Gépi látás

GKNB_INTM038

Pénzérme számláló alkalmazás

github.com/tivadark/Computer_vision

Kamondy Tivadar

ZXCJX6

Győr, 2020/2021/1

Tartalomjegyzék

1 Bevezetés

Az alábbiakban összefoglalva a Széchenyi István Egyetem Bsc mérnökinformatikus szak Gépi látás (GKNB_INTM038) kurzusára elkészített féléves beadandó feladatomról számolok be. A féléves dolgozatom célja egy olyan alkalmazás elkészítése volt, amellyel a számítógépes képfeldolgozást illetve a gépi látás egyes elemeit magába foglalva, szemléltetni tudom egy működő megvalósításon keresztül. Ebből a célból így tehát egy pénzérme számláló alkalmazást készítettem el. Az program lehetővé teszi a felhasználó számára, hogy egy bemeneti képet megadva konkrét eredményeket kapjon. Pontosabban egy olyan bemeneti kép adható meg, amelyen különböző értékű és tetszőleges értékű pénzérmék találhatók meg, részben előre definiált paraméterekkel. A továbbiakban a leírást két részre csoportosítva, a felhasználói és fejlesztői szempontból közelítve is kifejtem részletesebben.

```
>>> %Run newCoins.py

  200 Ft : x 3
  20 Ft : x 3
  50 Ft : x 2
  10 Ft : x 2
  100 Ft : x 2
  5 Ft : x 4
  Teljes osszeg:  1000  Ft

>>>
```

Felhasználó számára megjelenítendő eredmény.

Balra a bemeneti, jobb oldalon a kimeneti képek.

2 Felhasználói dokumentáció

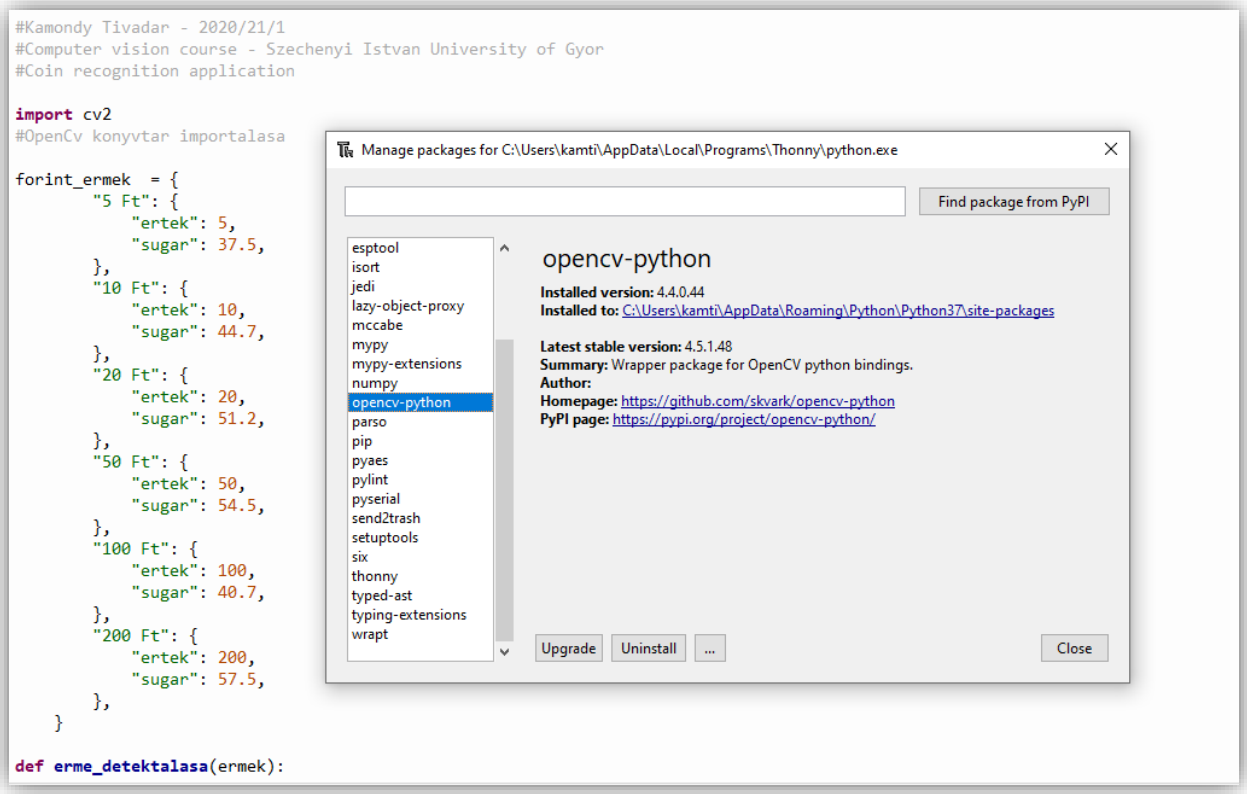
2.1 Fejlesztői környezet - Thonny

Az alkalmazás fejlesztését a Thonny intergrált fejlesztői környezet segítségével valósítottam meg. Ehhez a Pyhton programozási nyelvet választottam, azon belül is a 3.7.7-es verziót, ami egész pontosan egy 2020 márciusi kiadás. A phyton egy széles körben elterjedt programozási nyelv, amely számos előnyt képes biztosítani a többi nyelvvel szemben a számítógépes látás területén, de széles körben használják a mesterséges intelligencia és gépi tanuláshoz is. A Thonny fejlesztői környezet segítségével nem volt szükség külön a programozási nyelvi csomagot telepítésére, mivel az előbb említett verzióval beépítve települ a környezet is. A környezet egy egyszerű és letisztult felületet biztosít, ahova a kódot beírva egyszerűen fordításra majd futtatásra is van lehetőség. Tulajdonságai miatt akár kezdőknek is opcionális választás lehet.

Forrás: **[1]**

2.2 Felhasznált könyvtár - OpenCV

A feladat megvalósításához felhasználtam az OpenCV (Open Source Computer Vision Library) könyvtárat, amely segítségével akár valós idejű számítógépes képfeldolgozás alkalmazások fejleszthetők. A könyvtár főleg a számítógépes képfeldolgozásra fókuszál, de támogatja az olyan funkciókat is, mint például ami a videófelvételek elemzését vagy az arc- illetve tárgy felismerést segítik.



Az OpenCV könyvtárat a Thonny fejlesztői környezeten belül is lehetőség van hozzáadni, így nem kell külön letölteni majd importálni semmit. Egyszerűen a menüpontból kiválasztva **Tools → Manage packages**, és az alábbi listából kiválasztva az **opencv-pyhton** csomag telepíthető a fejlesztői környezethez. Itt lehetőség van később a már telepített csomagok kezelésére is, mint ahogy a képen látható az aktuális telepített verzió 4.4.0.44, de azóta már jött ki egy újabb, egyszerűen az *Upgrade* gomb megnyomásával frissíthető ez. De lehet egyéb olyan csomagokat is telepíteni amelyek nincsenek bent a listában, vagy egyszerűen eltávolítani azokat amelyekre már nincsen szükség.

Forrás: **[2]**

2.3 Használat

2.3.1 A program paraméterezése és beállításai

Az alkalmazás pontos működéséhez fontos, hogy a futtatást megelőzően a felhasználó beállítson illetve finomhangoljon néhány olyan értéket, amelyek változhatnak az indítások során. Mivel az algoritmus a kamera rögzített beállított pozíciója alapján dolgozik, ezért első lépésként fontos az egyes pénzérmékhez tartozó értékek vizsgálata. Ez képkockánként változó, tehát megnézzük hogy az adott pénzérmének mekkora az átmérője képkockában, ezt felezzük így megkapjuk a körhöz tartozó sugár értékét, majd esetleg a középértékeken még javítva az arányok jobb elosztása miatt módosítunk az értékeken. Ez a lépés könnyen megtehető akár a Paint segítségével, így kijelölünk a vizsgálandó érmét, majd megvizsgáljuk hogy az átmérője vízszintesen mennyi képkockát foglal magába. Így tehát ha a fentebb is található tesztképekre szeretnénk megkapni a helyes értékeket, ezek szerint a beállított helyes paraméterezés:

```
forint_ermek  = {
        "5 Ft": {
            "ertek": 5,
            "sugar": 37.5,
        },
        "10 Ft": {
            "ertek": 10,
            "sugar": 44.7,
        },
        "20 Ft": {
            "ertek": 20,
            "sugar": 51.2,
        },
        "50 Ft": {
            "ertek": 50,
            "sugar": 54.5,
        },
        "100 Ft": {
            "ertek": 100,
            "sugar": 40.7,
        },
        "200 Ft": {
            "ertek": 200,
            "sugar": 57.5,
        },
    }
```

A felhasználó szempontjából másik szükséges információ az úgynevezett küszöbérték beállítása lehet, amelynek segítségével a képek közötti eltérések csillapíthatók. Tehát ennek segítségével a hibaszűrés csökkenthető illetve nagyítható bizonyos értékig.

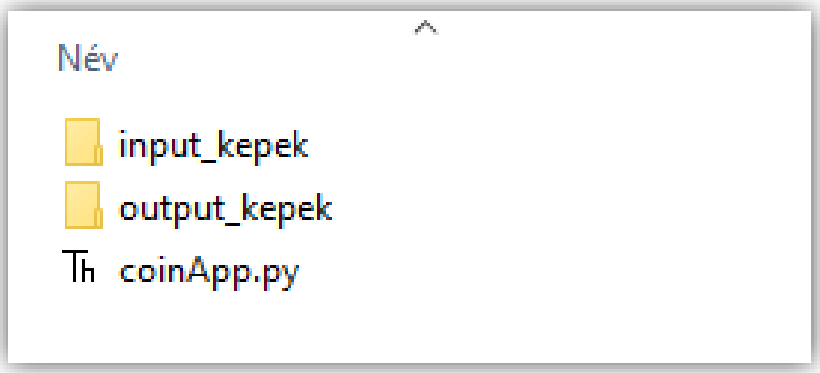
```
80      adat,kep = osszeg_szamitas(ermek_adat,1.5,ermek_detektalt_kep)
```

Szintén érdemesebb nagyobb felbontású képek esetén ezt a számot magasabbra emelni, hiszen a pixelek sűrűbben helyezkednek el ilyen estekben, így növelni kell a csillapítást. Viszont ha a képeink például egy ipari felhasználásban, jól- vagy tökéletesen megvilágított környezetben kerülnek elkészítésre akkor ennek az értéke is lehet minél kisebb, hiszen nincs szükség ilyenkor a nagyobb hibaküszöb meghatározásához a környezeti tényezők miatt.

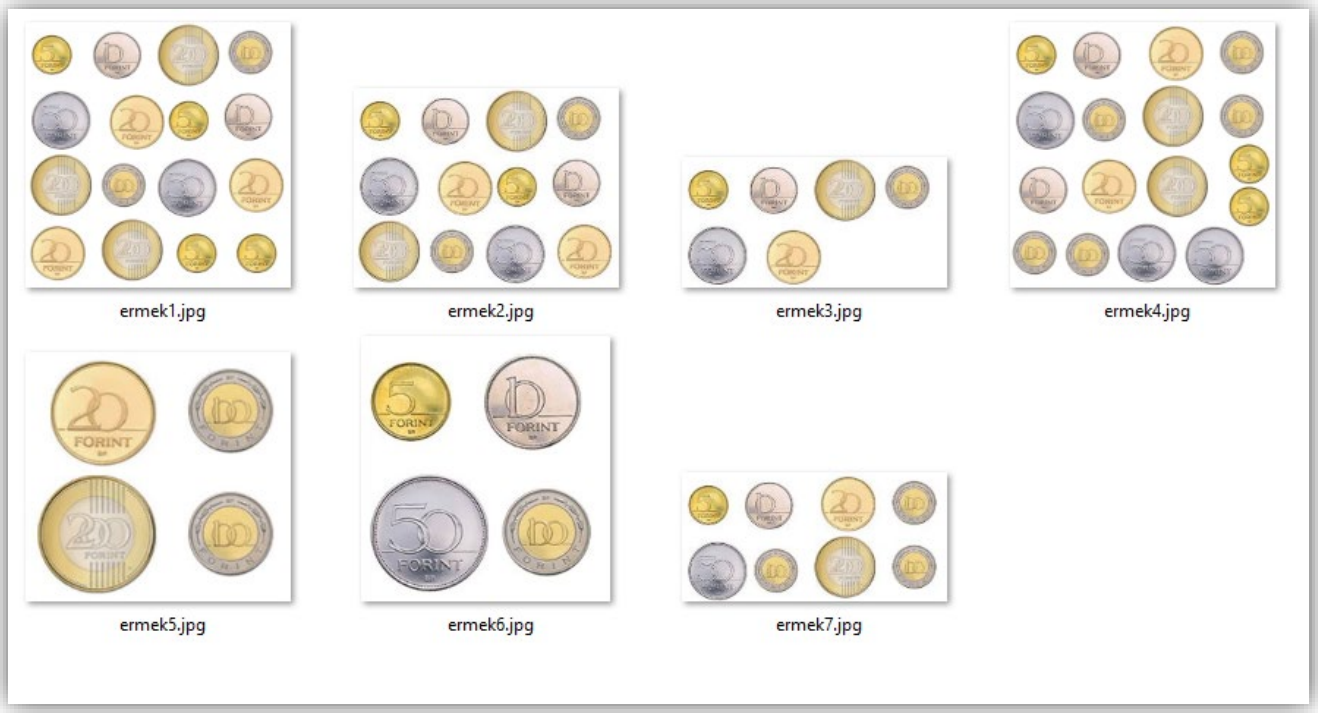
2.3.1.1 Képek ki- és bemeneti helyének beállítása

Az alkalmazás használata ez előzőekben tárgyalt Thonny fejlesztői környezettel nagyon egyszerű, viszont fontos hogy a környezet az előzőek szerint helyesen legyen beállítva,

tehát a könyvtár importálása is megtörténjen. Továbbá a használathoz az alábbi mappaszerkezet és elnevezéseket kell követni tehát:



A futtatható állomány mellett a két mappa megléte, az egyik a felhasználó által generált bemeneti képeket kell, hogy tartalmazza.



Ezeknek szintén célszerű figyelni a nevére, hiszen a kódban az OpenCV, az elérési hely és név alapján olvassa be mindig az aktuális bemeneti képet. Így tehát, hogy a program pontosan melyik képet importálja be a könyvtárunkból, a 72. sorban, a main tagot követően lehet beálltani.

```
78  if __name__ == "__main__":
79      kep = cv2.imread('input_kepek/ermek121.jpeg')
```

Itt csupán a futtatható állomány melletti mappa szerkezetet kell megadni, tehát adott esetben az *input_kepek* nevezetű mappából az *ermek121* jpeg kiterjesztésű képet fogja a program választani bemeneti képként. Hasonlóan így tehát az exportálandó képek helyével, és elnevezésével is:

```
82      cv2.imwrite("output_kepek/ermek_Hough.jpg",ermek_detektalt_kep)
```

Itt az egyes detektált érmék lementése, amelyeken végrehajtódott a Hough transzformáció.

```
cv2.imwrite("output_kepek/ermek_darabertek_korvonal.jpg",kep)
```

Az alábbi résznél pedig a végleges exportálandó eredmény, amelyeken szerepelnek már az egyes érmékhez tartozó értékek is. Függvény második paramétereként itt tehát már a kész képet kapja meg.

Miután ez mind egyezik, futtatható a program és az eredmény a parancssorban látható, illetve megjelennek a létrejött képek is a kimeneti mappában.

3 Fejlesztői dokumentáció

3.1 Programkód és felépítés

Az algoritmus felépítése négy részre bontható, ahol az első részben definiálásra kerülnek a különböző pénzérmékhez tartozó adatok a későbbi számítások egyszerűsítése végett. A második és harmadik részben pedig két fő funkció került implementálásra, ahol az előbbi a pénzérmék detektálásáért, az utóbbi pedig az érmék értékeinek kiszámításaiért felelős. A kód legutolsó részében meghatározásra kerülnek a különböző útvonalak, illetve meghívásra kerülnek a függvények, majd végül egy teljes összeg számítás segítségével a felhasználó számára kiíratásra kerülnek az eredmények.

```
#Kamondy Tivadar - 2020/21/1
#Computer vision course - Szechenyi Istvan University of
Gyor
#Coin recognition application

import cv2
#OpenCv konyvtar importalasa

forint_ermek  = {
        "5 Ft": {
            "ertek": 5,
            "sugar": 37.5,
        },
        "10 Ft": {
            "ertek": 10,
            "sugar": 44.7,
        },
        "20 Ft": {
            "ertek": 20,
            "sugar": 51.2,
        },
        "50 Ft": {
            "ertek": 50,
            "sugar": 54.5,
        },
        "100 Ft": {
            "ertek": 100,
            "sugar": 40.7,
        },
        "200 Ft": {
            "ertek": 200,
            "sugar": 57.5,
        },
    }
```

3.1.1 Első kódrészlet

Az alábbi kódrészletben fentebb meghatározom a pénzérmékhez tartozó különböző adatokat, mint például a név, érték, és a különféle méretű érmékhez tartozó sugár. Ezt a python (dictionary) szótár segítségével valósítom meg, mivel itt tudok tárolni különböző adat párokat kulcs és érték szerint, így később könnyebb lesz dolgozni ezekkel. A szótárakban nem megengedettek a duplikált adatok, tehát nem lehet két ugyanolyan értékhez tartozó kulcs, továbbá nincs sorrend sem ezért nem lehet indexelés alapján rákeresni egy-egy értékre.
A kódrészlet első sorában pedig egyszerűen importálódik az OpenCV csomag, ezáltal a későbbiekben lehetőség van a használatára.

```
def erme_detektalasa(ermek):
    kep = cv2.cvtColor(ermek, cv2.COLOR_BGR2GRAY) #a kep
szurkearnyalatossa konvertalasa
    kep = cv2.medianBlur(kep, 21) #a kep elhomalyositasa a
kep aprobb adatainak eltorlese vegett

    kor_alakzatok = cv2.HoughCircles(
        kep,  # bemeneti kep
        cv2.HOUGH_GRADIENT,  # detektalas tipusa
        1,
        50,
        param1=100,
        param2=50,
        minRadius=5,  # minimalis sugar meret
        maxRadius=350,  # max sugar
    )

    #kor_alakzatok detektalasa es a zold korvonal
kirajzolasa korulottuk
    for felismert_kor_alakzatok in kor_alakzatok[0]:
        x_koord, y_koord, detektalt_sugar =
felismert_kor_alakzatok
        detektalt_ermek = cv2.circle( #a kor megrajzolasa a
koordinatak es sugarak felhasznalasaval
            ermek,
            (int(x_koord), int(y_koord)),
            int(detektalt_sugar),
            (0, 255, 0),
            4,
        )

    return kor_alakzatok[0],detektalt_ermek
```

3.1.2 Második kódrészlet

A kód további részeit függvényekben határozom meg, így ezek későbbi meghívásával egyszerűbben lehet számolni, valamint átláthatóbb lesz a program. Ebben a részben, mint ahogy a metódus nevében is látható az érméknek a detektálása a cél. A függvénynek itt egy paramétere van, amit megkap, ez pedig maga a kép, amit a fő függvényben (majd az utolsó kódrészletben) kapunk meg az OpenCV könyvtár segítségével. Miután a függvény rendelkezik a képpel, már dolgozni is tud vele, így a minél hatékonyabb képfeldolgozás érdekében különféle átalakításokat kell végezni a képen. Az egyik ilyen átalakítás, mint például a kép szürkeárnyalatossá konvertálása, majd ezután pedig a kép elhomályosítása. A kép szürkeárnyalatossá való alakítása során tehát a kép saját színtere kerül átalakításra. OpenCV beépített funkcióját használva az első paraméterként az aktuális képkockát, a másodikként pedig a konverzió típusát kapja a függvény. A kép elhomályosítása szintén az OpenCV egyik funkciójának a segítségével történik. A mediánszűréssel kiszámítható minden egyes képkocka az adott kernel ablak alatt, ezáltal bizonyos képkockák kicserélhetők a medián értékekkel, ahol esetlegesen túl nagy az eltérés. Ez rendkívül hatékony módszer a só-bors zaj eltávolításához. Érdekesség, hogy amíg a Gaussian és doboz szűrésnél a szűrt érték lehet olyan értékű, ami az eredeti képen nincs jelen, addig ennél a megoldásnál csak olyan elemmel számolhatunk, ami az eredeti képen is megtalálható, ezáltal még hatékonyabban csökkentve a zajt.

Forrás: **[3]**

A következő lépésben, a kódban a Hough transzformáció segítségével megkeressük a képen található kör alakzatokat. Egy kör matematikailag leírható az $(x - x_{középpont})^2 + (y - y_{középpont})^2 = y^2$ alábbi egyenlettel, ahol $(x_{középpont}, y_{középpont})$ a kör középpontját jelöli, és az r pedig a kör sugara. Az OpenCV azonban egy jobb megoldást használ, az egyszerűsítés miatt, így a Hough Gradiens metódussal számol ami az élek gradiens információit használja. Az ehhez tartozó két paramétert a funkcióban meg kell adni *param1* illetve *param2* néven. Minél nagyobb az itt beállított küszöbérték, annál nagyobb a lehetősége a felismert körök számának. Viszont a megfelelő értéket fontos megtalálni, hiszen ha nincs meg akkor hamis eredmények is kerülhetnek a számításba. Szintén fontos a minimum távolság paraméter helyes beállítása, ennek is fontos szerepe van abban hogy az összes kör, illetve hibás észlelések nélkül menjen végre a számítás.

Forrás: **[4]**

Ezt követően egy ciklus segítségével az algoritmus végig megy a meglévő kör alakzatokon és minden egyes detektáláson végighalad, majd egy egyszerű OpenCV funkció segítségével kiemeli az egyes körök sugarait, amelyeket detektált az előző lépésben. A funkció paraméterei sorrendben, mint a kép, kör koordinátái, sugár, megrajzolandó jelölés színe, vastagság. Visszatérési érték pedig egy adott detektált elem, kép lesz.

```
def osszeg_szamitas(ermek_adat,kuszobertek
,korvonalas_ermek):
    adat = {} #exportalando szotar
    for erme_adat in ermek_adat: #ermeken vegig iteralas
        for k in forint_ermek : #vegig iteralas az ermek
eloredefinialt adatain
            if abs(erme_adat[2] - forint_ermek
[k]["sugar"]) <= kuszobertek: #ha talalkozik a
kuszobertekkel
                if k in adat.keys(): #ha a kulcs nem
talalhato akkor hozzon letre egy ujat
                    adat[k] += 1
                else:
                    adat[k] = 1 #ermekhez hozzaadas
                cv2.putText(korvonalas_ermek,
str(forint_ermek [k]["ertek"]), (int(erme_adat[0]),
int(erme_adat[1])), cv2.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 0),
4) #az adott ermere az ertek kiiratasa
    return adat, korvonalas_ermek #adat es a kep
visszaadasa mar az ertekekkel egyutt
```

3.1.3 Harmadik kódrészlet

Ebben a részben az egyes érmékhez tartozó értékek is számításra kerülnek. Itt az első kódrészletben meghatározott szótár is felhasználásra kerül, hiszen az egyes érmékhez tartozó értékek is itt vannak meghatározva. Egy ciklus segítségével végig iterálunk ezeken az adatokon, és megkeressük, hogy az adott érméhez melyik sugár tartozik, olyan módon hogy a ciklus megkeresi az egyes érmékhez tartozó sugarakat, amelyek a szótárban definiálásra kerültek. A kód működésének hatékonysága növelésének érdeke miatt itt szükség volt egy küszöbértékre, így még nagyobb pontossággal határozhatók meg a különböző érmék értékei. Tehát ha a vizsgált sugár, a szótárban hozzá legközelebb eső párjához nagyban hasonlít akkor valószínűsíthető az, hogy egyezés van, így tehát a program megtalálta az adott érméhez tartozó értéket. Ezt követően egy OpenCV funkció segítségével a már előzőleg felhasznált és módosított eredeti bemeneti képre rákerülnek az egyes érmékhez tartozó értékek is.

```
if __name__ == "__main__":
    kep = cv2.imread('input_kepek/ermek1.jpg')

    ermek_adat,ermek_detektalt_kep = erme_detektalasa(kep)
#ermek detektalasa

cv2.imwrite("output_kepek/ermek_Hough.jpg",ermek_detektalt_
kep)
    adat,kep =
osszeg_szamitas(ermek_adat,1.5,ermek_detektalt_kep) #az
ermek osszegenek szamitasa
    teljes_osszeg = 0

    for a in adat:
        print(a,": x", adat[a])
        teljes_osszeg += adat[a] * int(a.split(" ")[0])
    print("Teljes osszeg: ", teljes_osszeg)

cv2.imwrite("output_kepek/ermek_darabertek_korvonal.jpg",ke
p)
```

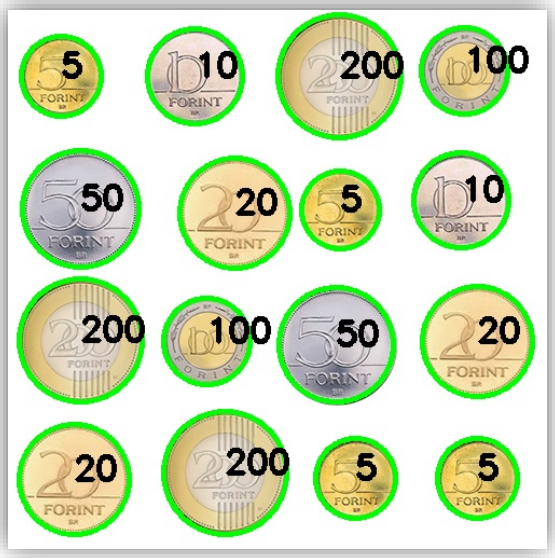
3.1.4 Negyedik kódrészlet

A legutolsó részben pedig definiáljuk a bemeneti és kimeneti képek elérési útvonalait az OpenCV-nek, illettve az egyes elkészített metódusok meghívásra kerülnek. Ezt követően egy iterálás segítségével a kód végig megy az értékeken, és egy összegzést csinál, így eredményként a felhasználó nem csak a kimeneti képeket kapja meg, amiken már a pontos adatok szerepelnek, hanem egyúttal a konzolban látható számára a teljes összeg, és hogy egy adott valutából mennyi darab is található a képen.

4 Összegzés, tapasztalatok

Az alkalmazás fejlesztése során számos kérdés és gondolat felvetődött, hogy hogyan is lehetne az egész algoritmust még hatékonyabbá fejleszteni. A jelenlegi megoldással a megadott különböző darabszámú, elhelyezésű és értékű pénzérmékkel tesztelve lett. Ezekkel a program minden esetben helyesen számolt, hibátlanul felismerte az pénzérméket és nem hibázott. Továbbá ezelőtt tesztelve volt különböző olyan képekkel, amelyekben több a zavar vagy az olyan befolyásoló tényező, mint például az árnyékok. Azonban a legtöbb esetben itt is jó eredmény adott vissza az alkalmazás, viszont voltak olyan esetek, amelyeknél eltérő értékek jöttek eredményül, hiszen az árnyékok különböző szögben és mértékben módosították a hátteret és erre már sok esetben nem elég a sima számítógépes képfeldolgozás, hanem a tökéletes eredmény érdekében gépi tanulás megoldásokkal lehet javítani a sikerességet.

4.1 Tesztelés különböző képekkel




5
10
200
100
50
20
5
10
200
100
50
20


5
10
200
100
50
20

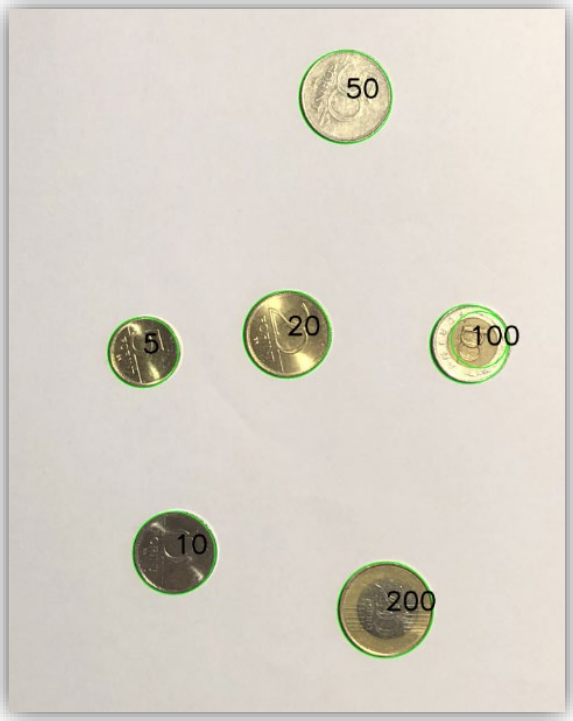

5
10
20
100
50
100
200
100
5
10
20
200
5
100
100
50
50

A felismerés enyhén árnyékolt pénzérmés képekkel is működik.


50
5
100
10
200
20

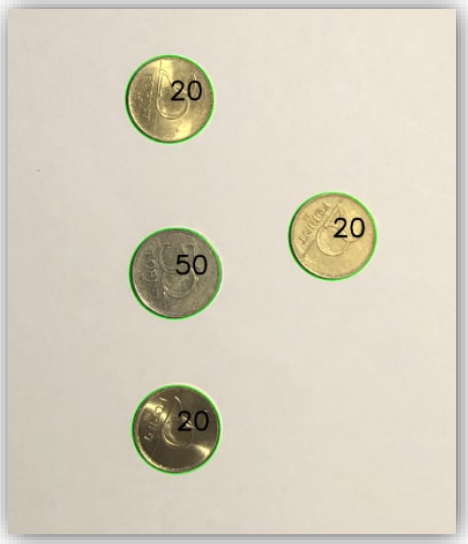

5
5


5
5
5
20
5
5

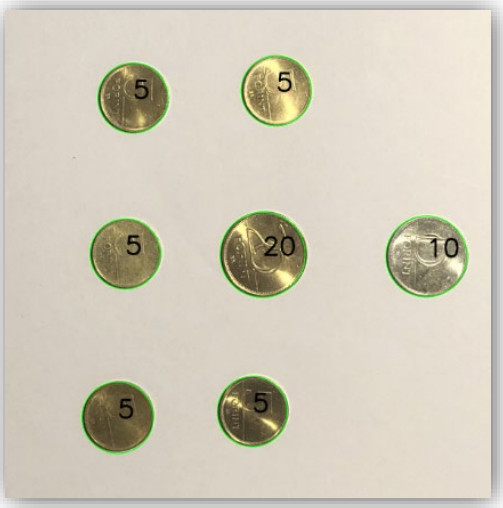

50
5
20
100
10
200


5
200
5
5
10

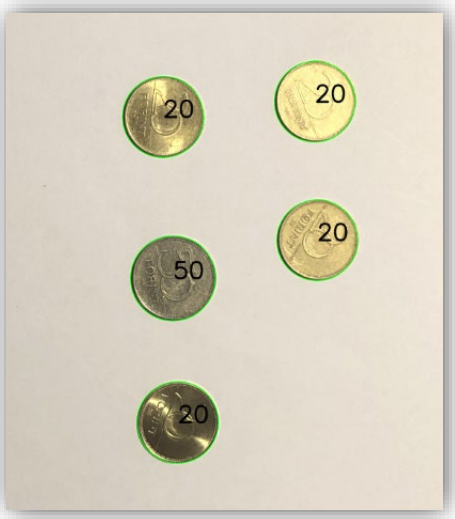

20
5
10
100
50
200


20
20
50
20

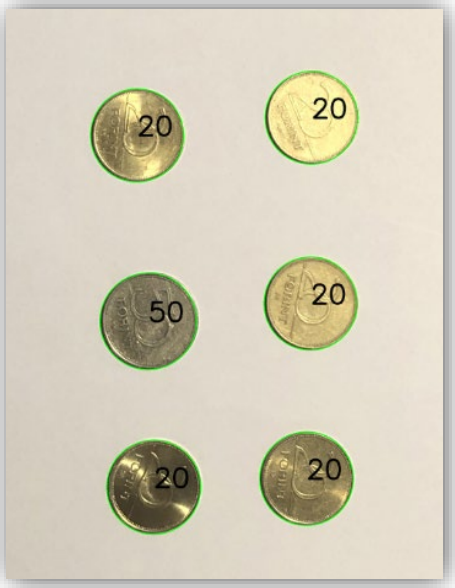

20
10
5
10
100
10
50
200
10


5
5
5
20
10
5
5

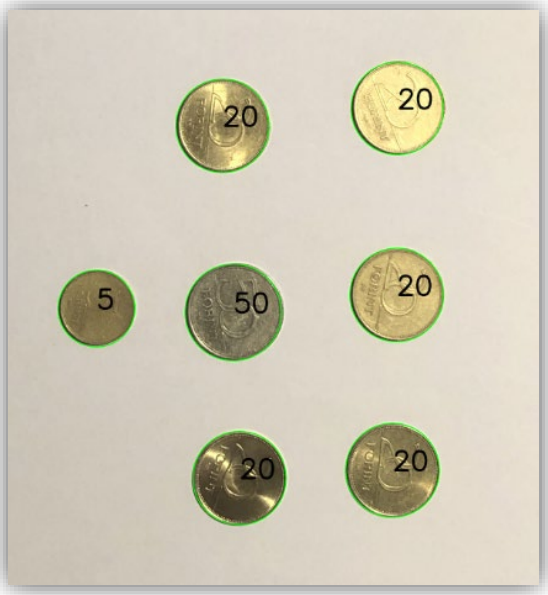

10
10
5
10
10


20
20
20
50
20

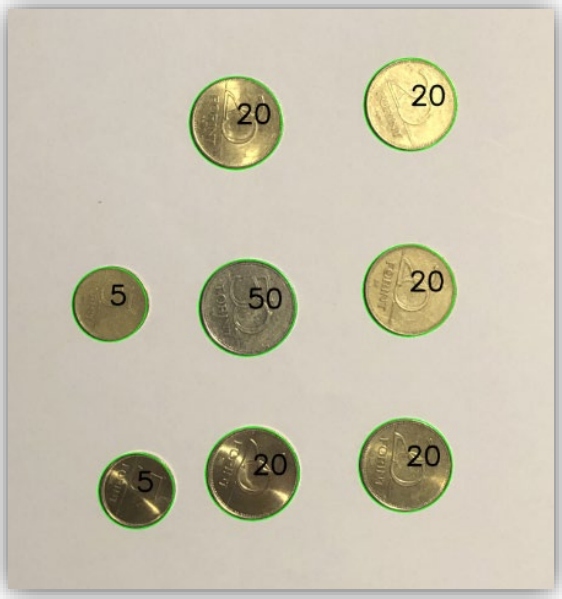

10
10
10
10


20
20
50
20
20
20

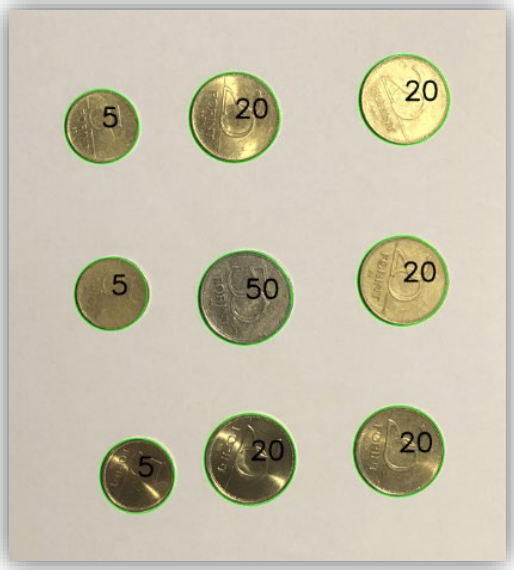

5
10
200

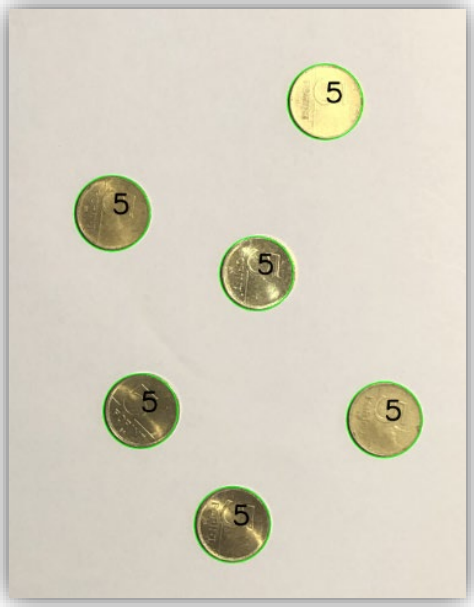

20
20
5
50
20
20
20

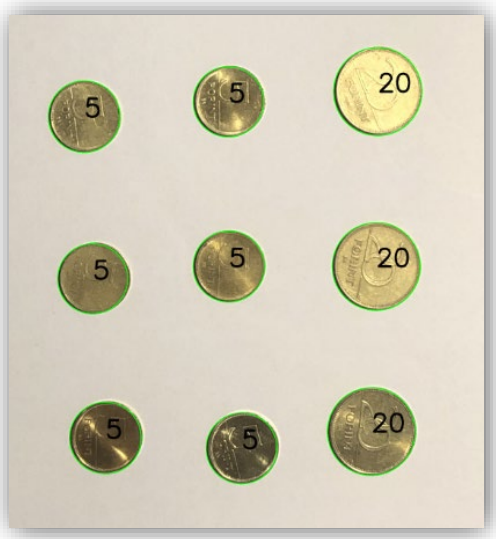

20
5
10
200

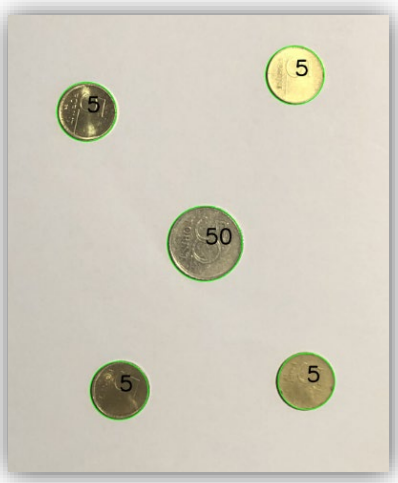

20
20
5
50
20
5
20
20

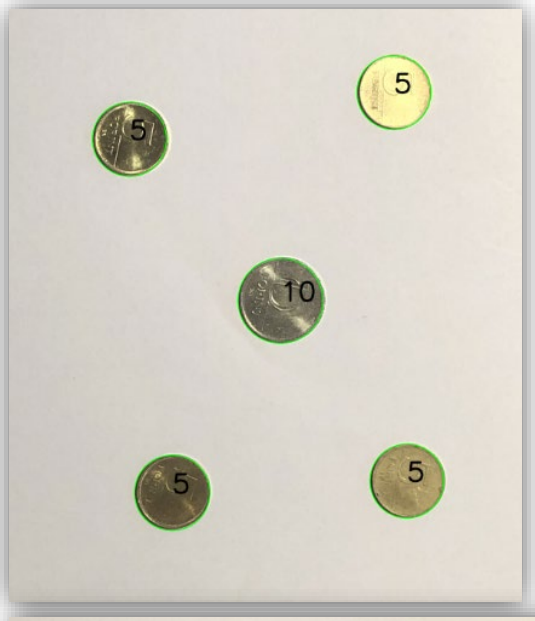

20
5
10
200
50


5
20
20
5
50
20
5
20
20


5
5
5
5
5
5

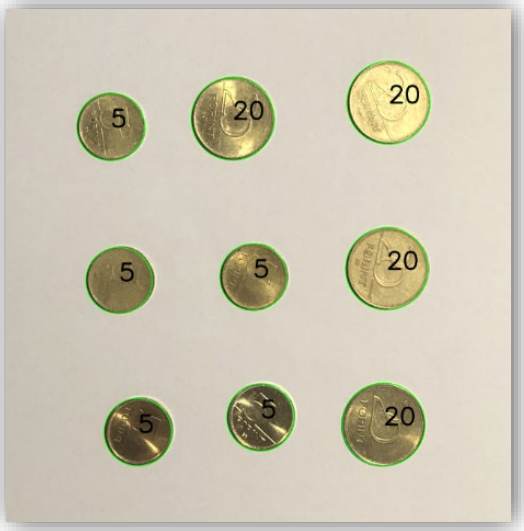

5
5
20
5
5
20
5
5
20


5
5
50
5
5


5
5
10
5
5


5
5
20
5
5
20
50
5
5
20


5
5
20
5
5
20
50
5
5
50
5
5
20

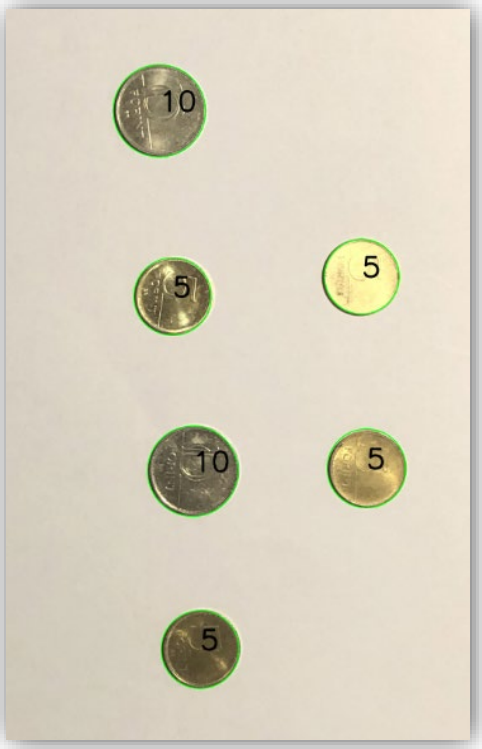

5
20
20
5
5
20
5
5
20

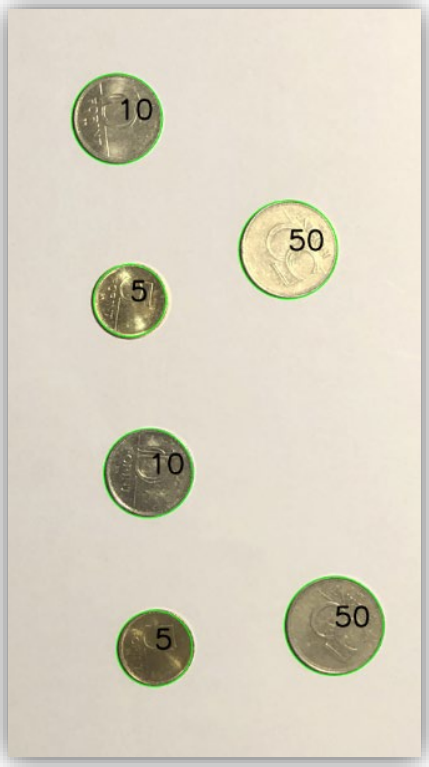

5
20
5

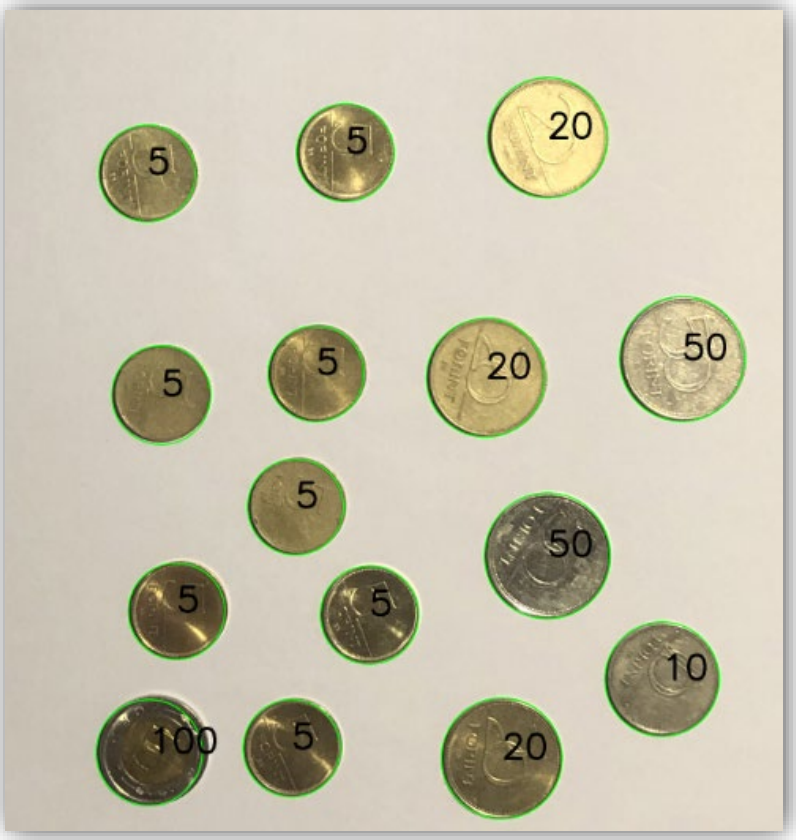

5
5
20
5
5
20
50
50
5
5
5
20


10
5
10
5


5
5
20
5
5
20
50
5
50
5
5
5
20


10
5
5
10
5
5

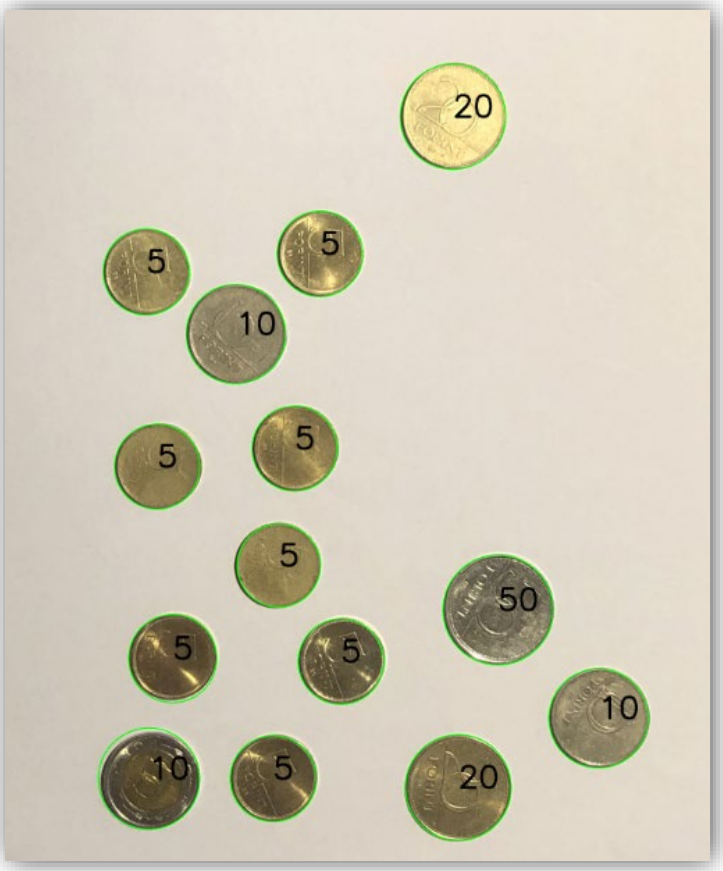

10
50
5
10
5
50

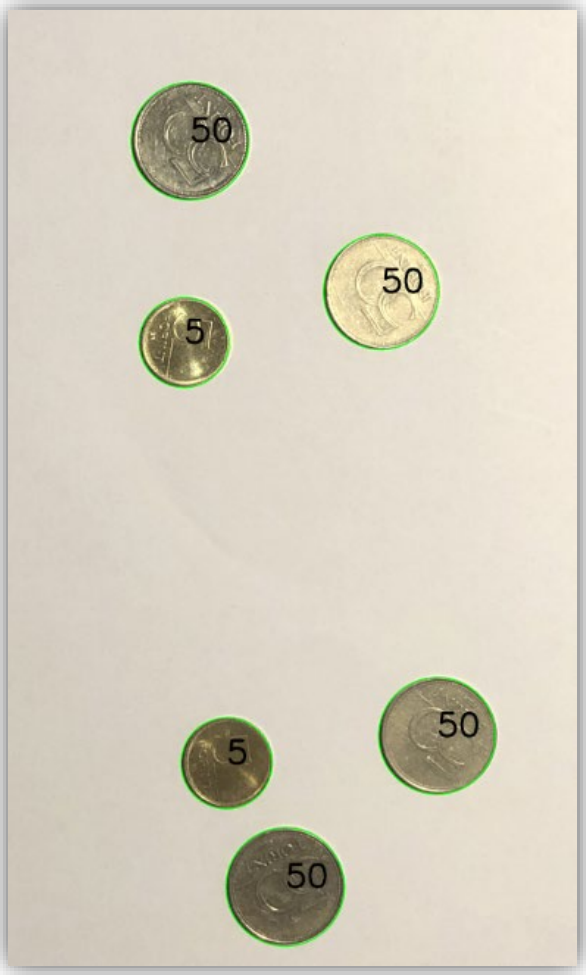

5
5
20
5
5
20
50
5
5
50
5
5
10
100
5
20

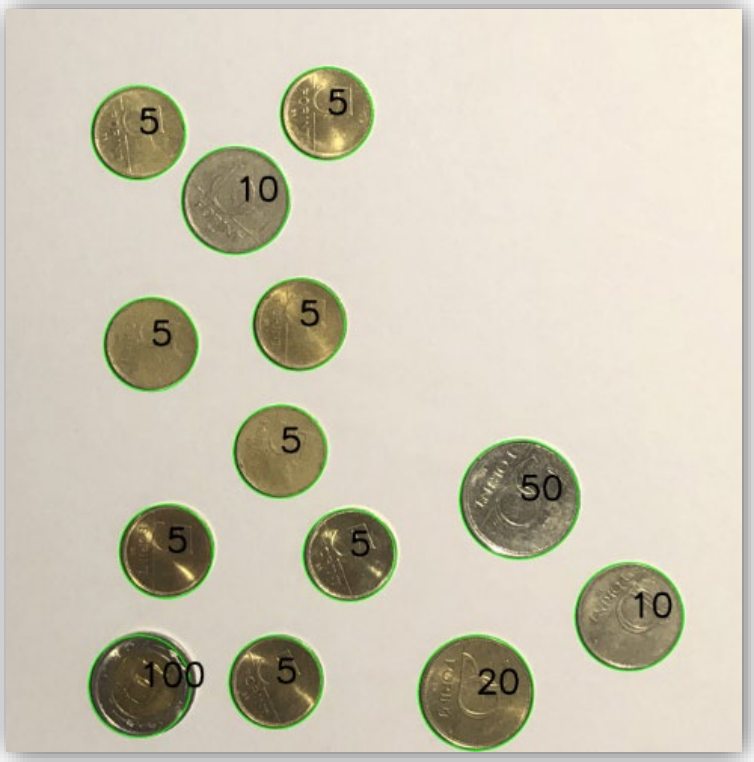

50
5
10
5
50


5
5
20
10
5
5
20
50
5
50
5
5
10
100
5
20

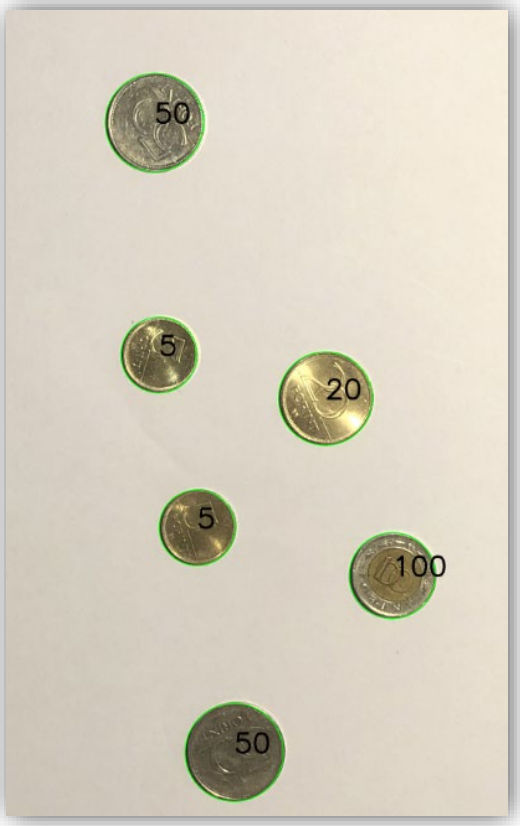

50
5
5
50


20
5
5
10
5
5
5
50
5
5
10
10
5
20

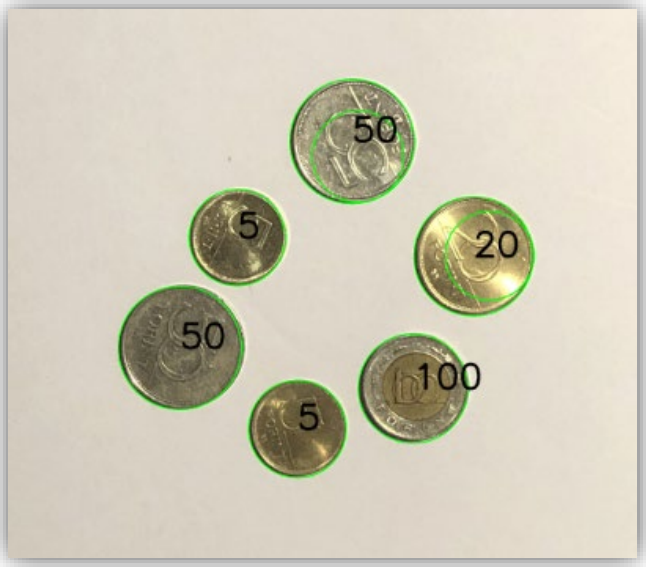

50
50
5
5
50
50

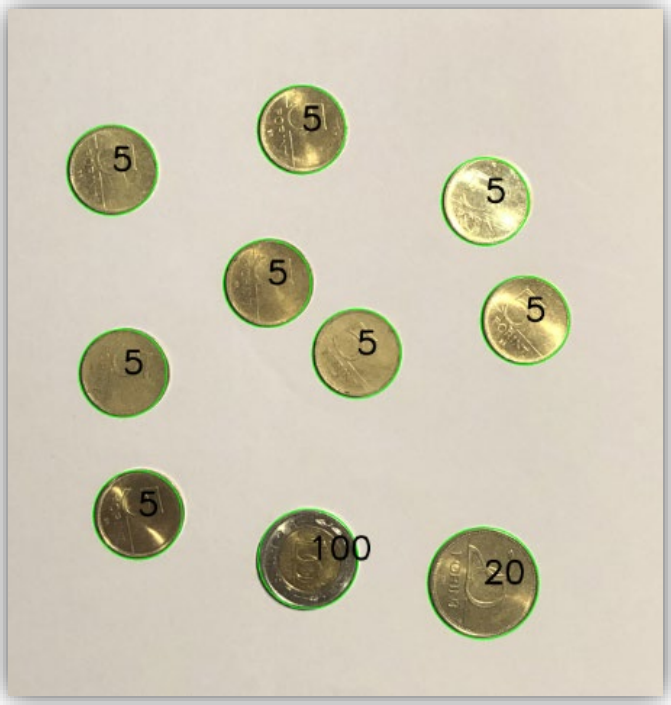

5
5
10
5
5
5
50
5
5
10
100
5
20

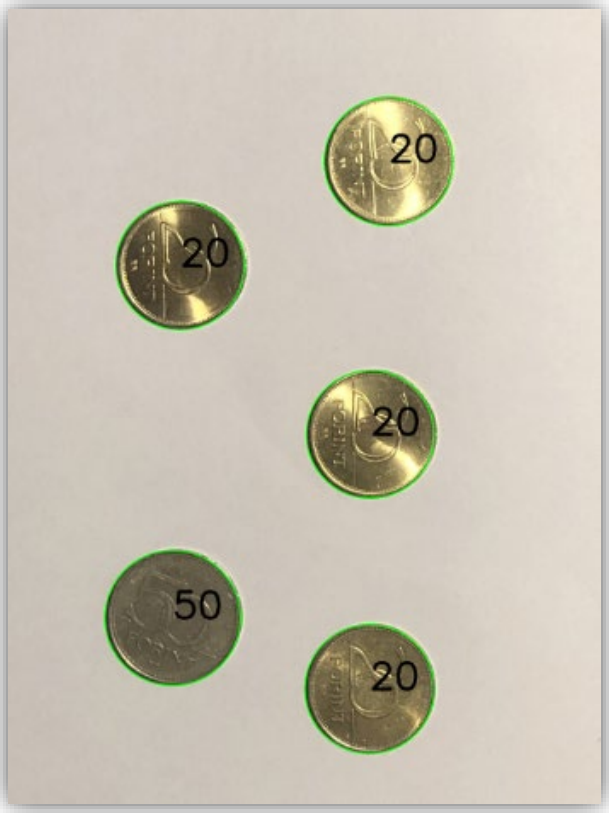

50
5
20
5
50

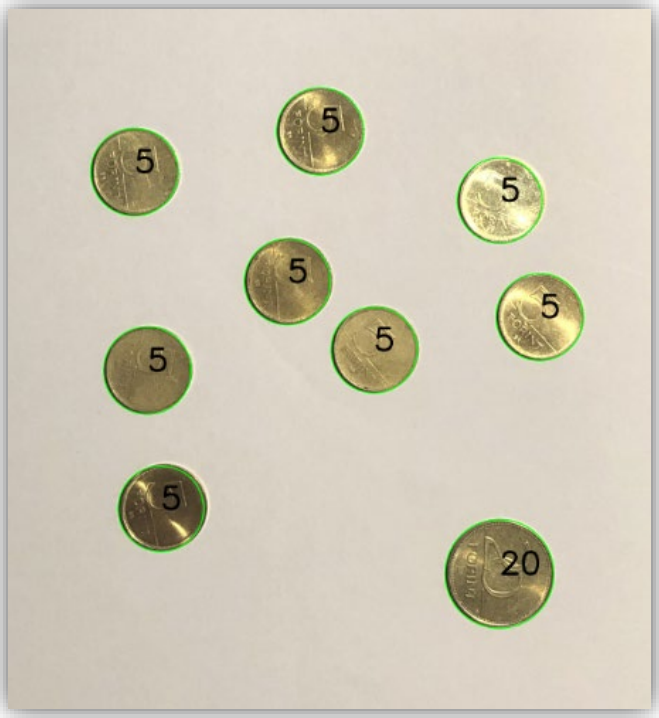

50
5
20
5
100
50

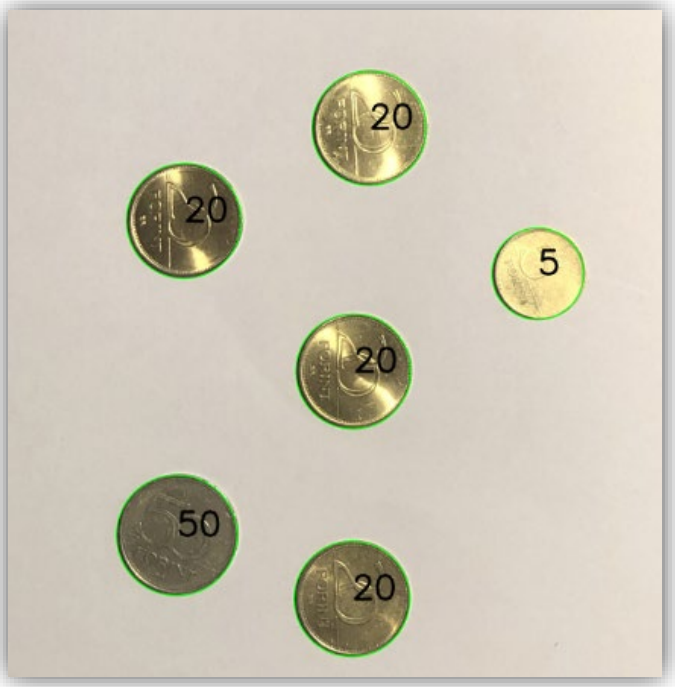

5
5
5
5
5
5
5
5
100
10
20

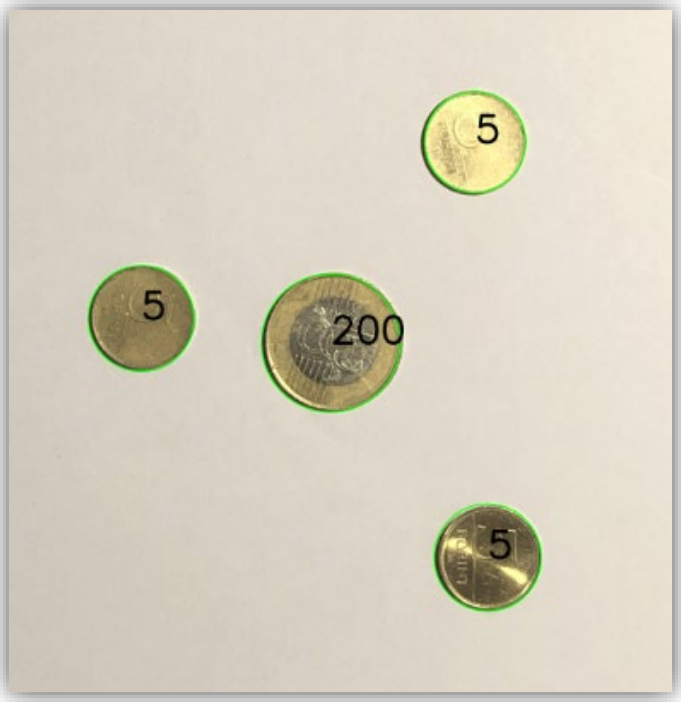

50
5
20
50
100
5

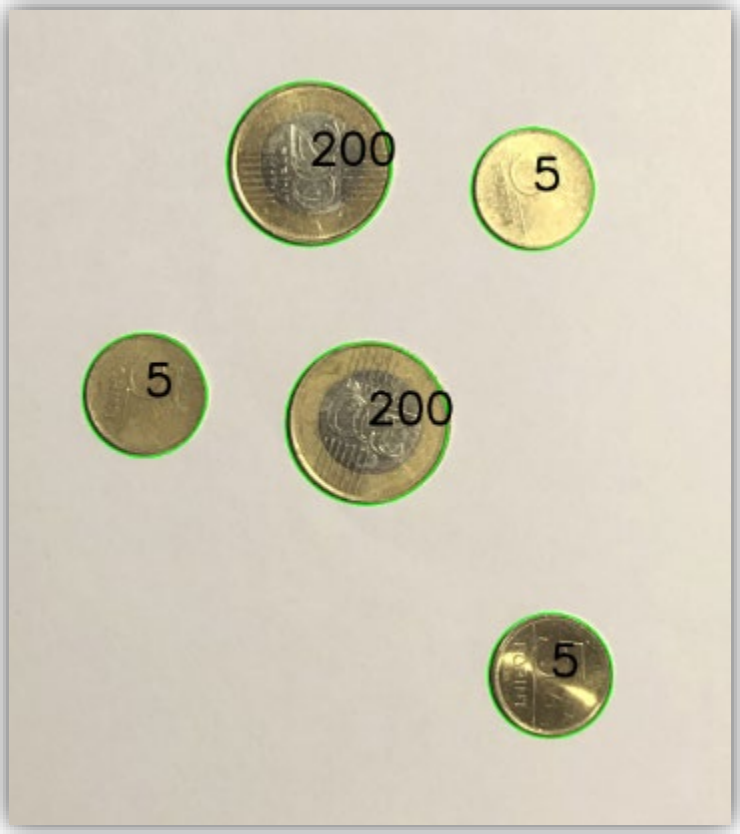

5
5
5
5
5
5
5
5
100
20

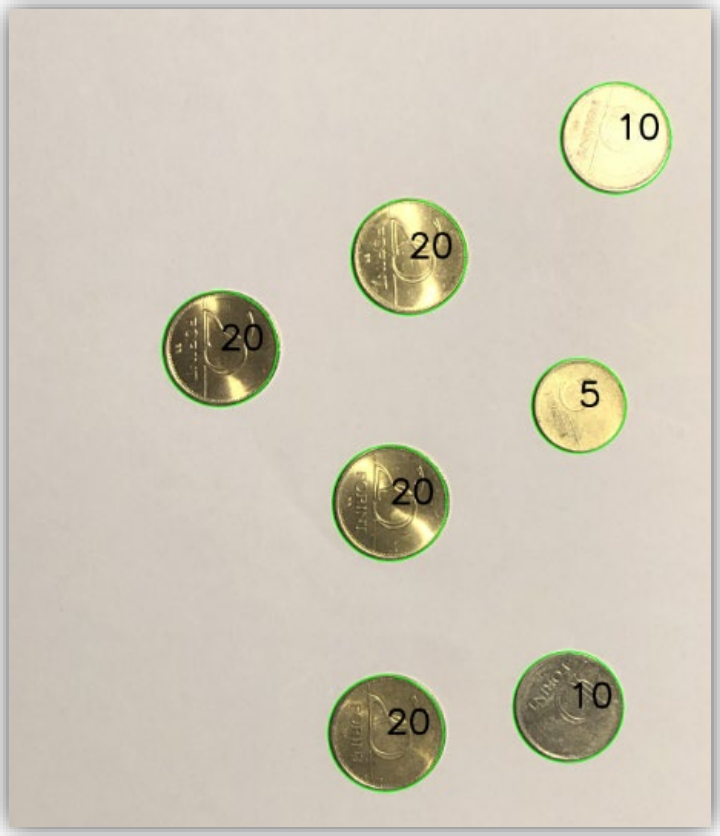

20
20
20
50
20

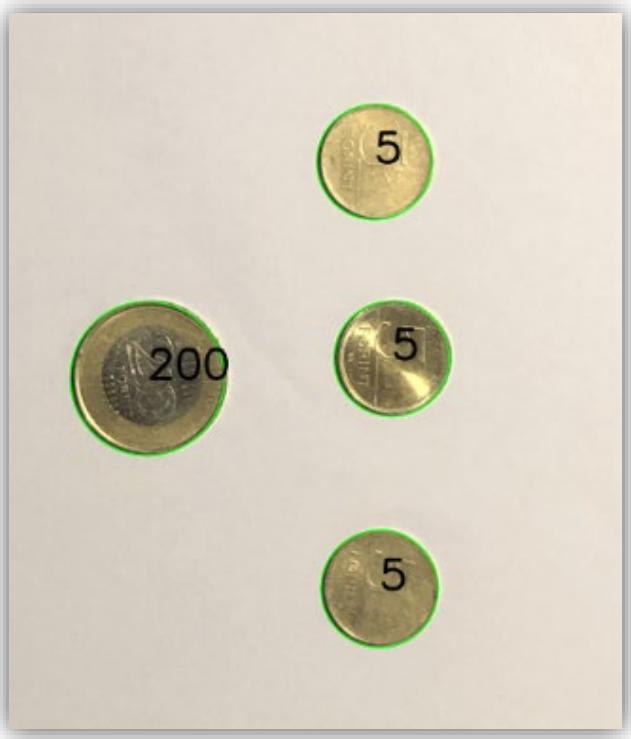

5
5
5
5
5
5
5
5
20

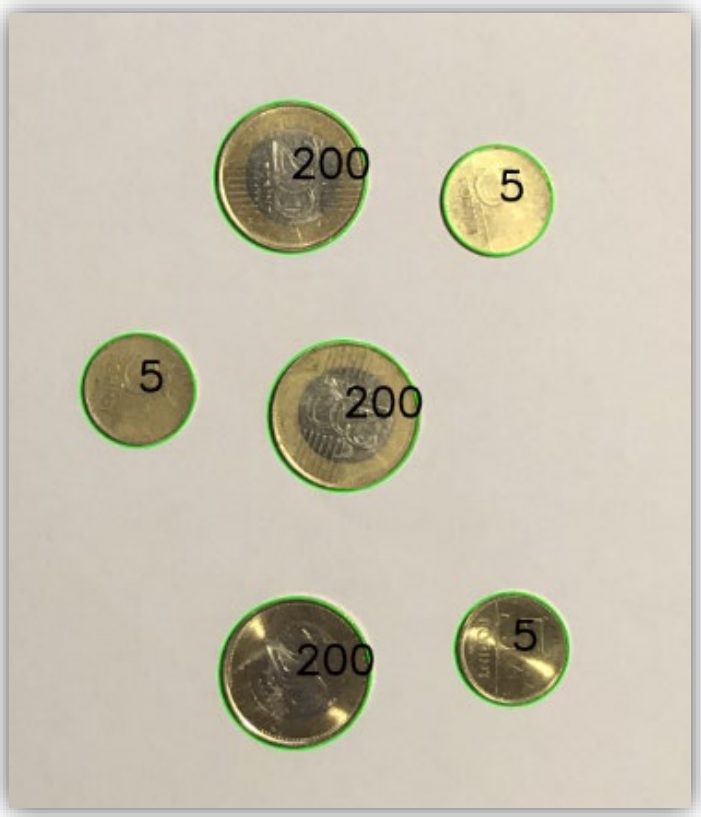

20
20
5
20
50
20


5
5
200
5


200
5
5
200
5

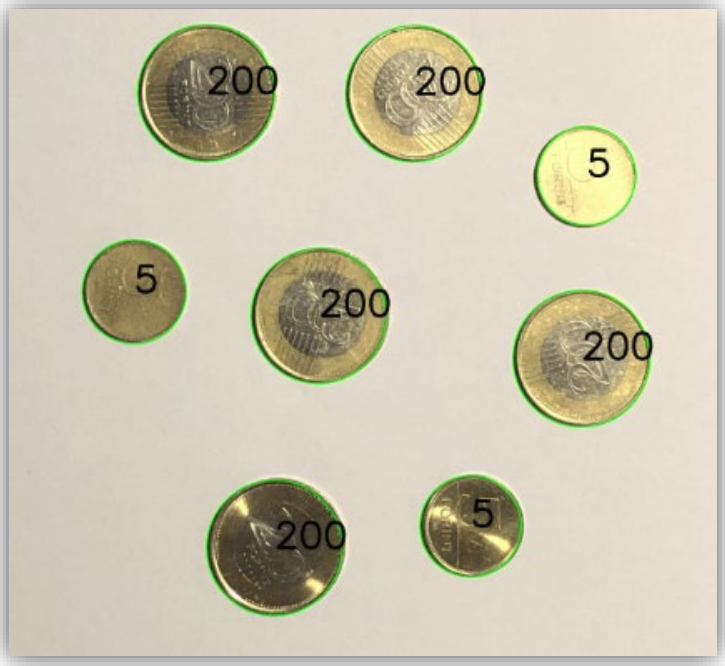

10
20
20
5
20
20
10

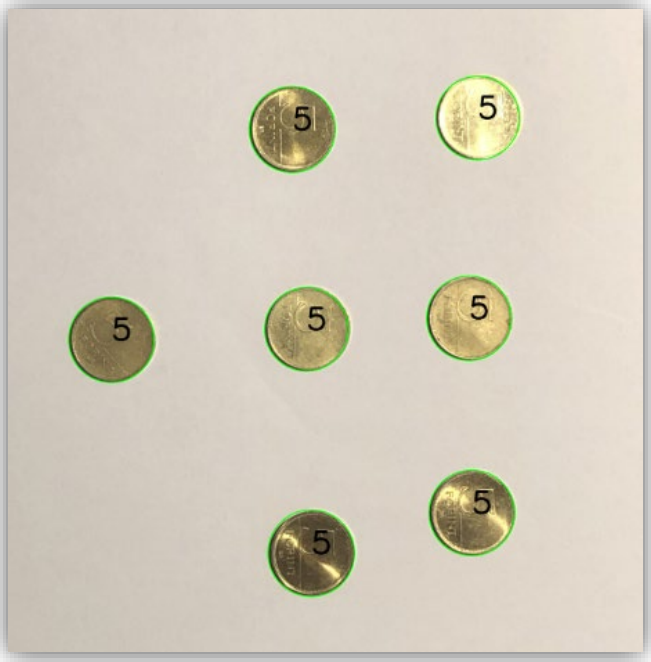

5
200
5
5

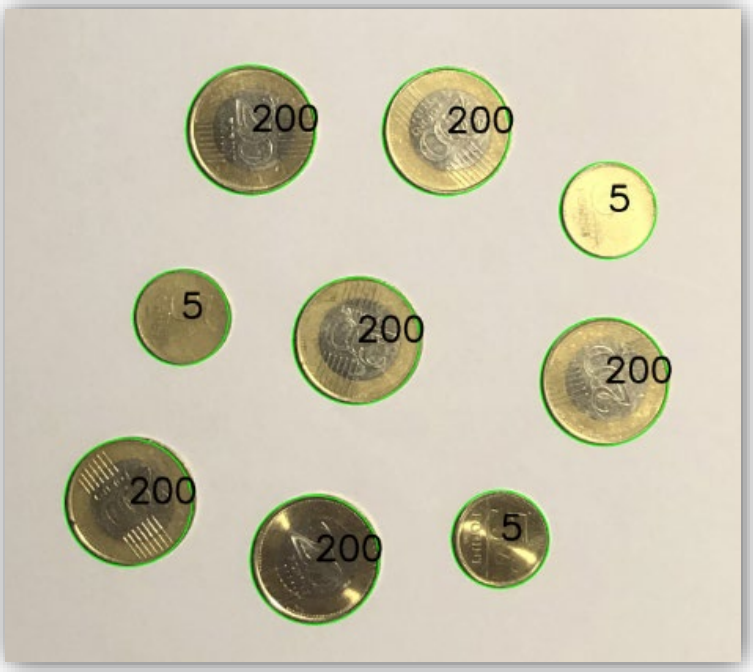

200
5
5
200
200
5


200
5
5
200
200
200
5


20
5
20
5
20
5


200
200
5
5
200
200
200
5


5
5
5
5
5
5
5


200
200
5
5
200
200
200
200
5


5
20
5
50
50
5
20

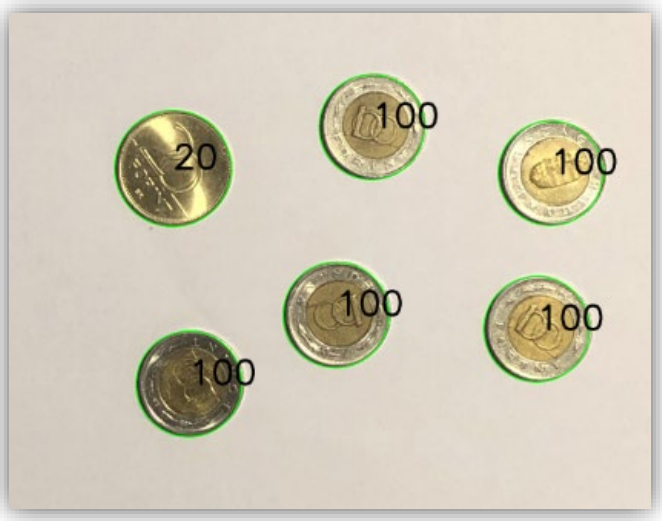

5
5
20
50
50
5
50
50

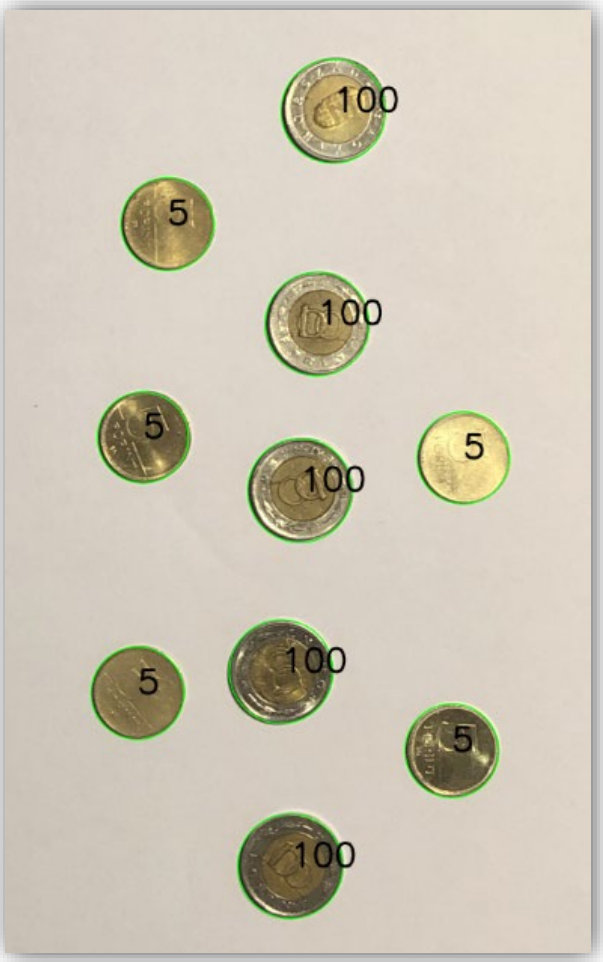

5
10


5
20
5
10

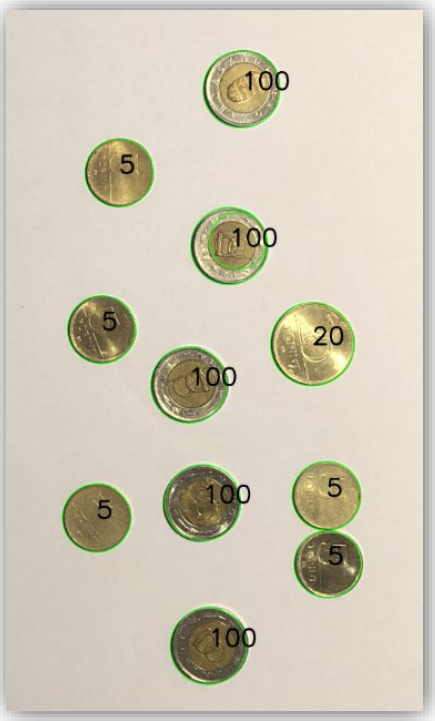

20
5
10

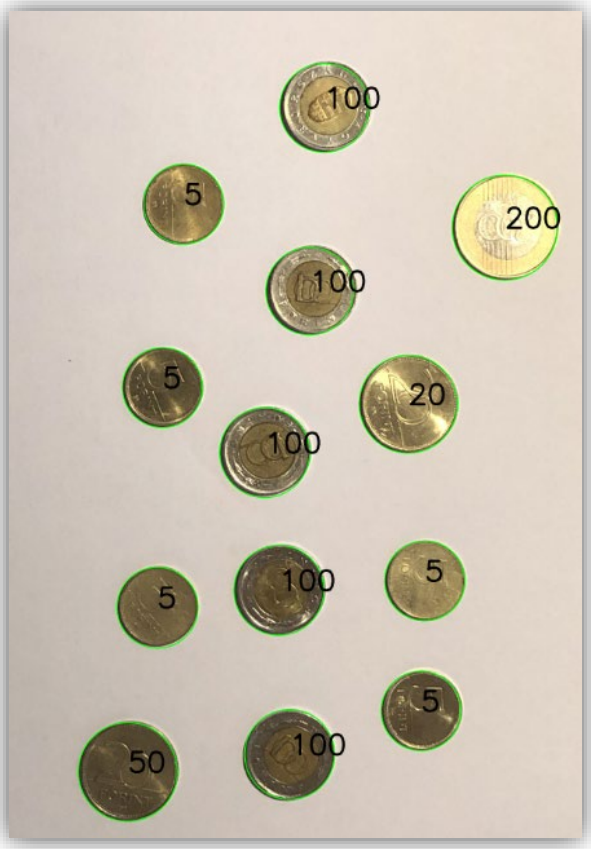

5
20
100
5
10

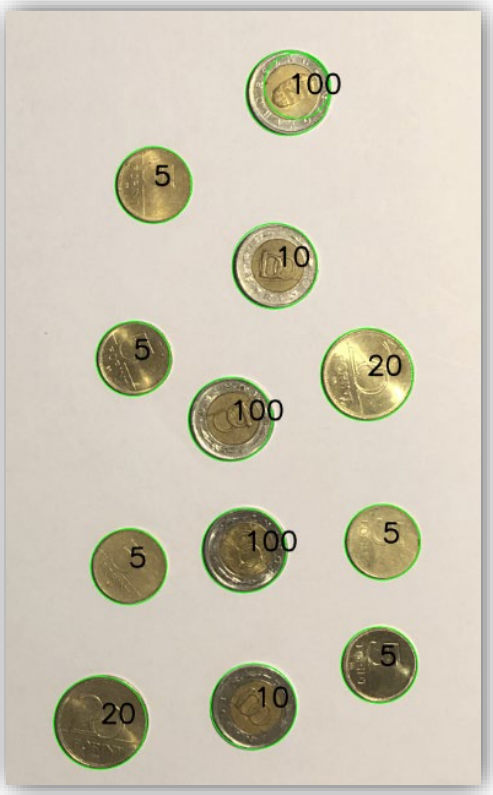

20
100
100
100
100
100


100
5
100
5
100
5
5
100
100
5


100
100
100
100
100


100
5
100
5
20
100
5
100
5
5
100


100
5
200
100
5
20
100
5
100
5
5
50
100


100
5
10
5
20
100
5
100
5
5
20
10


5
5
200
5
10

5 Felhasznált irodalom

[1] Thonny fejlesztői környezet

[2] OpenCV könyvtár és dokumentáció

[3] OpenCV mediánszűrés és kép transzformációk –szürkeárnyalatossá alakítás

[4] OpenCV Hough kör transzformáció/ Hough gradiens metódus