Gépi Látás

TKNB_INTM038

Vonalkód, QR-kód olvasása

Halász Dávid

MXVQT4

Tatabánya, 2022

Tartalomjegyzék

Tartalom

Bevezetés

A digitális forradalommal Beköszönő gyorsütemű technológiai modernizáció az addig hagyományos üzleti modellek újragondolását követelte. Az információs és kommunikációs technológia új, hatékonyabb azonosítási eljárások életre hívására Sarkalta a gazdaság szereplőit. Az internet elterjedésével és a nyomában kibontakozó új kereskedelmi gyakorlatokból fakadóan a termékek, illetve munkafolyamatok jobb transzparenciáját támogató fejlesztések, mint a vonalkód használata általánossá és nélkülözhetetlenné vált.[1]

Vonalkódok segítséget nyújthatnak a raktárkészlet nyilvántartásában, csomagok nyomon követésében, áruk eladásának gyorsításában, személyazonosításban, sőt felsorolhatatlanul sok területen. Előnyük abban rejlik, hogy elég csak egy eszközzel kezelni a szimbólumot és máris azonosíthatjuk amit szeretnénk, ahelyett, hogy ez rengeteg papírmunkával járna, és ugye az ember hajlamos hibázni, ha sok a papírmunkája. A gépek viszont leveszik a vállunkról ezt a terhet.[1]

A vonalkód legnagyobb előnye, hogy megfelelteti a termékeket a könnyű és pontos azonosíthatósági elvárásoknak, így jelentősen csökkenti a vállalkozások adminisztrációs terheit. Szerepe mind az élelmiszer, mind más ágazatokban meghatározó. Amennyiben egy termék azonosítószámmal és vonalkóddal érkezik az országba, úgy a termék életútja egészen a gyökerekig visszakövethető. A szállítmányozásban (küldemény azonosítás), illetve légiközlekedésben (feladott poggyász utókövetés) is előszeretettel alkalmazzák a munkafolyamatok nyomon követésére.[1]

Elméleti háttere

A program megvalósítás szempontjából két részre osztható fel, az elméleti részben ezeket fogom levezetni.

2.1 Detektálás

Mivel tudjuk, hogy a vonalkód körül, mögött mindig fehér szín található így azokat a területeket keressük meg az adott képen, ahol különböző műveletek után nagyobb egybefüggő fehér terület található.

Első lépésként csökkentjük a kép méretét majd létrehozunk egy szürkeárnyalatos képet. Ezután egy küszöbölést hajtunk végre a képen, hogy egy bináris képet kapjunk vissza, majd létrehozzuk a kép inverz változatát. A kép inverz változatán egy morfológiai szűrést, dilatációt hajtunk végre, így megnöveljük a kép fehér tartományát és csökkentjük a számunkra zajos képpontok számát, majd egy kontúr keresést is csinálunk rajta.

Az így kapott kontúr vonalakra meghatározzuk mekkora a legkisebb téglalap, amikbe azok belefférnek, és ezen területeket az eredeti képről kivágjuk. A kivágott részekről eldöntjük, hogy azok szélesebbek-e mint 95 pixel, mert az általam használt megoldás az annál kisebb felbontású vonalkódokat nem tudja dekódolni.

2.2 Dekódolás

A dekódolásnál a detektáló függvény által kivágott részeket egyesével fogjuk vizsgálni. Első lépésnek szürkeárnyalatos, majd binárissá alakítjuk a képünket, ezután széltében 1 pixel vastagságban végig megyünk a képen ,és meghatározzuk az egymást követő fehér és fekete pixelek hosszát.

Ezt a számsort 5 részre bontjuk: kezdő védővonalak, bal oldali minta, középső védővonalak, jobb oldali minta, és a végén lévő védővonalakra. A dekódolást a baloldali mintával kezdjük kiszámoljuk az arányokat, és az előre megadott számpárosokat végig ellenőrizve megállapítjuk, hogy hányas szám, és hányas paritás tartozik az adott vonalminta részhez. Ha végigértünk a mintán, akkor a paritások összegéből megállapítjuk az első szám összegét.

A végén még modulo 10 kalkulációval megállapítjuk a checkszumot és ezzel a számmal ellenőrizhetjük, hogy helyesek voltak-e a számításaink [2]

Kivitelezés

Előfeldolgozás

3.1 Detektálás

3.1.1 detect függvény

1. kódrészlet

```
import cv2
import numpy as np


def detect(img):
    scale_percent = 640/img.shape[1]
    width = int(img.shape[1] * scale_percent)
    height = int(img.shape[0] * scale_percent)
    dim = (width, height)
    resized = cv2.resize(img, dim, interpolation = cv2.INTER_AREA)

    gray = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY)
    ret, thresh =cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY_INV +
cv2.THRESH_OTSU)
    kernel = np.ones((3, 20), np.uint8)
    thresh = cv2.dilate(thresh, kernel)

    original_sized = cv2.resize(thresh, (img.shape[1],img.shape[0]), interpolation
= cv2.INTER_LINEAR)
    contours, hierarchy =
cv2.findContours(original_sized,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)
```

Az **1. kódrészletben** láthatóak legfelül a projekthez használt könyvtárat. A cv2 library tartalmazza az OpenCV nevű nyílt forráskódú könyvtára, amely alapvető funkciókat tartalmaz a gépi látás és a képfeldolgozás témaköréhez kapcsolódóan. A numpy pedig egy nyílt forráskódú kiegészítő csomag a Python programozási nyelvhez, mely a nagy, többdimenziós tömbök és mátrixok használatát támogatja egy nagy magas szintű matematikai függvénykönyvtárral.

A **detect** nevű függvény paraméternek egy képet vár. Meghívása esetén első lépésként arányosan csökkenti a kép méretét a **cv2.resize** metódus használatával. Ilyenkor a legcélszerűbb a **`cv2.INTER_AREA`** paramétert adni, ilyenkor terület alapú számítást hajt végre, míg nagyításnál a **`cv2.INTER_LINEAR`**-t jobb használni.(gyorsabb)[3]

A kép átméretezésére az utána következő műveletek kevésbé gépigényes lefutása miatt van szükség. A méretezés után a **cv2.cvtcolor** metódussal szürke árnyalatúvá alakítjuk a képünket, majd a **cv2.threshold** metódussal egy küszöbölést hajtunk végre a képen. Ez annyit jelent, hogy minden pixelt megvizsgálva az algoritmus a küszöbértékektől függően, hogy az alatt vagy felett van a pixel értéke nullára vagy kettőszázötvenöt-re állítja őket. A **`cv2.THRESH_BINARY_INV`** paraméter megadása esetén a kép inverz változatát kapjuk, a **`cv2.THRESH_OTSU`** paraméter átadása esetén pedig számol nekünk az általunk megadott értékeknél optimálisabb küszöbértéket. Ezután létrehozunk egy 3x20 kernelt és a **cv2.dilate** metódus használatával megnöveljük a fehér képpontok számát a képen.

A **cv2.resize** metódussal visszaállítjuk a kontúr keresésre felkészített képünket az eredeti méretére, majd a **cv2.findContours** metódust meghívjuk. A **`RETR_LIST`** paraméter minden kontúr detektálását adja meg, a **`cv2.CHAIN_APPROX_SIMPLE`** miatt pedig a kontúrok vízszintes, függőleges és átlós végpontjait fogja csak megadni, így kisebb lesz a memória használat.

2. kódrészlet

```
    candidates = []
    added_index = []
    for cnt in contours:
        rect = cv2.minAreaRect(cnt)
        box = cv2.boxPoints(rect)
        box = np.int0(box)

        cropped = crop_rect(rect,box,img)
        width = cropped.shape[1]

        if width>95:
                candidate = {"cropped": cropped, "rect": rect}
                candidates.append(candidate)
        index = index + 1
    return candidates
```

A **2.kódrészletben** egy **for** ciklussal végigmegyünk a **cv2.findContours** metódus által detektált éleken, a **cv2.minAreaRect** függvénnyel megállapítjuk a legkisebb területű, forgatott téglalapot, amiben a kontúr vonalunk elfér. Utána **cv2.boxPoints** függvénnyel megállapítjuk a téglalap négy csúcsát és a **np.int0** függvénnyel egész típussá alakítjuk az értékeket.

A **crop_rect** függvényt lejjebb, a **3.kódrészletben** fogom pontosabban ismertetni. A kivágott részről megállapítjuk, hogy szélesebb-e mint 95 pixel, ha igen akkor hozzáadjuk a **candidates** tömbhöz, és ezt adja vissza eredményül a **detect** függvényünk.

3.1.2 crop_rect függvény

3. kódrészlet

```
def crop_rect(rect, box, img):
    W = rect[1][0]
    H = rect[1][1]
    Xs = [i[0] for i in box]
    Ys = [i[1] for i in box]
    x1 = min(Xs)
    x2 = max(Xs)
    y1 = min(Ys)
    y2 = max(Ys)
    center = ((x1+x2)/2,(y1+y2)/2)
    size = (x2-x1, y2-y1)
    cropped = cv2.getRectSubPix(img, size, center)
    angle = rect[2]
    if angle != 90:
        if angle > 45:
            angle = 0 - (90 - angle)
        else:
            angle = angle
        M = cv2.getRotationMatrix2D((size[0] / 2, size[1] / 2), angle, 1.0)
        cropped = cv2.warpAffine(cropped, M, size)
        croppedW = H if H > W else W
        croppedH = H if H < W else W
        croppedRotated = cv2.getRectSubPix(cropped, (int(croppedW), int(croppedH)),
(size[0] / 2, size[1] / 2))
        return croppedRotated
    return cropped
```

A **3.kódrészletben** A **crop_rect** függvényt láthatjuk, amely paraméternek a feljebb meghatározott téglalapokat, azoknak a sarokpontjait és az eredeti képet várja. Először kiszedjük külön változóba a téglalap szélességét, magasságát, majd kiszámoljuk a közepét és a méretét.

Mivel a koordináták nem egész számok, így a **cv2.getRectSubPix** függvényt kell használnunk a képek kivágásához.

Majd megvizsgáljuk, hogy az adott téglalap szöge egyenlő-e 90 fokkal, ha nem akkor szükségünk lesz egy forgatási mátrixra kell az **Affin transzformáció**hoz erre a legegyszerűbb függvény a **cv2.getRotationMatrix2D** ami, paraméternek az elforgatás középpontját **(size[0] /2, size[1]/2**), a forgatás szögét (**angle**) és egy skála értéket (**1.0**) vár. Az **Affin transzformáció**hoz a **cv2.warpAffine** függvényt alkalmazzuk, ehhez szükségünk van a forrásképre (**cropped**), az előbb létrehozott forgatási mátrixra (**M**) és méretre (**size**).[4]

Végül újra meghívjuk a **cv2.getRectSubPix** függvényt, hogy az elforgatott képünket a számunkra legmegfelelőbb méretben kivágjuk, hogy a dekódolásnál minél kevesebb zavaró tényező legyen.

3.2 Dekódolás

3.2.1 decode függvény

4. kódrészlet

```
import cv2

def decode(img):
    gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
    ret, thresh =cv2.threshold(gray, 200, 255, cv2.THRESH_BINARY_INV +
cv2.THRESH_OTSU)
    ean13 = None
    is_valid = None
    for i in range(img.shape[0]-1,0,-1):
        try:
            ean13, is_valid = decode_line(thresh[i])
        except Exception as e:
            pass
        if is_valid:
            break

    return ean13, is_valid, thresh
```

A **4.kódrészletben** A **decode** függvényt találjuk, amely a **cv2.cvtcolor** függvénnyel szürke árnyalatúvá alakítja a képünket, majd a **cv2.threshold** függvény segítségével egy küszöbölést.

Utána egy **for** ciklusban 1 pixel vastag szeletekre szedjük a képet és meg hívjuk rá a **decode_line** függvényt ennek működését az 5.kódrészletben látható. A **decode_line** függvény visszaadja, hogy talált-e vonalkódot és ha igen akkor még magát a dekódolt számsort.

3.2.2 **decode_line függvény**

5. kódrészlet

```
def decode_line(line):
    bars = read_bars(line)
    left_guard, left_patterns, center_guard, right_patterns, right_guard =
classify_bars(bars)
    convert_patterns_to_length(left_patterns)
    convert_patterns_to_length(right_patterns)
    left_codes = read_patterns(left_patterns,is_left=True)
    right_codes = read_patterns(right_patterns,is_left=False)
    ean13 = get_ean13(left_codes,right_codes)

    is_valid = verify(ean13)
    return ean13, is_valid
```

A **5.kódrészletben** a **decode_line** függvényt találjuk, ez a függvény magában annyira nem érdekes, meghívja a többi dekódoláshoz szükséges függvényt, és a végén visszaadja eredményül, hogy talált-e vonalkódot és ha igen akkor még magát a dekódolt számsort. A dekódolást végző függvényeket a következő kódrészletekben mutatom be.

3.2.3 **read_bars, replace_255_to_1 függvény**

6. kódrészlet

```
def read_bars(line):
    replace_255_to_1(line)
    bars = []
    current_length = 1
    for i in range(len(line)-1):
        if line[i] == line[i+1]:
            current_length = current_length + 1
        else:
            bars.append(current_length * str(line[i]))
            current_length = 1
    bars.pop(0)
    return bars

def replace_255_to_1(array):
    for i in range(len(array)):
        if array[i] == 255:
            array[i] = 1
```

A **6.kódrészletben** a **read_bars** és a **replace_255_to_1** függvények láthatóak a **replace_255_to_1** egyszerűen végigmegy az átadott tömbön és minden kétszázötvenöt intenzitású pixelt egyre cserél. Ezután a **read_bars** függvényben összeszámoljuk a nullák és egyesek számát majd abban a sorrendben ahogy voltak hozzáadjuk a bars tömbhöz, így megkapjuk az egymást követő vonalakat.

3.2.4 **classify_bars, convert_patterns_to_length függvény**

7. kódrészlet

```
def classify_bars(bars):
    left_guard = bars[0:3]
    left_patterns = bars[3:27]
    center_guard = bars[27:32]
    right_patterns = bars[32:56]
    right_guard = bars[56:59]
    return left_guard, left_patterns, center_guard, right_patterns, right_guard

def convert_patterns_to_length(patterns):
    for i in range(len(patterns)):
        patterns[i] = len(patterns[i])
```

A **7.kódrészletben** a **classify_bars** függvény látható, amiben egyszerűen szét bontjuk a **read_bars** függvény által létrehozott számsorozattokat attól függően, hogy a vonalkód mely részét képezi. A **convert_patterns_to_length** függvény megszámolja a nullák és egyesek darab számát majd behelyettesítve belerakja a tömbbe.

3.2.5 **read_patterns, get_AT függvény**

8. kódrészlet

```
def read_patterns(patterns,is_left=True):
    codes = []
    for i in range(6):
        start_index = i*4
        sliced = patterns[start_index:start_index+4]
        m1 = sliced[0]
        m2 = sliced[1]
        m3 = sliced[2]
        m4 = sliced[3]
        total = m1+m2+m3+m4;
        tmp1=(m1+m2)*1.0;
        tmp2=(m2+m3)*1.0;
        at1 = get_AT(tmp1/total)
        at2 = get_AT(tmp2/total)
        if is_left:
            decoded = decode_left(at1,at2,m1,m2,m3,m4)
        else:
            decoded = decode_right(at1,at2,m1,m2,m3,m4)
        codes.append(decoded)
    return codes
def get_AT(value):
    if value < 2.5/7:
        return 2
    elif value < 3.5/7:
        return 3
    elif value < 4.5/7:
        return 4
    else:
        return 5
```

A **8.kódrészletben** a **read_patterns** és a **get_AT** függvény található. A **read_patterns** paraméternek a **convert_patterns_to_length** függvény által feldolgozott tömböt, és bool változott várja, amiből eldönti, hogy bal vagy jobb oldali mintáról van szó.

Összesen huszonnégy eleme van egy tömbnek ,egy szám dekódolásához négy elemre van szükség.

Elkezdünk négyesével végig menni rajta, kiszámoljuk először a négy szám összegét, majd ezt az értéket elosztjuk az első két szám összegével, utána pedig a második és a harmadik szám összegével. Így megkapjuk, hogy az adott fekete-fehér vonalpáros(pl: **m1-m2**) hossza, hány százalékát teszi ki az éppen általunk dekódolt érték a teljes hosszából(**total**).

A **get_AT** függvénnyel meghatározzuk, hogy az adott százalékhoz mekkora szám párosul, majd ezeket is átadva meghívjuk a **decode_left** és **decode_right** fügvényeket.

3.2.6 **decode_left, decode_right függvény**

9. kódrészlet

```
def decode_left(at1, at2, m1, m2, m3, m4):
    patterns = {}
    patterns["2,2"]={"code":"6","parity":"O"}
    patterns["2,3"]={"code":"0","parity":"E"}
    patterns["2,4"]={"code":"4","parity":"O"}
    patterns["2,5"]={"code":"3","parity":"E"}
    patterns["3,2"]={"code":"9","parity":"E"}
    patterns["3,3"]={"code":"8","parity":"O","alter_code":"2"}
    patterns["3,4"]={"code":"7","parity":"E","alter_code":"1"}
    patterns["3,5"]={"code":"5","parity":"O"}
    patterns["4,2"]={"code":"9","parity":"O"}
    patterns["4,3"]={"code":"8","parity":"E","alter_code":"2"}
    patterns["4,4"]={"code":"7","parity":"O","alter_code":"1"}
    patterns["4,5"]={"code":"5","parity":"E"}
    patterns["5,2"]={"code":"6","parity":"E"}
    patterns["5,3"]={"code":"0","parity":"O"}
    patterns["5,4"]={"code":"4","parity":"E"}
    patterns["5,5"]={"code":"3","parity":"O"}
    pattern_dict = patterns[str(at1) + "," + str(at2)]
    code = 0
    use_alternative = False
    if int(at1) == 3 and int(at2) == 3:
        if m3+1>=m4:
            use_alternative = True
    if int(at1) == 3 and int(at2) == 4:
        if m2+1>=m3:
            use_alternative = True
    if int(at1) == 4 and int(at2) == 3:
        if m2+1>=m1:
            use_alternative = True
    if int(at1) == 4 and int(at2) == 4:
        if m1+1>=m2:
            use_alternative = True
    if use_alternative:
        code = pattern_dict["alter_code"]
    else:
        code = pattern_dict["code"]
    final = {"code": code, "parity": pattern_dict["parity"]}
    return final
```

A **9.kódrészletben** a **decode_left** függvényt láthatjuk. Ha **at1** és **at2** értéke három, négy vagy ezek bármely kombinációja akkor meg kell vizsgáljuk, hogy módosított értéket kell-e nekik adnunk az előre definiált mintákból(**patterns**).

A vizsgálat után megnézzük, hogy mely mintára illeszkedik az **at1, at2** értéke akkor megkapjuk az immáron dekódolt számunkat, és a hozzá tartozó paritást, amivel később meghatározzuk az első számjegy értékét.

10. kódrészlet

```
def decode_right(at1, at2, m1, m2, m3, m4):
    patterns = {}
    patterns["2,2"]={"code":"6"}
    patterns["2,4"]={"code":"4"}
    patterns["3,3"]={"code":"8","alter_code":"2"}
    patterns["3,5"]={"code":"5"}
    patterns["4,2"]={"code":"9"}
    patterns["4,4"]={"code":"7","alter_code":"1"}
    patterns["5,3"]={"code":"0"}
    patterns["5,5"]={"code":"3"}
    pattern_dict = patterns[str(at1) + "," + str(at2)]
    code = 0
    use_alternative = False
    if int(at1) == 3 and int(at2) == 3:
        if m3+1>=m4:
            use_alternative = True
    if int(at1) == 4 and int(at2) == 4:
        if m1+1>=m2:
            use_alternative = True
    if use_alternative:
        code = pattern_dict["alter_code"]
    else:
        code = pattern_dict["code"]
    final = {"code": code}
    return final
```

A 10.kódrészletben a **decode_right** függvény ugyan ezt hajtja végre annyi különbséggel, hogy itt már kevesebb mintával kell dolgoznunk, és itt már nem nincsen paritás. Az első szám értékét a vonalkód bal részén határozzuk csak meg.

3.2.7 **get_ean13, get_first_digit, verify függvény**

11. kódrészlet

```
def get_ean13(left_codes,right_codes):
    ean13 = ""
    ean13 = ean13 + str(get_first_digit(left_codes))
    for code in left_codes:
        ean13 = ean13 + str(code["code"])
    for code in right_codes:
        ean13 = ean13 + str(code["code"])
    return ean13

def get_first_digit(left_codes):
    parity_dict = {}
    parity_dict["OOOOOO"] = 0
    parity_dict["OOEOEE"] = 1
    parity_dict["OOEEOE"] = 2
    parity_dict["OOEEEO"] = 3
    parity_dict["OEOOEE"] = 4
    parity_dict["OEEOOE"] = 5
    parity_dict["OEEEOO"] = 6
    parity_dict["OEOEOE"] = 7
    parity_dict["OEOEEO"] = 8
    parity_dict["OEEOEO"] = 9
    parity = ""
    for code in left_codes:
        parity = parity + code["parity"]
    return parity_dict[parity]

def verify(ean13):
    weight = [1,3,1,3,1,3,1,3,1,3,1,3,1,3]
    weighted_sum = 0
    for i in range(12):
        weighted_sum = weighted_sum + weight[i] * int(ean13[i])#szorzás 1 és 3-al
    weighted_sum = str(weighted_sum)
    checksum = 0
    units_digit = int(weighted_sum[-1])
    if units_digit != 0:
        checksum = 10 - units_digit
    else:
        checksum = 0
    print("The checksum of "+ean13 + " is " + str(checksum))
    if checksum == int(ean13[-1]):
        print("The code is valid.")
        return True
    else:
        print("The code is invalid.")
        return False
```

A **11. kódrészletben** a **get_ean13** függvényben állítjuk össze a végleges kódunkat. Először meghívjuk a **get_first_digit** függvényt, ami a **decode_left** függvényben létrehozott paritásból megmondja az első szám értékét. Majd **for** ciklussal végig megyünk az előzőleg dekódolt két kódrészen és összefűzzük őket.

A **verify** függvényben végezzük az ellenőrzést, itt derül ki, hogy sikeres volt-e a dekódolás.

Összeadjuk a páros számú számjegyeket, majd hárommal szorozzuk az eredményt. Utána összeadjuk a páratlan szám számjegyeket és összeadjuk az előző számolás eredményével, majd megvizsgáljuk, hogy maradék nélkül osztható-e tízzel, ha nem akkor kiszámoljuk, hogy mennyit kéne hozzáadni, hogy tíz többszöröse legyen, ez a szám lesz a checksum. Ha checksum megegyezik a számsorunk utolsó számával akkor sikeres volt a dekódolás.[5]

Tesztelés

4.1 **Táblázat(részlet)**

1. Táblázat: tesztek eredménye

05102009203.jpg		Nem
05102009205.jpg		Igen
05102009206.jpg		Nem
05102009207.jpg		Igen
05102009208.jpg		Igen
05102009209.jpg		Igen
05102009210.jpg		Igen
05102009211.jpg		Igen
05102009212.jpg		Nem
05102009213.jpg		Igen
05102009214.jpg		Igen
05102009215.jpg		Nem
05102009216.jpg		Nem
05102009218.jpg		Nem
05102009219.jpg		Nem
05102009220.jpg		Nem
05102009222.jpg		Nem
05102009223.jpg		Igen
05102009224.jpg		Igen
05102009225.jpg		Nem
05102009226.jpg		Igen
05102009227.jpg		Nem
05102009228.jpg		Nem
05102009229.jpg		Nem
06102009240.jpg		Igen
06102009241.jpg		Igen
Sikeres:	69	60,00%
Sikertelen:	46	40,00%
Összes:	115	

4.2 **Tesztelés eredménye**

Az **1. táblázatban** láthatóak a teszt eredmények utolsó sora, a teljes Excel tábla csatolva lesz a Word dokumentumhoz. Összesen 115 darab képen lett tesztelve, ebből 69 azaz 60% sikerült helyesen detektálni és dekódolni. A legtöbb hibát a rossz detektálás okozta, ha picit takarásban van a vonalkód, vagy nagyobb távolságról lett fotózva a kép akkor már problémát jelent, hogy a további feldolgozásra alkalmas módon ki lehessen vágni. Detektálásnál több kernel méretet is kipróbáltam, ezek közül a 3x10 ami a legjobban bevált, itt azért elég sokat sikeresen olvasott és a sebességgel sem volt gond.

Felhasználói leírás

A program futtatásához kelleni fog egy fejlesztői környezet , én a PyCharmot használtam, de bármely más is megfelelő lehet, ahol lehet python csomagokat telepíteni.

Telepítendő csomagok:

- cv2
- numpy
- os

Csomag telepítés után, ha detektálni szeretnénk, akkor a detect.py fájl kell futtatnunk, ha olvasni szeretnénk, akkor a read.py.

Mind a két fájl végén található cv2.imread() függvényben kell megadni az általunk használni kívánt képet, figyelni kell a helyes elérési útvonalra és a fájl formátumra. Formátumok közül a jpg,jpeg,png-t teszteltem, ezeket ajánlott használni.

Felhasznált irodalom

[1] https://www.codeinfo.hu/blog/mi-is-valojaban-a-vonalkod
[2] https://ident.hu/vonalkod-alapok
[3] https://www.inf.u-szeged.hu/~tanacs/pyocv/tmretezs.html
[4] https://www.inf.u-szeged.hu/~tanacs/pyocv/affin_transzformcik.html
[5] https://ident.hu/vonalkod-alapok