig 0-255 ig terjed. A 0 a legalacsonyabb míg a 255 a legmagasabb intenzitást jelenti. A színcsatornáknál egy színcsatorna, egy bájtot jelent. Emiatt, ha egy RGB színkód (0,0,0) akkor fehér színt lát az ember, míg (255,255,255), akkor feketét. Itt fontos megjegyezni, hogy az emberi szem korlátai miatt, ha az előbbi fekete értéket (254,254,254)-re változtatjuk, akkor számunkra ránézésre nem lesz látható és érzékelhető különbség, viszont egy gépi algoritmus esetén természetesen lesz. A videófelvételeket képkockánként lépegetve fogom az algoritmusnak táplálni, ezért a videót digitális képek sorozatának is fel lehet fogni. Az egyes képek pedig pixelekből (magyarul: képpont) épülnek fel. A képpontok értékei pedig a nekik megfelelő RGB színkód értéke. A megoldásban hisztogram használata is szerepel, hiszen a küszöbérték kereső függvény. A hisztogram egy matematikai függvény, amely itt a képben található színcsatornákhoz megadja, mennyi képpont található abból a képben. A képen a teszt videó „levelek1” hisztogramja látható.



MEGVALÓSÍTÁS TERVE ÉS KIVITELEZÉSE

A megvalósítás vázlatos terve két fő részből áll. Magából a mozgás detektáló algoritmusból, valamint a háttér kinyerése. Az első rész fő elemei a következőek: szürkeárnyalatos konverzió, abszolút differencia számítás, zajszűrés, küszöbértékelés, nyújtás(?), kontúrok keresése majd téglalapok rajzolása a detektált mozgó objektumok köré. A háttér kinyerését pedig a következő bekezdésben fejteném ki részletesen.

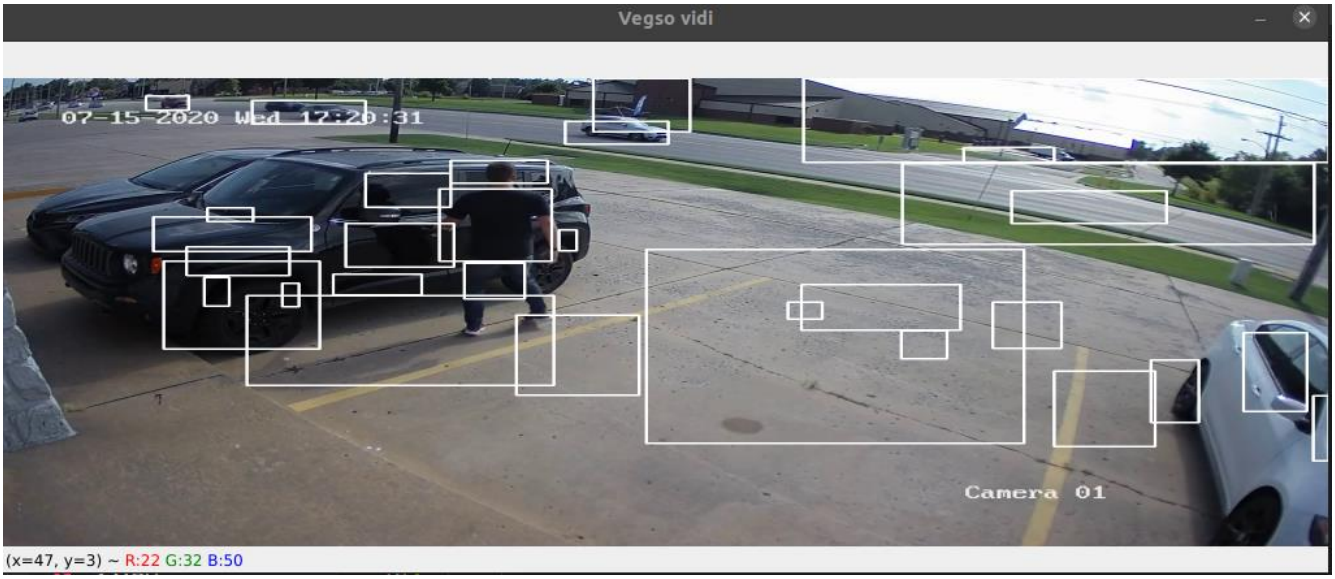
Háttér kinyerése

A *cv2. CAP_PROP_FRAME_COUNT()* függvénnyel kinyerjük a videó fájlban lévő képkockák számát. Ebből a numpy *random.uniform()* függvény segítségével mintát veszünk, a mintavétel nagyságát a *size* argumentummal tudjuk megadni. Ebből a 30 darab képkockából majd egy képet állítunk elő. Az így megkapott képen pedig medián számítást végzünk. A numpy a mediánt az adott tengely mentén számolja. A több dimenziós tömbből egy egydimenziós tömböt képez. A mediánt a következő matematikai képlettel számítja:

$median = n + 1/2$ ha páros az elemszám.

$median = \frac{\left(\frac{n}{2}\right)+\left(\frac{n}{2}\right)+1}{2}$ ha páratlan az elemszám.

Háttér sikeres kinyerése és medián számolása



a „barik1” tesztesetből.

Szürkeárnyalat konverzió

Az eredetileg színes képeket egy *cv2.cvtColor()* függvény segítségével szürkeárnyalatos képpé konvertálom.

$$J(x,y) = 0.299 \times I_r(x,y) + 0.587\, I_g(x,y) + 0.114\, I_b(x,y)$$

Erre egyrészt a függvények számítási segítsége miatt van szükség, hiszen nem 3 csatornán, hanem 2 csatornán lesz csak adat. Emiatt a számítási idők lényegesen rövidülnek. Valamint azért is szükséges a ebben a probléma megoldásban a konverzió, mert a szín mint információ, nincs hatással a mozgásokra, irreleváns információ az esetünkben, hogy a labda ami pattog az piros e vagy kék, most a pattogás és annak nyomonkövetése a szempont. A fény intenzitás változás a mozgás detektálás alapja az algoritmusban, így egy szürkeárnyalatos képpel tovább dolgozni előnyös.

A képen a szürkeárnyalatos kép látható.

Abszolút differencia számítás

A következő lépés a kiszámolt medián háttér és a jelenlegi képkocka egymásból való kivonása. Erre is van többféle módszer, hiszen ki lehet futni a tartományból mind az összeadásnál, mind a kivonásnál. Az általam választott függvény a *cv2.absdiff()*, amely az abszolút különbéget veszi a két kép között, így nem lesz a 0-255 értéktartományból kilépés.

$$\mathrm{dst}(I)=\mathrm{saturate}(|\mathrm{src1}(I)-\mathrm{src2}(I)|)$$

Ahol a saturate() a mátrixokat váltja át hasonló formába. Ez a lépés a mozgás detektálás alapja, hiszen ahol semmilyen mozgó objektum nincs, ott 0 vagy ahhoz közeli lesz a kivonás értéke, ahol viszont van eltérés a két kép között (fényváltozás), ott feltehetőleg mozgás történik.

Zajszűrés

A következő lépés a zajszűrés, az itt használt függvény a *cv2.GaussianBlur()*. Ez a függvény a Gauss szűrőt használja. Paraméternek pedig a kernel méretét lehet beállítani, ez a kernel méret jelen esetben 5x5 volt. A zajszűrést konvolúció művelettel hajtja végre. A konvolúció egy matematikai lineáris művelet. Egy példán tekintve: inputként kapunk egy képet, aminek minden egyes pixelét számokkal reprezentáljuk. Az egyszerűség kedvéért legyen a kép fekete alapon egy kézzel írott szám. Ahol rajzoltunk ott világosabb pixeleket találunk, ahol nem ott feketéket. Ezeket a részeket reprezentálva egy pixel 0-1 közötti értéket vehet fel, ahol a 0 jelenti a teljes fekete pixelt míg az 1 a teljesen fehért. Ezeket a pixeleken egy úgynevezett „kernel” járja végig, ami nem más, mint egy alacsony dimenziójú mátrix. A végig járás alatt, pedig a kernel minden egyes eleme összeszorzódik az adott kép megfelelő pixelének az értékével, és az így kapott szorzatok összege adja meg a kimeneti pixel értékét.

$$J(x,y) = \sum_{u=0}^{U}\sum_{v=0}^{V} K(u,v)I(x+u-\frac{U-1}{2}, y+v-\frac{V-1}{2})$$

Az egyik probléma a széleknél van, ahol véget ér a kép, és nem található pixel. A beépített OpenCV alapméretezett megoldást használom, ami a BORDER_REFLECT_100.Reprezentálása: gfedcb|abcdefgh|gfedcba

A képen a zajszűrés utáni kép látható.

Küszöbérték keresés

A küszöbérték keresés lényege, hogy egy adott értékhez viszonyítva mondjuk meg, hogy az adott pixelpont 0-ás vagy 1-es értéket vesz fel. Ezt az értéket kézzel is be lehet állítani, a saját megoldásban viszont a beépített cv2-es Otsu algoritmust használom. Ami egy hisztogramot állít elő a képből, majd minimalizálja az osztályon belüli varianciát, és annak az értékét adja meg küszöbértéknek.

$$\sigma_{2w}(t)=q_1(t)\sigma_{21}(t)+q_2(t)\sigma_{22}(t)$$

ahol

$$q_1(t)=\sum_{i=1}^{t}P(i)\&q_2(t)=\sum_{i=t+1}^{I}P(i)$$

$$\mu_1(t)=\sum_{i=1}^{t}iP(i)q_1(t)\&\mu_2(t)=\sum_{i=t+1}^{I}iP(i)q_2(t)$$

$$\sigma_{21}(t)=\sum_{i=1}^{t}[i-\mu_1(t)]^2P(i)q_1(t)\&\sigma_{22}(t)=\sum_{i=t+1}^{I}[i-\mu_2(t)]^2P(i)q_2(t)$$

Így az így kapott értéktől függően a kép minden egyes képpontja vagy teljesen fekete vagy fehér értéket vesz, attól függően, hogy meghaladja-e a megkapott küszöbértéket.

$$J(x,y) = \begin{cases} 0, & ha\ I(x,y) < kiszámolt\ küszöbérték \\ 255, & különben \end{cases}$$

A képen az Otsu algoritmus eredménye látható.

Körvonal keresés majd határoló dobozok rajzolása

A cv2.findContours() függvénnyel oldom meg az első lépést. A függvény egy bináris képen körül határolja a fehér pontokat és visszaadja a megtalált pontokat. Majd ezekre a pontokra a cv2.boundingRect() függvénnyel egy körül határoló téglalap pontjait megadom. Ha az adott téglalap területe (cv2.contourArea) túllép egy bizonyos határt, akkor mozgásnak veszem és egy körül határoló színes téglalapot rajzolok köré a cv2.rectangle() függvénnyel.

Elvárt működés közben körvonalak rajzolása saját teszt videón.

TESZTELÉS

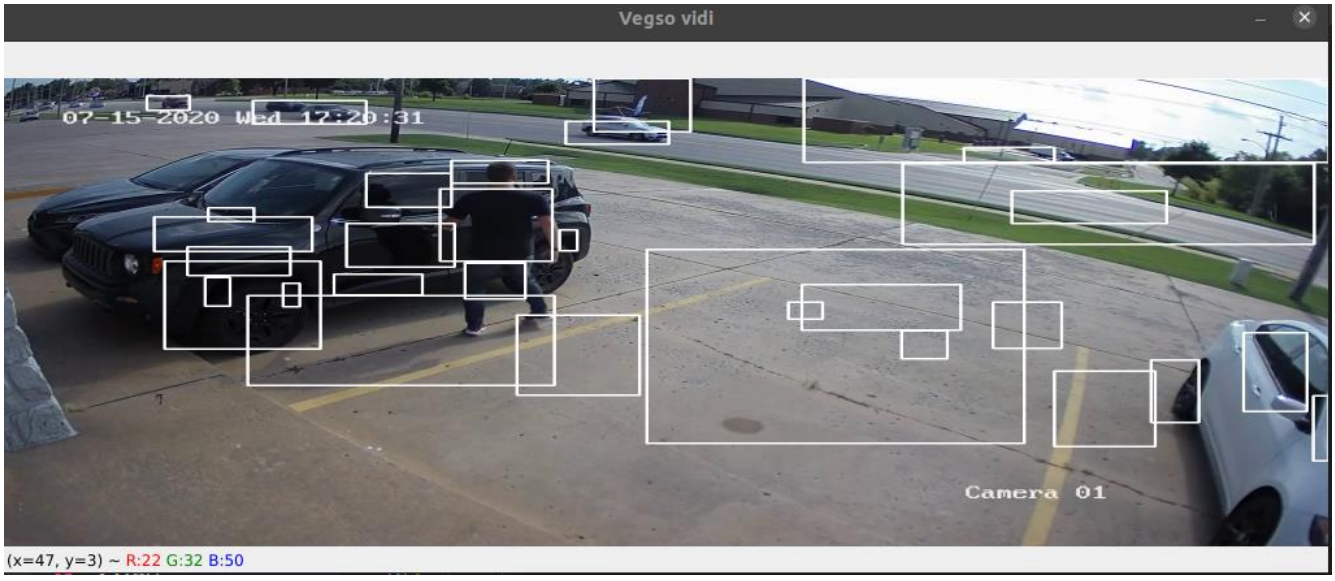
A teszteléshez használt videóanyagot saját kezűleg vettem fel nagyrészt, vannak benne interneten található anyagok is. Az algoritmus a háttér kinyerése miatt, egy fix pozíciójú kamera videóanyagát tudja optimálisan feldolgozni. A kamera apró mozgása, a háttér hirtelen változása emiatt az ok miatt befolyásolja az eredményt. A küszöbérték keresésénél pedig a kiválasztott algoritmusból adódik, ha nagy kontraszt van a kép részletei között, akkor nem optimális az eredmény.

Tesztesetek lebontása

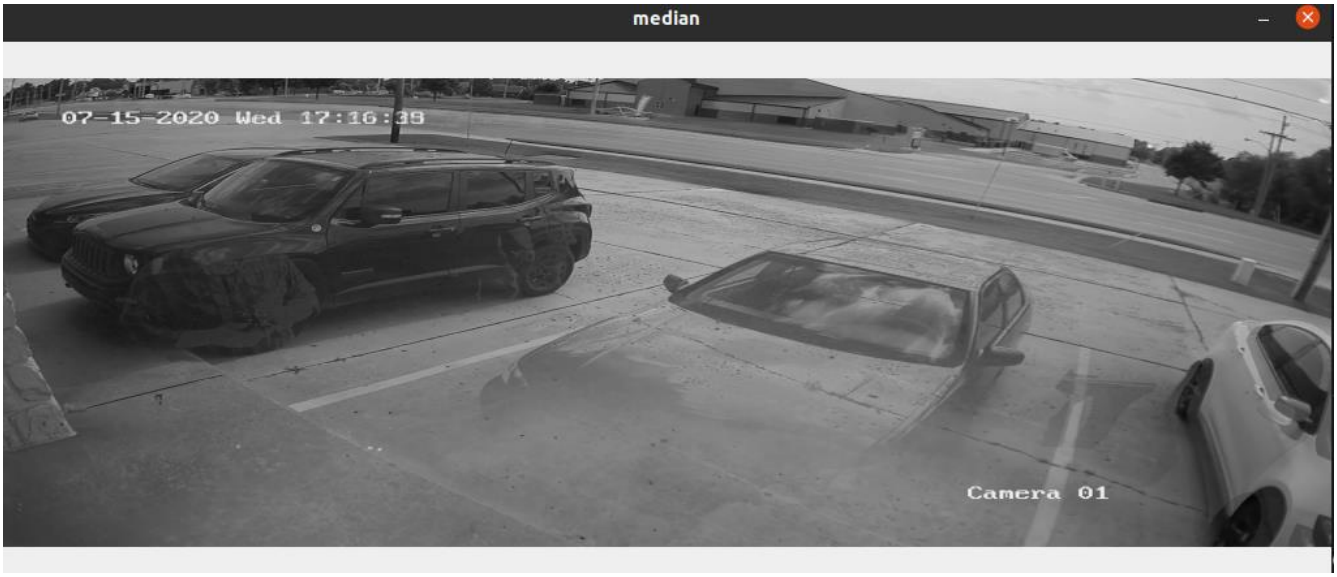
A megadott teszt videó címét és annak a lebontását fogom megadni ebben az alfejezetben.

„sc1” – A fix kamera miatt megfelelően működik az algoritmus, valamint nincsenek nagy kontrasztok a képben. Amint az ember kimozog a képből, az algoritmus a fehér, világos számokat és órát adja be mozgásnak, amik pluszban jelen vannak a képben.

„sc2” – A kamera itt is fix, viszont az előtér nagy változásai miatt, nem megfelelő a háttér kinyerése azon a részen, emiatt sok hibás észlelést végez. A háttérben látható autóúton, viszont elvárt módon működik az észlelés, ott sikeres volt a háttér kinyerése. A helyszín változása miatt téves észlelések vannak, emellett a nem megfelelő fényviszonyok is okoznak téves észlelést. A képen a hibás észlelést lehet látni az



előtérben, míg az úton sikeresen jelzi a mozgó járműveket.



A képen látni lehet, hogy a háttér kinyerésénél hibák voltak, ez befolyásolja a detektálást.

„sc5” – Az elvárt észlelések történnek, viszont a megfelelő fény miatt nehezen detektálja az ember mozgását. Itt az Otsu algoritmus választása miatt lesz ilyen eredmény.

„barik1” – Itt a kamera apró mozgása miatt, valamint a nagy kontrasztos háttér miatt van sok téves észlelés az elején. Amint a bárányok beérnek a képbe a mozgásukat megfelelően érzékeli. A fehér színük miatt egy nagy mozgásnak érzékeli az algoritmus a bárányokat, nincs felkészítve a sok hasonló objektum egy csapatban való mozgására, nem tudja megkülönböztetni a különálló objektumokat.

„kerek2” – A kamera mozgása itt is téves észleléseket eredményez. A kerék mozgását viszont sikeresen érzékeli, viszont mivel a kerék üres, a belsejét nem érzékeli. Nincs felkészítve az algoritmus az ilyen típusú mozgásokra, ahol a mozgáson belül „lyukak” találhatóak.

„kutya1” – Megfelelően érzékeli a mozgást, a kamera mozgása miatt észlel tévesen.

„levelek1” – A kamera mozgása miatt megint téves észlelés, amint a levelek a képbe érnek, helyesen érzékel, viszont ez nagyon kevés ideig történik.

„mento_kor1” – A kamera mozgása itt is bezavar, ha a képben van a mozgás akkor helyes az észlelés.

„si1” – Helyes a mozgás érzékelés. Több zavaró tényező miatt vannak téves észlelések. Nagy háttérkontraszt miatt tévesen érzékel. A kamera apró mozgása itt az egyenletesnek mondható fehér háttér miatt nem jön ki annyira, viszont a kamera felvételének a távolsága csökken, emiatt téves észlelések vannak. A fókusztávolság változására nincs felkészítve, hiszen a kinyert háttér emiatt nagy mértékben torzul.

„si2” – Az „si1” esetben leírtak érvényesek.

„si3” - Az „si1” esetben leírtak érvényesek.

„teli4” – Helyes érzékelés, a labda fehér foltjai hasonlóak a fehér háttérhez, emiatt nem egy teljes kerek objektumnak érzékeli az algoritmus a pattogó labdát. Ezt a hibaesetet a „kerek2”-ben leírt hibához tudnám hasonlítani, egy objektumot kisebb részekként érzékel az algoritmus.

Összegzés

A tesztesetekből látható, hogy a kamera mozgása és nem fix pozíció nagy hatással van az eredményességre. A háttér kinyerése miatt több előfeltételnek is teljesülni kell a megfelelő eredményhez. Valamint a nagy kontraszt miatt is hibás eredményt ad az algoritmus. A komplexebb objektumok is téves eredményt adnak, ahol van üres rész belül vagy éppen a háttérrel megegyező foltok vannak az objektumon. A sok hasonló objektum elkülönítésre sincs felkészítve az algoritmus. A megfelelő körülmények mellett, eredményesen érzékeli a mozgást az algoritmus. A háttér kinyerése miatt, több mozgást is fel tud ismerni egyidejűleg, viszont a különálló objektumokat nem érzékeli. A keresendő kontúr paraméter finomításával lehetne tovább optimalizálni az algoritmust. A mozgást sikeresen felismeri az algoritmus, és egy téglalap rajzolásával mutatja, hogy hol van mozgás a képen.

FELHASZNÁLÓI LEÍRÁS

A repository letöltése után lehetőség van a Linux operációs rendszer felhasználóinak a program kipróbálására. A program futtatásához a következő programok szükségesek:
Python 3.8.10 verzió
OpenCV 4.2.0 verzió
numpy csomag
A csomagokat a requirements.txt-be kimentettem és egy pip parancs segítségével egyszerre le lehet tölteni a követelményeket a txt segítségével.
python3 -m pip install -r requirements.txt

Az általam felvett teszt videók rendelkezésre állnak a test/ almappában, viszont van lehetőség saját készítésű, új videó tesztelésére is. A választott saját videót a mappakönyvtárba kell mozgatni, ahol megtalálható az arg_main.py nevű fájl. Ezután egy python3 arg_main.py -<saját_video_neve.kiterjesztése> paranccsal lehet tesztelni az új videót. Ha az egyik alap tesztesettel szeretné kipróbálni a program működését, akkor egy szövegszerkesztő segítségével meg kell nyitni a main.py programot. A program XXX.sorában látni lehet egy sort, ami XXX=”test1.MOV”. Itt a test1.MOV helyére, úgy, hogy az idézőjelek megmaradjanak kell beírni a választott teszt videó nevét. Ezután el kell menteni a szerkesztést, és egy python3 main.py paranccsal már az új beírt tesztesetet fogja az algoritmus feldolgozni.

IRODALOMJEGYZÉK

Hollósi János – Gépi látás előadások, Széchenyi István Egyetem, Győr, 2021

https://ms.sapientia.ro/~vajdat/education/imageprocessing/Labor_SegmentalasKuszoboles_Otsu/Otsu%20algoritmus%20I.pdf

https://www.techopedia.com/definition/32309/computer-vision

https://link.springer.com/chapter/10.1007/0-387-24579-0_5

https://docs.opencv.org/3.4/index.html