



---

# FORGALOMSZÁMLÁLÓ ALKALMAZÁS DOKUMENTÁCIÓJA

---



KÉSZÍTETTE: VIGH DÁNIEL

# Tartalom

Bevezetés, háttérismeret .....	2
Tervezés, és végül a megvalósítás .....	4
Tesztelés .....	11
Felhasználói leírás, algoritmus használhatósága: .....	16
Felhasznált irodalom: .....	17

## Bevezetés, háttérismeret

Az általam választott téma a forgalomszámláló alkalmazás elkészítése volt. Mivel először tanulom ezt a tárgyat, illetve magát a Python nyelvet így ezen a nyelven történt meg az elkészítése is, amely hosszú tanulási folyamat eredményeként jött létre. Elsősorban, amikor kiválasztottam a témát voltak elképzeléseim, és terveim, hogy kellene ezt az alkalmazást megcsinálni. Több órán keresztül jártam a téma után, és a témakörben, majd megtaláltam a „MOG” rövidítésű szót, amelynek a segítségével aztán képes voltam megalkotni a saját művemet. Persze szükséges volt az órai részvétel is, ahhoz, hogy le is tudjam programozni, illetve az elméleti tudás is, hogy teljes képem legyen róla. Az első, ami azonnal beugrott az, hogy egy vonal segítségével tudnám én ezt számlálni, de azon kívül sok fogalmam nem volt a képfeldolgozásról. A félév elején talán a 3-4. óra környékén egyre inkább állt össze az adott terv, az elképzelés, amelyeket használnom kell majd a projekthez. Ezeket mindig jegyzeteltem, és kiemeltem, hogy könnyebb legyen az itthoni áttekintés miatt. Igazából az elején nem voltam biztos benne, hogy Python nyelvvel kellene megcsinálnom, hiszen eddig még nem tanultam, csak C-t, de részletesebb utánakeresés után rájöttem, hogy ezzel lehet a legérdekesebb mindenképpen, főleg, hogy órán is ezekre csináltunk gyakorlati példákat.

Jelen esetben ugyebár egy mozgóképről beszélünk, ahol van egy állandó háttér, illetve vannak a mozgó járművek, melyeket detektálni kell. Sok problémát okozott a videó megtalálása is az internetről, hiszen traffic kamerákat nem igazán tesznek/tehetnek fel. Aztán végül találtam pár megfelelő videót, amely nem volt nagy felbontású, és nagy méretű, így még gyenge teljesítményű laptopomon is tesztelni tudtam a programot. Bár megmondom őszintén párszor így is kifagyott a gépezet. Ezután voltak további feladatok. Az elején számos elméleti dologgal szeretném kezdeni. Az első és a legfontosabb a MOG ismertetése, részletes bemutatása.

A MOG egy úgy nevezett background subtractor, tehát magyarul egy „háttér kivonás”. Szinte minden „látással” kapcsolatos alkalmazásban használatos, hiszen az OpenCV keretein belül elérhető. A háttér kivonás számos olyan esetben használható, amikor például egy statikus kamera veszi valahova belépő, vagy kilépő járművek, vagy emberek számát. Technikailag ki kell vonni a mozgó előteret a statikus háttérből. Az alábbi képen látható emberek az eredeti kameraképen.



Majd ezután, ha alkalmazzuk rajta a MOG-ot:



Jól látható, hogy a nem mozgó háttér feketével „jelölt” viszont a mozgóképet fehérrel jelölik. Jól kivehető így, hogy fekete, és fehér színeket használ csak a MOG. Viszont, mint sok mászt ezt is tökéletesítették, és létrejött MOG2 néven egy fejlettebb háttér-kivonással. A MOG2, és a MOG között annyi különbség van „csupán”, hogy míg a MOG nem foglalkozik a tárgyak, vagy emberek árnyékával a MOG2 igen. Az alábbi képen jól látható módon.



Ahogy látjuk ugyanaz a kép, viszont itt megjelenik szürke színnel az árnyékolás.

Még egy fajtája ismert a MOG-nak, viszont ez nem a MOG3 nevet kapta, hanem a GMG nevet, tehát a BackgroundSubtractorGMG. A GMG már olyan detektálásra képes, hogy nem csak az

árnyékolásra figyel, hanem az úgynevezett zajokat is kiszűri. Ez sokszor fontos, ha nem megfelelő az adott mozgókép minősége, és szükség van az élek jobb minőségű megtalálására. Az alábbi képen látható eredmény jött létre a fent említett GMG miatt.

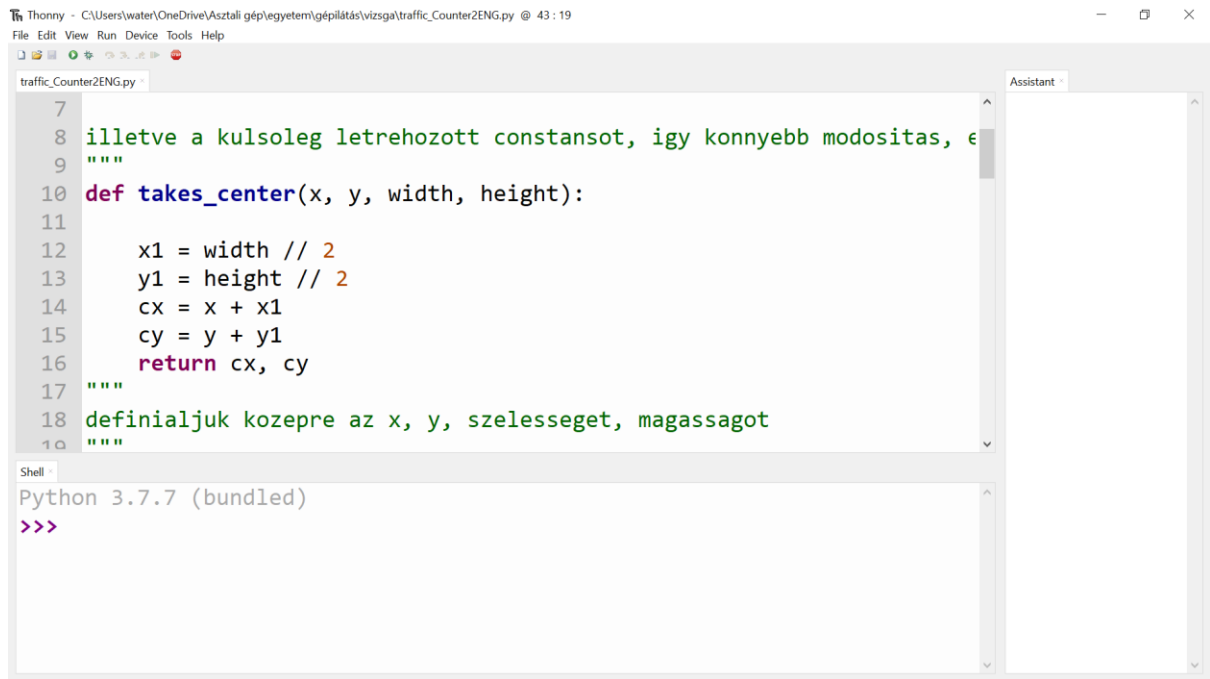


A képen nem látható az árnyékolás szürke színnel jelen esetben, hanem itt a kontúrok, szélek kerülnek előtérbe.

### **Tervezés, és végül a megvalósítás**

Tanulmányozva a MOG működését tudtam, hogy megtaláltam a számomra szükséges algoritmust. Viszont ahhoz, hogy az algoritmus megfelelően működjön, és elég gyors legyen szükség volt késleltetésre, erről majd később. Először is megvolt a terv miszerint, ha egy vonalat tudok a mozgóképre helyezni, illetve egy detektálást, majd egy számlálót, ami számlálja a vonalon áthaladó forgalmat, és minden áthaladó jármű után a számláló csak 1-et számoljon.

Először ehhez szükségem volt egy úgy nevezett pozícionálásra. Definiáltam a „takes-center”-t, amelyben létrehoztam egy  $x$ , és egy  $y$ -t, illetve a szélességet, és a magasságot egyaránt. Majd a változásokat, beleraktam egy  $c(\text{enter})x$ , és egy  $c(\text{enter})y$ -ba. Az  $x1$ , és  $y1$  a szélességet, és a magasságot jelentette. Az alábbi módon:



```
7
8 illetve a kulsoleg létrehozott constansot, így könnyebb modositás, e
9 """
10 def takes_center(x, y, width, height):
11
12     x1 = width // 2
13     y1 = height // 2
14     cx = x + x1
15     cy = y + y1
16     return cx, cy
17 """
18 definialjuk kozepre az x, y, szelesseget, magassagot
19 """
```

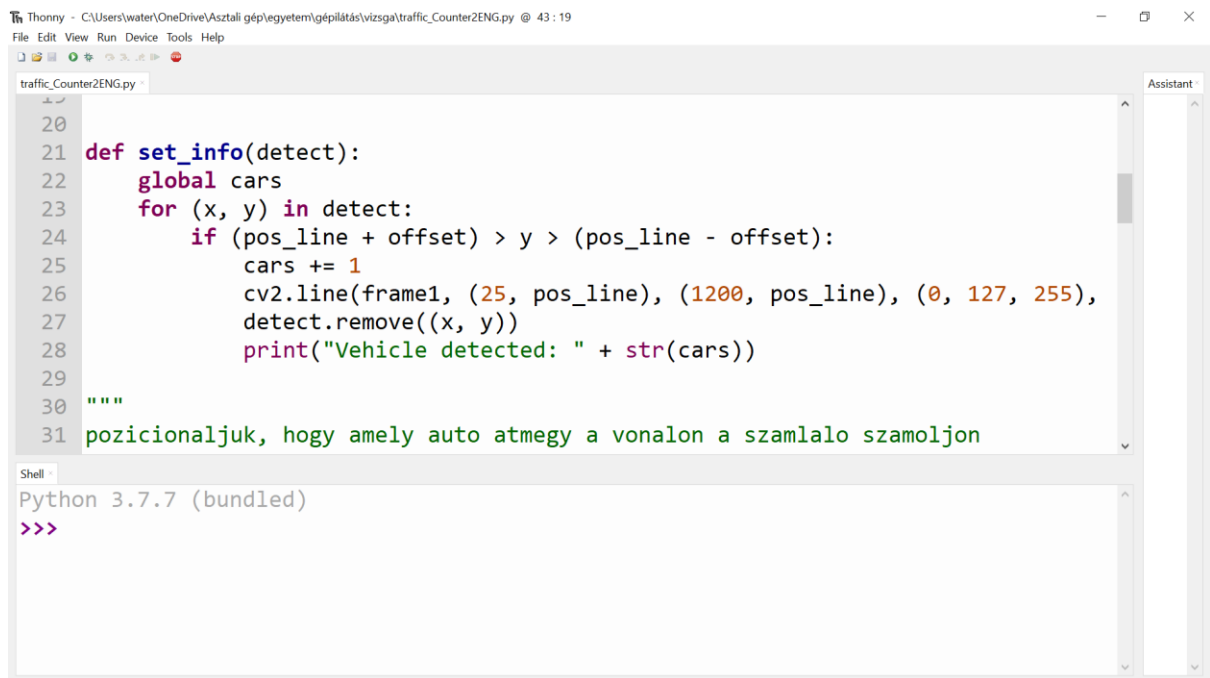
Shell

Python 3.7.7 (bundled)

>>>

Majd szükségem volt a minimum magasságra, és a minimum szélességre is, illetve a megengedett képpont mennyiségére. Ezeknek létrehoztam egy külön konstans-ot, amelyet beimportáltam. Ez a megoldás azért jobb, ha esetlegesen teszteléskor kijön valami hiba elég ha átirom ott, és nem kell nagyon bogarászni. Persze bele is lehetett volna írni a „mainbe”, de ez így egy jobb megoldás, illetve az importálás teljesen megszokott folyamat, tehát nem fog bezavarni. A vonal színét és pozicionálását megterveztem. Mindenképpen úgy kellett kiválasztanom az adott színét a vonalnak, hogy az ne zavarjon be, és jól prezentálható legyen, így egy kék színt választottam. Nyilván lehetett volna ez bármely más szín is, de ez elég jól

kitűnt, így esett a világosabb élénk kékre a választásom.



The screenshot shows the Thonny Python IDE interface. The main editor window displays a Python script named `traffic_counter2ENG.py`. The script defines a function `set_info` that takes a `detect` object as input. Inside the function, a global variable `cars` is declared. A `for` loop iterates over the `detect` object, and an `if` statement checks if the `y` coordinate is within a specific range (`pos_line + offset > y > (pos_line - offset)`). If true, the `cars` counter is incremented by 1, a line is drawn on the video frame, and the detected object is removed from the `detect` list. A `print` statement outputs the current count. A docstring at the bottom explains the function's purpose in Hungarian. Below the editor is a shell window showing the Python 3.7.7 (bundled) prompt.

```
20
21 def set_info(detect):
22     global cars
23     for (x, y) in detect:
24         if (pos_line + offset) > y > (pos_line - offset):
25             cars += 1
26             cv2.line(frame1, (25, pos_line), (1200, pos_line), (0, 127, 255),
27                 detect.remove((x, y))
28             print("Vehicle detected: " + str(cars))
29
30 """
31 pozicionaljuk, hogy amely auto atmegy a vonalon a szamlalo szamoljon
```

Shell  
Python 3.7.7 (bundled)  
>>>

A vonal hosszúságát úgy adtam meg, hogy a 25, és az 1200 között szerepeljen. Ez volt az a hosszúság, amely pont átért a videóban lévő utat. Gondolkodtam rajta, hogy miért nem helyeztem rá az egész képernyőre, vagyis igazából úgy is kezdtem, csak a videóban nem csak az autót szerepelt, hanem egy háttérben lévő benzinkút is, ahol közlekednek az autók, és ha túl hosszú a vonal, akkor azt is belevettük volna a számolásba, viszont a cél nem ez volt, hanem az adott útszakaszon lévő forgalomra korlátozva számolni. Abban az esetben, ha másik videón tesztelem lehet, hogy szükségem lesz a változtatásra, de ez manuálisan megtehető.

```

while True:
    ret, frame1 = cap.read() # Készítsuk el a video kepcockait
    tempo = float(1 / delay) #ezert kellett a time, hogy alkalmazhassunk delayt
    sleep(tempo) # Késleltetjük a feldolgozást
    grey = cv2.cvtColor(frame1, cv2.COLOR_BGR2GRAY) #fekete-feherre alakítsuk a keretet
    blur = cv2.GaussianBlur(grey, (3, 3), 5) #A kep hibait el kell tavolitani, mivel a video csak mp4 tipusu

    img_sub = subtraction.apply(blur) #homalyt kivonjuk a keprel

    dilat = cv2.dilate(img_sub, np.ones((5, 5))) #suritjuk ami maradt a kivonasbol
    kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (
        5, 5)) #Letrehozunk egy 5x5 matrixot, es a matrixot 0,1 kozotti resnel ellipszist keszitunk

    dilated = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel) #kepjavitas, itt igazabol a pixelhibakat kuszoljuk a morph segitsegevel, automata kitoltes

    dilated = cv2.morphologyEx(dilated, cv2.MORPH_CLOSE, kernel)

    contour, img = cv2.findContours(dilated, cv2.RETR_TREE, cv2.CHAIN_APPROX_SIMPLE)
    cv2.line(frame1, (25, pos_line), (1200, pos_line), (255, 127, 0), 3)
    for (i, c) in enumerate(contour):
        (x, y, w, h) = cv2.boundingRect(c)
        validate_outline = (w >= width_min) and (h >= height_min)
        if not validate_outline:
            continue

```

Ezután jött a While True rész, avagy a fő része az algoritmusnak. Ebben a részben szükség volt ugye a MOG-ra. Itt először is el kellett ugye indítani a videót magát, ehhez szükség volt egy „ret, frame1 = cap.read()” parancsra. A frame1-et előzőleg már deklaráltuk a cap.read pedig az olvasást indítja el. Utána alkalmaztam egy késleltetést (delay). Erre azért volt szükség, ha esetlegesen begyorsul a forgalom, akkor is tudjuk követni. Utána némi átalakítást eszközöltem, élesítést, és egyéb dolgot, amitől a számlálás jobban fog teljesíteni. Volt homályos részek kivonása, és használtam a MorphologyEX parancsot, mely segítségével a pixelhibákat szűrtem ki.

Ezután szeretném részletesebben is bemutatni a **cv.dilate()**, **cv.morphologyEx()** parancsokat, hogyan is működnek, és miért volt ezekre szükség. A morfológiai transzformációk néhány egyszerű művelet, amelyek a kép alakján alapulnak. Általában bináris képeken hajtják végre. Két bemenetre van szüksége, az egyik az eredeti képünk, a második strukturáló elemnek vagy kernelnek nevezzük, amely eldönti a működés jellegét. Két alapvető morfológiai operátor az erózió és a tágulás. Ezután a változatai, például a nyitás, zárás, színátmenet stb. Látni fogjuk őket egyenként a következő kép segítségével:





Első fajtája a morfológiai transzformációknak az az erózió. Mint a szó maga jelenti, hogy „lepusztítás” itt is úgy mondván egy pusztítás, itt az előtér objektumának határait rombolja le (mindig próbálja fehérben tartani az előteret). Szóval a kernel átcsúszik a képen (mint a 2D konvolúcióban). Az eredeti képen lévő pixel (1 vagy 0) csak akkor tekinthető 1-nek, ha a kernel alatt található összes képpont 1, különben erodálódik (nullára állítja). Tehát a határ közelében lévő összes pixelt eldobjuk, a kernel méretétől függően. Szóval az előtér objektumának vastagsága vagy mérete csökken, vagy egyszerűen a fehér régió csökken a képen. Hasznos a kis fehér zajok eltávolítására. Itt van rá egy példa hogyan működik egy 5x5-ös kernelen.

```
import cv2 as cv
import numpy as np
img = cv.imread('j.png',0)
kernel = np.ones((5,5),np.uint8)
erosion =
cv.erode(img,kernel,iterations=1)
```

Ennek eredményeképpen:



A második fajtája az a „dilation” azaz érzékelés. Ezt használtam én. Erről azt kell tudni, hogy ez pont ellentette az erózióval. Itt egy pixel elem „1”, ha legalább egy pixel a kernel alatt „1”. Vagyis növeli a fehér részeket a képen. Mivel eltűnnek a zajok, emiatt a mozgó tárgyak „fehér foltja” növekedik. Hasznos egyébként még akkor, ha egy törött tárgy részeit szeretnénk összekapcsolni.

Én ezt így használtam:

```
dilat = cv2.dilate(img_sub, np.ones((5, 5)))
```

A harmadik fajta, amit én is használtam, igazából majdnem ugyanaz mint az erózió.

Így használható:

```
dilated = cv2.morphologyEx(dilat, cv2.MORPH_CLOSE, kernel)
```

Végeredmény:



Ezek a fontosabbak, amelyek gyakorta használhatunk ilyesfajta projektekhez. Egyébként ezeken kívül ismert még 4 fajta, de azokat csak speciális esetekben használjuk.

Ezek a closing (zárás), Morphological Gradient (Morfológiai színátmenet), Top Hat (cilinder), Black Hat (fekete cilinder/sapka).

A zárás (closing) fordított nyitás, tágulás, majd erózió következménye. Hasznos az előtérben lévő objektumok belsejében lévő kis lyukak vagy az objektumon található fekete fekete pontok bezárásakor.

Eredményeként ez a kép jön létre:



A morfológiai színátmenet bemutatja a különbséget a kép kitágulása és eróziója között.



A Top hat bemutatja a különbséget a bemeneti kép és a nyitás (opening) kép között. Az alábbi példa egy 9x9-es kernelre vonatkozik.



A Black hat pedig majdnem ugyanaz mint a top hat, csak itt nem a nyitási kép, hanem a zárási kép és a bemeneti kép közötti különbséget mutatja be.



A következő részben a strukturáló elemek ismertetéséről lesz szó, ezek közül használtam az egyik típust melynek neve elliptikus kernel, azaz MORPH.ELLIPSE. Az alábbi módon épül fel:

```
# Elliptical Kernel
```

```
cv.getStructuringElement(cv.MORPH_ELLIPSE,(5,5))
```

```
array([[0, 0, 1, 0, 0],
```

```
[1, 1, 1, 1, 1],
```

```
[1, 1, 1, 1, 1],
```

```
[1, 1, 1, 1, 1],
```

```
[0, 0, 1, 0, 0]], dtype=uint8)
```

Ebben az esetben használhattam volna szerintem téglalap alakút is, de ez volt igazából szimpatikus, és ezzel működött megfelelően a program.

## Tesztelés

Először csak egy videóval kezdtem el tesztelni az adott algoritmust, és igazából erre is készül el az adott modell. A video.mp4 fájl volt az említett fájl. Majd az internetről sikerült szereznem pár másik videófájl, vagyis amik igazából traffic camera kulcsszó alatt fellelhetőek voltak.

Amikor befejeztem az algoritmust már egyből sejtettem, hogy miben lesznek gondok, és milyen helyzetekben lesz megfelelő az általam elkészített program.

Igazából minden videóhoz el kell készítenem a hozzá passzoló algoritmust, ez igazából csak abból állt, hogy a vonalat máshová kellett pozícionálni, illetve a vonal méretét az adott részhez. Teszteltem 6 videón, igyekeztem mindenféle esetre elkészíteni. A késleltetés mértékét ahhoz állítottam, hogy a videóban mennyire gyors a forgalom, és mennyire sok a mozgókép része, amit detektálni kell. Ez többé kevésbé sikeres volt.

Ugye mint említettem szükséges volt importálnom a time-ot, ami igazából tényleg ehhez a részhez volt szükséges igazán. Az elején amikor time, delay nélkül próbáltam tesztelni az algoritmust, akkor az olyan jól sikerült, hogy a program (Thonny) teljesen kifagyott. Erre kutakodtam, kerestem a megoldást, majd végül sikeresen megtaláltam.

A Python sleep () hívást a program késésének szimulálására. Ez akkor kell, ha meg kell várni egy fájl feltöltését vagy letöltését, vagy egy grafika betöltését vagy a képernyőn való megjelenését. Előfordulhat, hogy szünetet kell tartania a webes API-hoz intézett hívások vagy az adatbázisba irányuló lekérdezések között. A Python beépített támogatást nyújt a program elalvására. Az idő modulnak van egy sleep () funkciója, amellyel felfüggesztheti a hívott dolog végrehajtását akárhány másodpercre is.

Ezután jönnek a videók, amelyeken tesztelve lett a program. Először is ugye az alap

**video.mp4:**



A képen jól látható, hogy átlagos forgalom van. A háttérben a szél fújja a fát, illetve a benzinkúton folyamatos forgalom van.

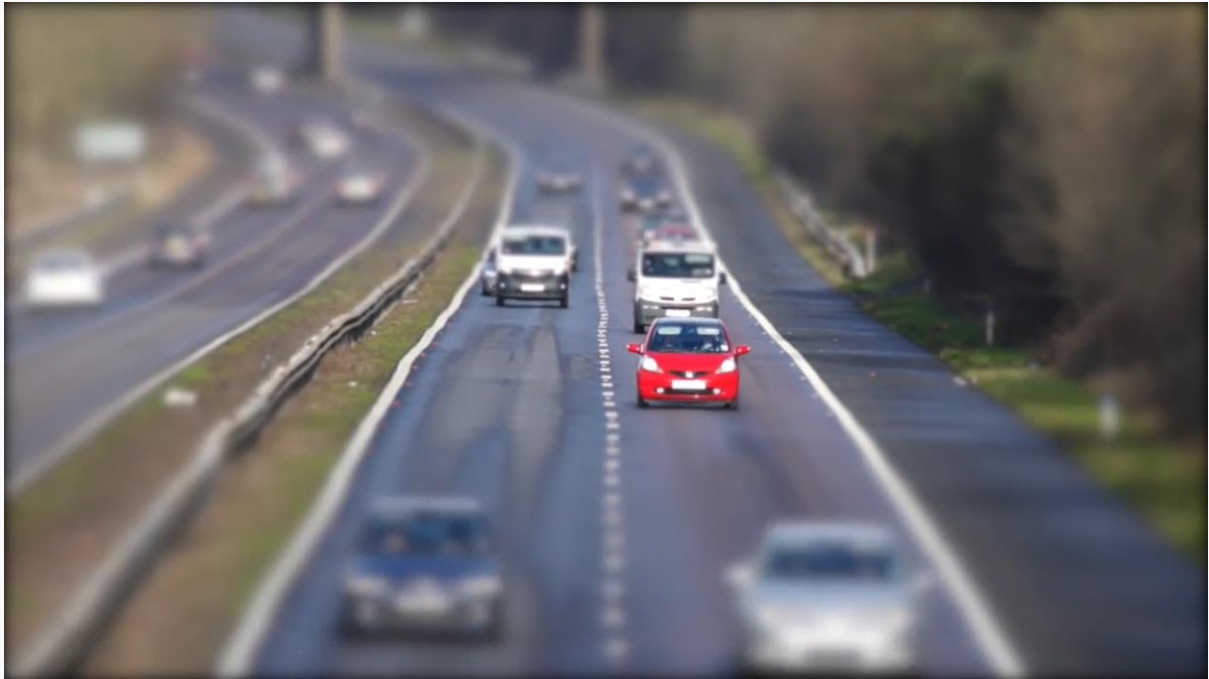
**video2.mp4:**



Kevés a forgalom, és a videót magát ez jellemzi. Természetesen a háttérben lévő fák mozgása itt is zavaró tényező. Persze a vonal hosszát, és pozícióját át kellett állítani az előző videóhoz képest.



**video3.mp4:**



Ezen a képen alapján látható egy homályosítás, illetve rettentő gyors, sűrű autópályás forgalom. Az algoritmusnak nem volt megfelelő az, hogy homályosítva van a háttér, viszont ha az éles részre tettem a vonalat többé kevésbé jól számolt addig a pontig ameddig már egymást nem érték az autók.

**video4.mp4:**



Hát sikeresen találtam egy olyan videót, amely szerintem a megírt algoritmusom egyik rákfenéje. Abban az esetben, ha nincsen pontosan beállítva az algoritmus, akkor ezen a videón az egész részt mozgónak érzékelte az algoritmus, és használhatatlan volt.

Amennyiben a megfelelő finomhangolás elvégzésre került a számláló relatíve jó eredményt adott.

#### **video5.mp4:**

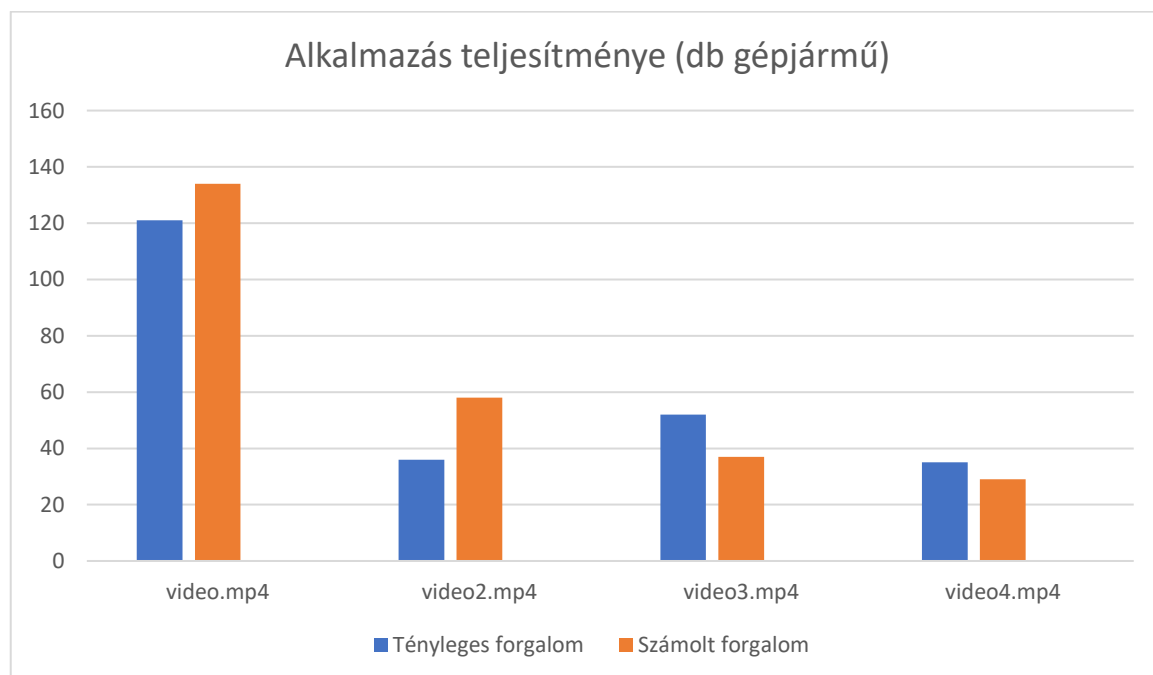


A forgalom nagyon gyors, valószínűleg német autópályás felvétel volt, ráadásul a kamera felfüggesztése sem volt megfelelő, és a szél hatására mozgott.

#### A tesztelés számszerűsítve:

Az első videón (video.mp4) úgy alakult, hogy 134-et számolt a számláló, viszont csak 121 gépjármű haladt át, az algoritmus 90%-os teljesítménnyel dolgozott, és csak akkor vétett hibát, ha egyszerre több kamion haladt át, nagyobb volt a forgalom. Ez egy egészen jó eredménynek mondható, de nem tökéletes. A következő a video2.mp4 teszt, amelynél áthaladt ténylegesen 36 autó, a számláló pedig 58-at számolt. Ez 62%-os teljesítményt jelent. A video3.mp4 teszt eredményei a következők lettek: az áthaladó forgalom száma 52 gépjármű volt, a számláló 37-et számolt. Ez annak tudható be, hogy a videó háttere homályosítva van. Ez egyébként egy 71%-os pontosságot jelent. A video4.mp4 fájl eredményei beformázás után egészen jó eredményt produkált. Ehhez is, illetve a többihez is szükséges egy belső formázás, viszont itt a szakasz hosszán is állítani kellett a többi esetben elég volt az elhelyezésén (ezt a constans.py fájlban kell megtenni). Amennyiben a formázás sikeres a program 29 autót számol, viszont 35 autó megy át. Itt is kevesebb a

számolt mennyiség mint a tényleges, viszont 83%-os pontosságot jelent. A következő video5.mp4 eredményei tökéletesnek tűnnek mégsem azok, sőt teljesen rossz, ha figyelmesen megnézzük. Ugyebár a tényleges forgalom 11 volt, az alkalmazás 12-t számolt, amely nagyon jó eredmény (92%-os pontosság). Viszont itt az alap kameramozgás miatt nem született jó eredmény ténylegesen, hiszen ha hosszabb let volna a video teljesen pontatlanul számolt volna az alkalmazás. Ha megnézzük a ToDetect ablakot láthatjuk a problémát. A video6.mp4-en is megtörtént a teszt, itt a helyzet ugyanolyan, mint a video5.mp4-en, hiszen a kamera rögzítése nem megfelelő, ezért folyamatosan mozog a szél miatt. Hiába van kis forgalom mégsem pontos a számláló, a tényszerű adatok miatt igen, viszont a működése nem megfelelő. A teszt itt 29 autót számolt, és 39 haladt át ténylegesen. Százalékosan nem rossz eredmény, de a működési probléma miatt nem megfelelő. Ebből a 6 videóból elegendő mennyiségű információ áll rendelkezésünkre, hogy számszerűsítés után kijelenthető, hogy az algoritmus kis, és közepes forgalmat minimum 80%-os pontossággal számol, abban az esetben, ha a constansban a finom hangolás megtörtént. A nagyobb forgalomnál, illetve a közelebbi kamerafelvételnél teljesít gyengébben, itt a 62%-os teljesítmény volt a legalacsonyabb, de ebben az esetben egymást érték az autók, és a kamionok, ráadásul egy homályosítás is volt a kamerafelvételen. Azoknak a videóknak az eredményeit ábrázolhatjuk diagrammal, amelyeknél az algoritmus jól működik. A video5.mp4 és a video6.mp4 eredményei nem megfelelőek, hiába adtak viszonylag jó pontosságot nem a valós információt kapjuk.





## Felhasználói leírás, algoritmus használhatósága:

Ahhoz, hogy az általam elkészített forgalomszámláló alkalmazás működhessen szükségünk van egy fordítóra, amelyhez én a Thonny-t használtam, de természetesen minden más egyéb fordító is tökéletes erre a célra, amely tud Python-t. Jelen esetben a Thonny 3.2.7. verzióját használtam, amely Python 3.7.7. verzióval történt a fordítás. A Thonny ingyenesen letölthető, és használható. Az alábbi linken bármikor elérhető (<https://thonny.org>), viszont abban az esetben, ha beírjuk a Google keresőbe, hogy „Thonny Download” az első találatok között ki fogja adni. Amit tudni kell a Thonny-ról, hogy ingyenesen letölthető, és rendkívül letisztult felülettel rendelkezik. Abban az esetben, ha feltelepítettük a Thonny-t nyissuk meg, és a következőképpen telepítünk olyan packageket, azaz külső csomagokat, melyekre szükségünk van. Ez jelen esetben az OpenCV, és a Numpy lesz. Ezt a legkönnyebben úgy tehetjük meg, hogy felül a „Tools” menüre kattintunk, majd ott kiválasszuk az „Open system shell...”-t. Ebben az esetben kidob egy parancssort, ahol utasításokat adhatunk. Ide a következő kifejezéssel telepíthetjük az OpenCV csomagot, amely tartalmazza a Numpy-t, illetve a még számunkra szükséges Time-t is. **A parancs:** „pip install opencv-contrib-python”. Ezután, ha befejeződött a telepítés, fel fogja ajánlani, hogy esetlegesen frissíteni szeretnénk-e, mivel lehetséges, hogy lesznek újabb verziók. Számunkra erre most nincsen szükség.

Ha ezekkel a formalitási telepítésekkel készen vagyunk már csak meg kell nyitni a kész algoritmust, amely az alábbi linkről letölthető [https://github.com/vighdani20/gepi\\_latas/blob/master/Traffic\\_Counter.py](https://github.com/vighdani20/gepi_latas/blob/master/Traffic_Counter.py). Amint a letöltés befejeződött, és megnyitottuk márcsak egy futtatásra van szükségünk, és az algoritmus lefut, viszont hibát fog adni, mivel nincsen letölve mellé sem a szükséges constants.py, sem pedig a tesztvideó. A constants.py az alábbi linkről tölthető le: [https://github.com/vighdani20/gepi\\_latas/blob/master/constant.py](https://github.com/vighdani20/gepi_latas/blob/master/constant.py). A tesztvideó pedig innen: [https://github.com/vighdani20/gepi\\_latas](https://github.com/vighdani20/gepi_latas). Jelen esetben innen a tesztvideók közül a video.mp4 fájlra lesz szükségünk. Amennyiben a hozzávaló komponensek egy mappában vannak a futtatás gombra kattintás után (zöld körön belüli fehér háromszög a Thonny-n belül, másik sor, balról a 4. ikon) a program lefordul, és az alábbi videón áthaladó forgalmat számláljuk. 2 felugró ablakunk közül az egyik a ToDetect, a másik pedig a Motorway Camera. Először nézzük meg a Motorway Camerát. A számláló felül, középen található a videón (Vehicle:). Ezen kívül találunk még egy vonalat (egyenest), amely kék színű, illetve hogyha az adott gépjármű áthalad rajta narancssárgára vált, és a számláló növekedik egyet. A másik ablak a

ToDetect nevet kapta (magyarul detektálni), itt láthatjuk az algoritmust működés közben, és hogy a MOG hogyan is dolgozik (az elméleti részben említett módon).

### **Felhasznált irodalom:**

[https://docs.opencv.org/master/d9/d61/tutorial\\_py\\_morphological\\_ops.html](https://docs.opencv.org/master/d9/d61/tutorial_py_morphological_ops.html)

<https://stackoverflow.com/questions/50987451/get-traffic-data>

<https://stackoverflow.com/questions/14494101/using-other-keys-for-the-waitkey-function-of-opencv/20577067>

[https://opencv-python-tutroals.readthedocs.io/en/latest/py\\_tutorials/py\\_imgproc/py\\_morphological\\_ops/py\\_morphological\\_ops.html](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_morphological_ops/py_morphological_ops.html)

[https://docs.opencv.org/3.4/d1/d5c/classcv\\_1\\_1bgsegm\\_1\\_1BackgroundSubtractorGMG.html](https://docs.opencv.org/3.4/d1/d5c/classcv_1_1bgsegm_1_1BackgroundSubtractorGMG.html)

<https://realpython.com/python-sleep/>