Érmék felismerése OpenCV használatával

Készítette: Pelz Ádám Márk

Tartalomjegyzék

1. Bevezetés 3
2. A feladat elméleti háttere 3
2.1 A kép feldolgozása 3
2.1.1 Szürkeárnyalatos kép létrehozása 4
2.1.2 Gauss simítás 4
2.1.3 Sobel operátor 5
2.1.4 Nem-maximális élek elnyomása 6
2.1.5 Hiszterézis küszöbölés 6
2.2 Dilatáció 6
2.3 Geometriai leírások 7
2.4 Jellemzők detektálása 7
2.5 Kulcspontok párosítása 8
3. Gyakorlati megvalósítás 8
4. Tesztelés 14
5. Felhasználói dokumentáció 17
Tartalomjegyzék 18

1. Bevezetés

A feladat pénzérmék felismeréséről szól. A bevitt képeken pénzérmék találhatóak, ezeken a képeken lévő érméket kell a programnak felismernie, és összeszámolni, hogy mekkora értékű pénz található a képen. A feladat akár egy való életben is felmerülő problémára adhat megoldást, ahol szeretnénk ellenőrizni, hogy egy fizetés alkalmával tényleg a megfelelő mennyiségű pénz kerül átadásra.
A program lefutásához meg kell adni egy képet, amiről a program meg tudja állapítani, hogy mennyi pénzérme van rajt. A megadott képet fel fogja dolgozni egy olyan formátumba, amin el lehet végezni a detektálást, megkeresi az érmék jellemzőit, összeveti a mintaképekkel és kiértékeli, hogy mennyi pénz található a képen. Az eredményt pedig egy időbélyeggel ellátott mappába fogja elhelyezni, ahol meg lehet tekinteni a részműveletek eredményeit is.
A probléma megoldására a Python programozási nyelvet fogom használni. A Python egy könnyen elsajátítható, magas szintű programozási nyelv. A nyelv a futtatási sebességgel szemben előnyben részesíti az olvashatóságot és a programozó munkájának megkönnyítését. A forrás- és tárgykód nincs különválasztva, tehát a kód a futási idő alatt fordul le, emiatt nem kell a programozónak a fordítással foglalkozni. A típusok kezelését, a dinamikus helyfoglalást is a háttérben végzi.
A képek feldolgozásához az OpenCV nevű könyvtárat választottam, ami egy Intel által fejlesztett ingyenes, nyílt forráskódú könyvtár. Az OpenCV rengeteg függvényt tartalmaz, amik segítségül szolgálnak a gépi látás megvalósításához.
Szükséges a numpy könyvtár használata is. Mivel a Python egy magas szinten elhelyezkedő programnyelv, ezért nem lehet vele alacsony szintű adatstruktúrákat létrehozni. Erre szolgál segítségül a numpy, ami függvényeket tartalmaz többdimenziós tömbök létrehozására, és kezelésére.

2. A feladat elméleti háttere

A feladatot a gépi látás eszközeivel lehet megoldani. Gépi látás során egy képet kiértékel a program, adatot gyűjt róla, majd ennek a hatására valamilyen vezérlési, szabályozási vagy értelmezési mechanizmus indul be. A művelet végén a képről egy leírás készül. Jelen esetben a programnak meg kell találnia a képen az érméket, és beazonosítani őket. A leírás az adott érmék értéke és a címletek összege lesz.
Az érmék felismeréséhez jellemzőket fog detektálni a program, ilyenek például a sarkok, élek. Hogy ezt végre tudja hajtani, ahhoz a képet a bevitel után át kell alakítani egy olyan formátumra, amiről a program képes lesz felismerni az érméket. Az érmék ki lesznek vágva egyenként a képből, így a háttérben lévő éleket és sarkokat nem fogja detektálni.

2.1 A kép feldolgozása

A jellemzők detektálásához a képet át kell alakítanunk egy bináris képpé, amin csak a szükséges információk szerepelnek. Egy ilyen képet éldetektálással tudunk elérni. Élről beszélünk, ha szomszédos képpontok között hirtelen nagy intenzitás-különbség lép fel. Az éldetektáláshoz Canny algoritmust fogok használni. A Canny egy több lépésből álló folyamat.

A Canny főbb lépései:

- Szürkeárnyalatos kép létrehozása
- Gauss simítás
- Sobel operátor
- Nem-maximális élek elnyomása
- Hiszterézis küszöbölés

A továbbiakban ezeket a lépéseket fogom taglalni.

2.1.1 Szürkeárnyalatos kép létrehozása

A pixelek színét színcsatornák értékei adják, RGB színmódú képeken három ilyen csatorna van: piros, zöld, kék. A csatornák egy szín értékét veszik fel, amit egy számmal írunk le, ennek tartománya függ attól, hogy hány bites színmélysége van a képnek, több bites színmélységgel többféle színárnyalatot lehet ábrázolni.

A szürkeárnyalatos képek alatt olyan képeket értünk, ahol minden pixelnek csak egy értéke van, ami a fény erősségét fejezi ki, vagyis a képpontoknak csak az intenzitását nézzük. Az ilyen képek csak a szürke árnyalataiból állnak. A szürkeárnyalatos képek létrehozásának eljárása egy pont-operáció. A pont-operáció során egy adott műveletet hajtunk végre minden egyes pixelen. A lokális-, vagy globális-operációkkal ellentétben nincs rá hatással a képpont környezete.

Szürkeárnyalatos képet úgy hozunk létre, hogy az eredeti képpont alapszínei alapján kiválasztunk egy értéket, ami a szürke képpont intenzitása lesz. Ennek az értéknek a kiválasztása történhet egy egyszerű átlagszámítással is, de jelen esetben az alábbi függvényt használja az OpenCV.

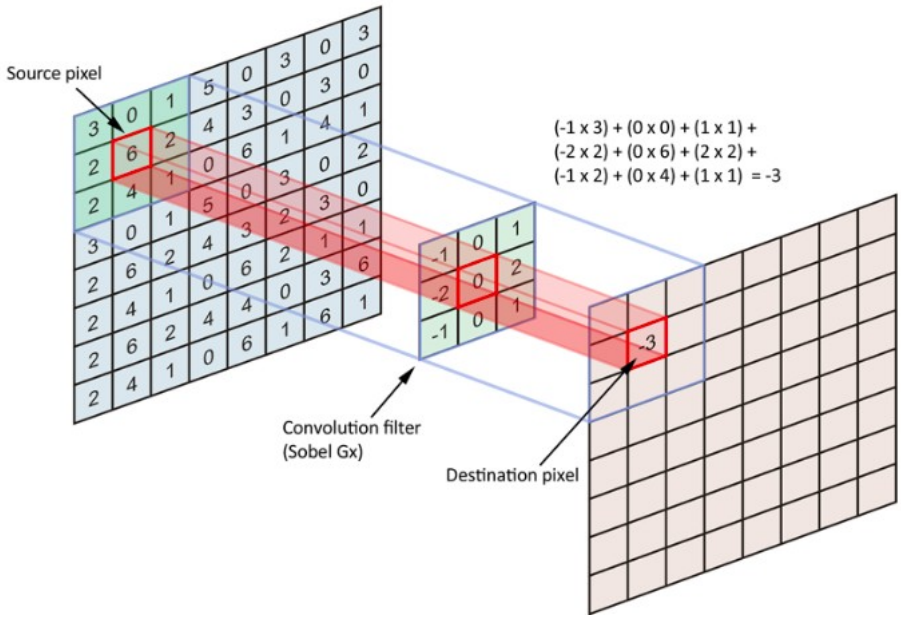
$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B$$

Szürkeárnyalatos képhez használt arányok

2.1.2 Gauss simítás

A szürkeárnyalatos kép elkészültével a képfeldolgozás következő lépése a Gauss simítás. Erre a zajok szűrése miatt lesz szükség, ezzel segítve az éldetektálást. A zaj alatt olyan jeleket értünk, amik hozzáadódnak a hasznos információhoz, és ezáltal nehezítik annak értelmezését. A képen keletkezhet zaj fényképezés közben, vagy digitalizálás(kvantálás) közben is. Zajszűrés folyamán megpróbáljuk a képen előforduló zajokat kiküszöbölni, viszont ez a képminőség rovására történik.

A Gauss-féle simítás egy lokális operáció, tehát a képpont környezetétől fog függeni. A művelet során konvolúciót fogunk végrehajtani egy csoportnyi pixelen. A konvolúció során a képpontok értékei egyenként szorozva lesznek egy mátrix értékeivel, majd összeadásra kerülnek. Ez a mátrix a konvolúciós kernel, méretétől függ, hogy a képpont mekkora környezetén hajtjuk végre a műveletet. Egy konvolúciós kernel értékei függenek attól, hogy a konvolúciót milyen indokkal hajtjuk végre, ugyanis több célra is felhasználható.



Konvolúció

Néhány példa Gauss kernelekre:

1/16

1	2	1
2	4	2
1	2	1

1/273

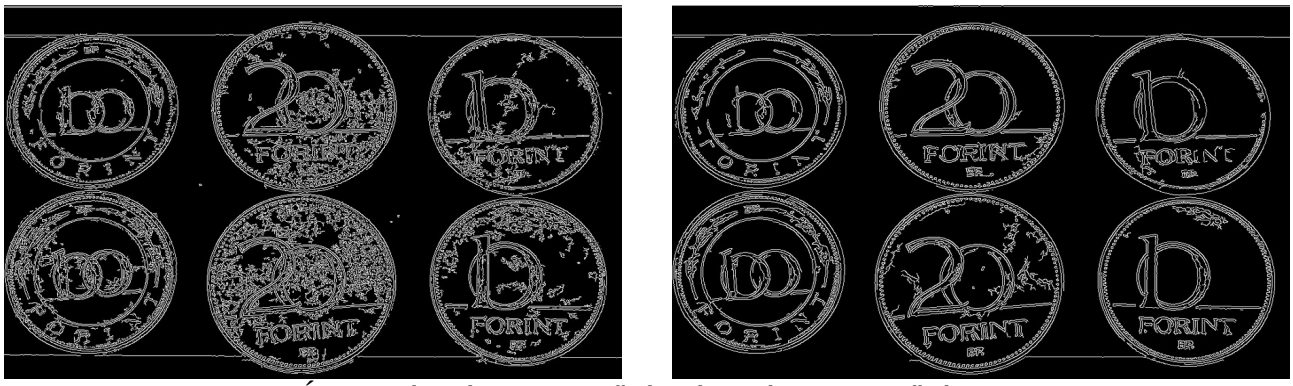
1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

1/1003

0	0	1	2	1	0	0
0	3	13	22	13	3	0
1	13	59	97	59	13	1
2	22	97	159	97	22	2
1	13	59	97	59	13	1
0	3	13	22	13	3	0
0	0	1	2	1	0	0

A Gauss kernel 3x3-as, 5x5-ös, és 7x7-es méretekben

Az elmosás eredménye legjobban egy éldetektált képen szemléltethető, ugyanis az alap képen emberi szemmel kevésbé feltűnő a különbség, viszont a programnak elengedhetetlen, hogy a minimális zajok is ki legyenek szűrve.



Éldetektált kép Gauss szűrés nélkül és Gauss szűréssel

Az ábrán jól látható, hogy Gauss szűrés nélkül a képen jóval nagyobb zaj van, a középső két érme így a program által észlelhetetlenné válhat.

2.1.3 Sobel operátor

Az éldetektálás a Gauss simításhoz hasonlóan konvolúcióval történik. Már volt róla szó, hogy éleknél a pixelek intenzitása hirtelen megváltozik. Ezt a változást legjobban grádiensek kiszámításával lehet kimutatni. A kiszámított grádiensnek lesz egy nagysága, és egy iránya.
A különbségek kiemelésére a Sobel operátort használja. A Sobel operátor két mátrixból áll, egy a vízszintes, egy a függőleges élek detektálására.

X – Direction Kernel

-1	0	1
-2	0	2
-1	0	1

Y – Direction Kernel

-1	-2	-1
0	0	0
1	2	1

Vízszintes és függőleges Sobel kernelek

A különbségek nagyságát és irányát az alábbi két függvénnyel lehet kiszámítani:

$$Edge\_Gradient\ (G) = \sqrt{G_x^2 + G_y^2}$$

$$Angle\ (\theta) = \tan^{-1}\left(\frac{G_y}{G_x}\right)$$

Ahol:

- ***Edge_Gradient (G):*** Az intenzitás nagysága
- ***Angle:*** Az intenzitás iránya
- G_x***:*** A vízszintes éldetektálás eredménye
- G_y***:*** A függőleges éldetektálás eredménye

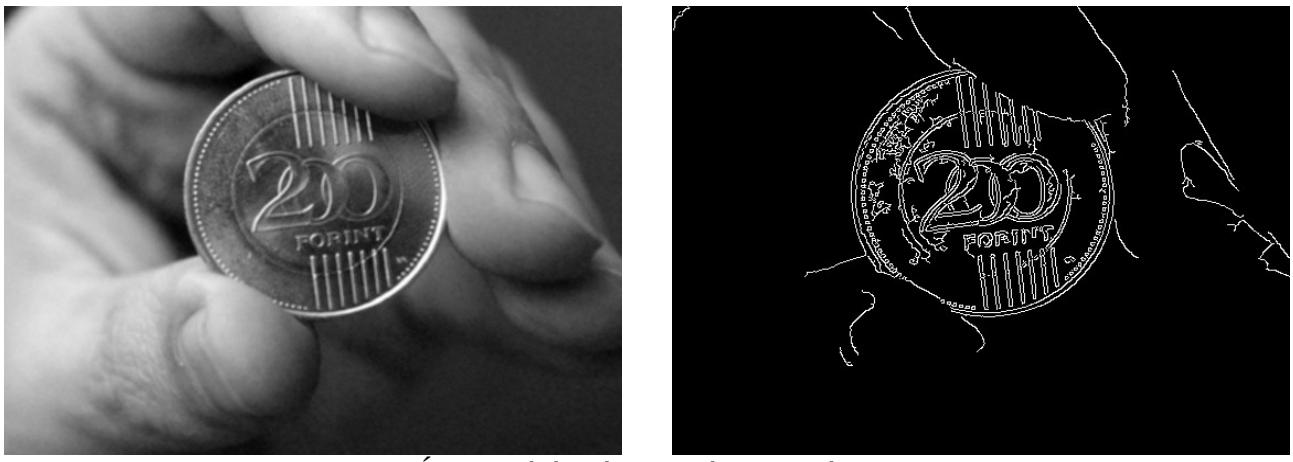
2.1.4 Nem-maximális élek elnyomása

Ezután a nem-maximális élek elnyomásra kerülnek, azaz egy lokális környezet alapján kinullázásra kerül minden érték, ami az adott környezeten belül nem a legnagyobb értéket veszi fel.

2.1.5 Hiszterézis küszöbölés

Utolsó lépésként ki kell szűrni további nem megfelelő éleket. Ez két előre megadott küszöbérték alapján történik, ha a felső küszöbérték felett van a pixel intenzitása, akkor megfelelő, ha az alsó küszöbérték alatt, akkor kinullázzuk, ha pedig a kettő között, akkor az fogja eldönteni, hogy van e él szomszédja a pontnak.

A Canny algoritmus futtatása után a következő képet kapjuk:

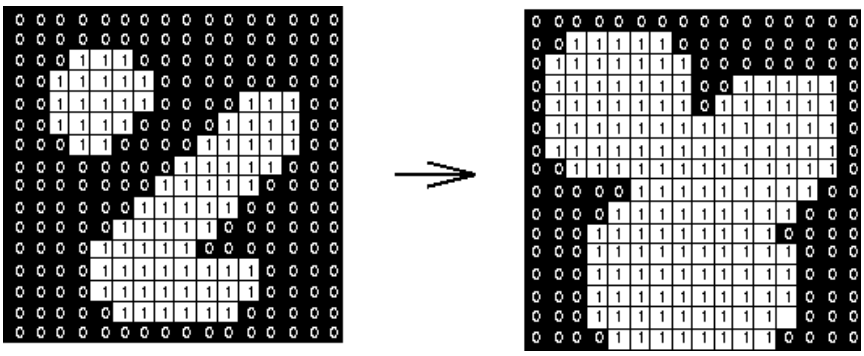


Éldetektálás végrehajtása egy képen

2.2 Dilatáció

A megfelelő felismeréshez segítséget nyújt egy morfológiai művelet, a dilatáció. A morfológia alaktant jelent. A morfológiai operációk lokális operációk, ahol meg kell adni a strukturáló elem méretét, és origóját. Az origó alapvető értéke a mátrix középső eleme.

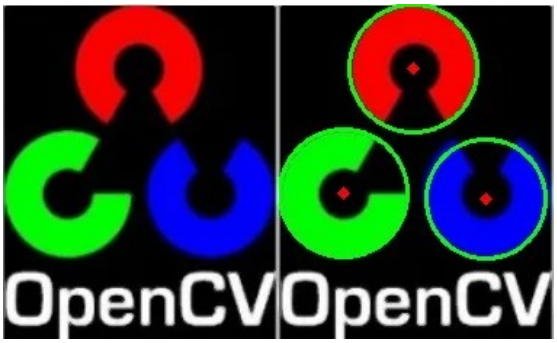
A dilatáció során a megfigyelésre kerül a képpont környezete, és az ott található szomszédos pontok. Ha van legalább egy pont, aminek ‘1’ az értéke, akkor a kiválasztott képpont fehér lesz, ha nincs, akkor fekete. A művelet eredménye a képen található fehér képpontok számának növekedése lesz. Szükség lehet rá fehérzaj szűrése után, ugyanis néhány körvonalban szakadás léphet fel, amit dilatációval tudunk korrigálni.



Dilatáció művelet egy bináris rajzon

2.3 Geometriai leírások

A kép feldolgozása után meg kell találni rajt az érméket, hogy elkezdődhessen az összehasonlítás. Ez fogja megadni, hogy tulajdonképpen mennyi érme van a képen, és segít őket elválasztani a háttértől, így nem fognak az érméken kívül eső jellemzők belekeveredni az összehasonlításba.
A körök detektálás Hough transzformációval történik. A Hough transzformáció egyenes vonalakat, vonalszakaszokat és köröket képes detektálni egy képen.

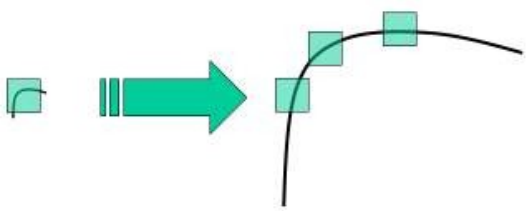


Hough körök detektálása az OpenCV logóján

A művelet végén lesz egy teljes lista a talált körök középpontjának koordinátáiról, illetve annak sugaráról. Miután ezeket az adatokat megkaptuk, már csak létre kell hozni egy bináris képet, ahol a teljes kép fekete, csak a kör területe van fehérrel kitöltve. Ennek segítségével megtörténhet a kép kivágása.

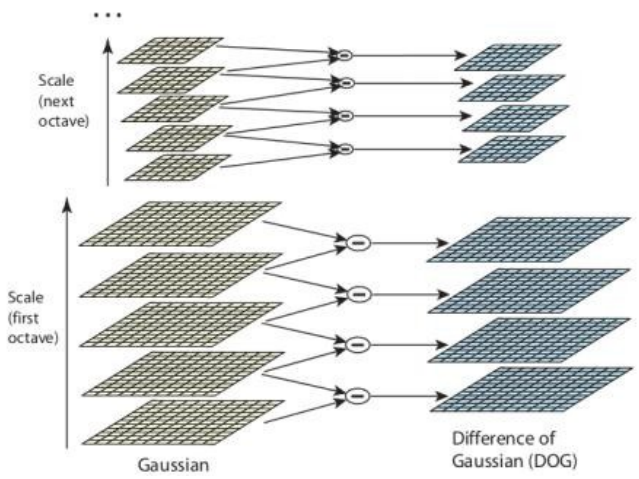
2.4 Jellemzők detektálása

A kivágás után elkezdőthet az érmék jellemzőinek detektálása. A detektáláshoz SIFT algoritmust használok. A SIFT egy skálainvariáns algoritmus, azaz nincs hatással a detektálásra a kép mérete. Az alábbi képen jól látható, hogy egy kisebb ablakkal ki lehet szűrni kisebb kulcspontokat, viszont egy nagyobbhoz már nem lenne elég.



Különböző méretű sarkok egy adott méretű ablakkal

A különböző méretű sarkok kiszűréséhez különböző méretű ablakokat kell használnunk. A művelet közben különböző σ skálázási paraméterrel fogja végignézni a környezeteket, a σ paraméter változtatásával különböző értékeket kap minden vizsgált méretre, a kisebb ablak jobban fog illeszkedni a kisebb sarokra, a nagyobb ablak pedig a nagyobb sarkokra. Az értékek lokális maximuma lesz az észlelés eredménye. A talált kulcspontokat küszöbölni kell. Ha az érték kisebb mint a megadott küszöbérték, akkor elutasításra kerül. A kulcspontokhoz hozzárendel egy tájolást, ami 0 és 360 fok közötti értéket vehet fel, ez lesz a pont leírója.



SIFT algoritmus

2.5 Kulcspontok párosítása

Az előző lépésben megtalált kulcspontokat, és leíróvektoraikat össze kell hasonlítani, hogy meg lehessen állapítani az érme típusát. Az összehasonlításhoz a FLANN algoritmust fogom használni. A FLANN algoritmus távolságok arányának vizsgálatával szűri ki a hamis egyezéseket. Egy adott kulcspont két legközelebbi egyezése közötti távolságarányt számítja ki, és ez jó egyezés, ha az érték egy küszöbérték alatt van.

3. Gyakorlati megvalósítás

A program kódja egy ***coindetect.py*** nevű fájlban helyezkedik el. A program futtatásához az alábbi modulokat használom fel:

```
import os
import argparse as arg
import datetime as dt
import numpy as np
import cv2 as cv
```

Importált modulok, 3-7 kódsor

- **os:** Eszközöket nyújt az operációs rendszer kezeléséhez.
- **argparse:** Parancssori paraméterek beadásához kell.
- **datetime:** Dátumok kezeléséhez való osztályt implementál.
- **numpy:** Mátrixok létrehozása miatt lesz rá szükség.
- **cv2:** Segítségével történik a képfeldolgozás.

A program indításához argumentumokat kell megadni a parancssorból. Ezek az argumentumok a fájl elérési útvonala, a körök várható sugara, és a Gauss szűréshez használt kernel mérete.

```
parser = arg.ArgumentParser()
parser.add_argument("source",help="Path of the source file.")
parser.add_argument("minrad",help="Minimum radius of the Hough Circles.", type=int)
parser.add_argument("gauss",help="Kernel size of Gaussian blur.", type=int)
args=parser.parse_args()

target=args.source
radius=args.minrad
gauss=args.gauss
```

Először létre kell hozni egy ***argparse*** osztályt, majd egyenként hozzá kell adni az argumentumokat. A ***help*** paraméter akkor fontos, ha valaki információt szeretne lekérni a paraméterekhez. Miután ez megvan, parszolni kell őket. A beadott inputok külön változókban lesznek elhelyezve a további munkához.

```
timestamp = dt.datetime.now()
forints=('5','10','20','50','100','200')
images=[]
matching=[]
```

Ezután meg kell adni további változókat. A ***timestamp***-re, azaz időbélyegre az eredmények könyvtára miatt lesz szükség, minden kimenet egy időbélyeggel ellátott mappába kerül. Az érmék várható típusai a ***forints*** nevű tuple-be vannak beírva. Az alsó két lista pedig a minták elérési útvonalainak, illetve az összehasonlítások eredményeinek tárolására lesznek létrehozva.

```
output='results/' + str(timestamp.year) + str(timestamp.month) + str(timestamp.day) + \
»   str(timestamp.hour) + str(timestamp.minute) + str(timestamp.second) + '/'
cutpath = output + 'cuts/'
samplepath = output + 'samples/'
matchpath = output + 'matches/'
os.makedirs(output)
os.makedirs(cutpath)
os.makedirs(samplepath)
os.makedirs(matchpath)

log=open(output+'log','a')
log.write("--- Coin detection on {} ---\n \
»   »     Date: {}\n \
»   »     Minimum radius: {} \n \
»   »     Gauss kernel size: {}\n\n \
»   »     ".format(name,timestamp,radius,gauss))
```

Az ***output*** változóba fog kerülni a kimeneti mappa elérési útvonala, ami a ***results*** mappából, és az időbélyegből fog állni. Ebben a könyvtárban lesz további három mappa, amik a kivágott érméknek, a mintaérméknek és az összehasonlítás eredményeinek tárolására lesznek létrehozva. Az inicializálás után az ***os*** osztály ***makedirs*** függvénye létrehozza ezeket a könyvtárakat. Végül létrehozza a log fájlt, és beleírja az első információkat.

```
for i in range(0,len(forints)):
    j=1
    images.append([])
    while os.path.isfile('db/'+forints[i]+'/'+str(j)):
        images[i].append('db/'+forints[i]+'/'+str(j))
        j=j+1
```

A mintaérmék használatához fel kell deríteni a ***db*** könyvtárban lévő képeket. Minden érmetípusra, amik a tuple-be lettek írva létezik egy könyvtár, amiben találhatóak a képek, számokkal ellátva névként, ezeket fogja a program az iterálás során beolvasni.

```
target = cv.imread(name)
blurred = cv.GaussianBlur(target, (gauss,gauss), 0)
edge_targ = cv.Canny(blurred, 30, 150, 2)
edge_targ = cv.dilate(edge_targ, (11,11), iterations = 2)

circles = circles(edge_targ,radius)
print(len(circles)," coins detected.")

cv.imwrite(output+"CannyEdge.jpg", edge_targ)
```

A kép feldolgozása OpenCV függvények segítségével. Először az ***imread*** függvény az elérési útvonal alapján beolvassa a képet egy háromdimenziós tömbbe. A következőekben egyenként részletezem a további függvényeket, és a paramétereket. A függvényeknek vannak további paramétereik, amik a programkódban nem kerülnek megadásra, ezekre nem fogok kitérni.

A ***GaussianBlur*** függvénnyel Gauss simítást végez a képen.
GaussianBlur(src, kszie, sigmaX)

- **src:** Forrásfájl elérési útvonala.
- **ksize:** A kernel mérete. A példában a (gauss,gauss) tuple adja meg, és a program futtatásakor bemeneti paraméterként kerül átadásra. A két értéknek pozitívnak és páratlannak kell lennie.
- **SigmaX:** Gauss kernel szórása X irányban. A függvénynek van SigmaY paramétere is, de ez alapvetően 0, ha nincs neki külön érték megadva.

Az éldetektálást a ***Canny*** függvény végzi.
Canny(image, threshold1, threshold2, apertureSize)

- **image:** Bemeneti kép.
- **threshold1:** Küszöbölés alsó határa.
- **threshold2:** Küszöbölés felső határa.
- **apertureSize:** Sobel operátor mérete.

A ***dilate*** függvény azért szükséges, mert az éldetektálás után a körvonalakat meg kell erősíteni.
dilate(src, kernel, iterations)

- **src:** Bemeneti kép.
- **kernel:** A kernel mérete.
- **iterations:** Meghatározza hányszor hajtódjon végre a dilatáció a képen. Jelen esetben kétszer.

Ezek után a ***circles*** függvény megkeresi a köröket a képen, az ***imwrite*** pedig a megadott elérési útvonalra létrehoz egy képet, amin a detektált kulcspontok szerepelnek.
A ***circles*** függvény egy saját függvény, ami így néz ki:

```
def circles(img,r):
    minrad=r
    maxrad=minrad*2
    distance=int(minrad*2.5)
    circles = cv.HoughCircles(img,cv.HOUGH_GRADIENT,1,distance, \
                              param1=75,param2=40,minRadius=minrad,maxRadius=maxrad)
    circles = np.uint16(np.around(circles))

    return circles[0]
```

Beolvasásra kerül az éldetektált kép, és a minimum sugár, amit paraméternek kellett megadni a program indításakor. A maximum sugár a minimum kétszerese lesz, továbbá meg kell adni egy minimum távolságot, hogy ne detektáljon egymást átfedő köröket, ez a minimum sugár két és félszerese lesz.

A ***HoughCircles*** paraméterei a következők:
HoughCircles(image, method, dp, minDist, param1, param2, minRadius, maxRadius)

- **image:** Bemeneti kép.
- **Method:** A detektálás metódusa.
- **dp:** Kép arányának inverze. A példában 1, ami azt jelenti, hogy nem változik a kép. Kettes értéknél fele akkora lenne a szélessége és magassága.
- **minDist:** Körök középpontjai közötti minimum távolság.
- **param1:** Metódus-specifikus paraméter.
- **param2:** Metódus-specifikus paraméter.
- **minRadius:** Minimum sugara a körnek.
- **maxRadius:** Maximum sugara a körnek.

A körök adatai egy tömbben kerülnek tárolásra. Ezeket az értékeket még kerekíteni kell egész számokra, és át kell konvertálni integerré.

```
i=0
count=0
for circle in circles:
    log.write("{}. circle\n".format(i+1))
    log.write("x: {}, y: {}, r: {}\n\n".format(circle[0],circle[1],circle[2]))
    mask = np.zeros(edge_targ.shape[:2], dtype="uint8")
    cv.circle(mask, (circle[0], circle[1]), circle[2], 255, -1)
    cut = cv.bitwise_and(edge_targ, edge_targ, mask=mask)

    cv.imwrite(cutpath+str(i+1)+". cut.jpg", cut)
    matching.append([circle,[]])
```

A detektálás után ki kell vágni a körök területeit a képből. Ehhez először létre kell hozni a ***numpy*** modul segítségével egy nullákkal feltöltött integer típusú mátrixot. Majd az OpenCV ***circle*** függvényével létre kell hozni egy bináris képet, ahol az adott kör területe fehér, a háttér pedig fekete. Az elkészült kép lesz a kivágáshoz használt maszk. Ennél a résznél kerül inicializálásra a ***count*** változó is, ami a végeredmény tárolására lesz használva. Az ***i*** változó az érmék indexe, a kiíratások miatt lesz rá szükség.

A ***circle*** függvénynek az alábbi paramétereket kell megadni:
circle(img, center, radius, color, thickness)

- **img:** A kép, amire a kört rajzolja a program.
- **center:** A rajzolt kör középpontjának koordinátája.
- **radius:** A rajzolt kör sugara.
- **color:** A szín, amivel rajzolja a kört.
- **thickness:** A körvonal vastagsága.

A ***bitwise_and*** függvénnyel a maszk fehér pixelei helyére fognak kerülni az eredeti kép pixelei, a háttér üres marad.
A kivágás után a program kiírja a képet a célkönyvtárba, és a matching nevű listához egy új elem kerül hozzáadásra, aminek a két eleme a detektált kör adatai, és a későbbi összehasonlítás eredményeinek tárolására szolgáló üres lista.

```
j=0
for forint in images:
    value=0
    k=0
    for coin in forint:
        sample = cv.imread(coin)
        res_sample = cv.resize(sample, (256,int((sample.shape[1]/sample.shape[0])*256)))
        blur_samp = cv.GaussianBlur(res_sample, (3,3), 2)
        edge_samp = cv.Canny(blur_samp, 30, 150, 2)
        edge_samp = cv.dilate(edge_samp, (1,1), iterations = 2)
        if(i==0):cv.imwrite(samplepath+forints[j]+"."+str(k+1)+". sample.jpg", edge_samp)
```

Hasonló műveleteket kell végrehajtani a mintaérmék képein, mint amit a bemeneti képen, annyi különbséggel, hogy ezen egy kicsinyítést is végez a program a ***resize*** függvénnyel. A kicsinyítésre a gyorsabb működés miatt van szükség. A mintákról készülni fognak képfájlok is, amik belekerülnek a kimenet mappájába, hogy ellenőrizni lehessen az éldetektálásuk sikerességét. A minták kiírása csak az első hasonlításnál történik. A ***value*** változó minden forint típusnál nullázódik, az összehasonlítás pontosságát mutatja, minél nagyobb az érték annál jobban illeszkedik az érme a mintához. A ***j*** és a ***k*** változók rendre a forint típusokat és a forintok számát indexelik, ezekre a kiírások miatt van szükség.

```
        k1,k2,d1,d2=sift(cut,edge_samp)
        matches=flann(d1,d2)

        value += len(matches)

        log.write("Match with {}: {}\n".format(coin,len(matches)))

        result=showMatches(target, res_sample,matches,k1,k2,d1,d2)
        result = cv.resize(result, (1600,900))
        cv.imwrite(matchpath+str(i+1)+'.'+forints[j]+"."+str(k+1)+". keypoints.jpg", result)
        k+=1
```

Az érme és a minta közötti összehasonlítás a ***sift*** és a ***flann*** függvényekkel történik. A ***sift*** vissza fogja adni a jellemzőpontokat, és a leírókat, a ***flann*** ezeket összehasonlítja, és visszaadja a matches változóba a jó párosításokat. Majd a jó párosítások száma hozzáadódik a value változóhoz. A párosítás értéke beleíródik a logba, illetve létrejön egy result kép, ami ábrázolja az összehasonlítás eredményét a két képre elhelyezett kulcspontokkal, és vonallal köti őket össze, ezzel mutatva, hogy melyik melyikkel illik össze.

A ***sift*** függvény az alábbi módon néz ki:

```
def sift(img1, img2):
    detector = cv.SIFT_create()

    k1, d1 = detector.detectAndCompute(img1, None)
    k2, d2 = detector.detectAndCompute(img2, None)

    return k1, k2, d1, d2
```

Bemenete a két kép, kimenete a képek kulcspontjai és leírói lesznek. A ***SIFT_create*** függvény hívja meg a SIFT osztályát, majd a ***detectAndCompute*** fogja a számítást elvégezni.

A ***flann*** függvény:

```
def flann(d1,d2):
    index_params = dict(algorithm = 1, trees = 5)
    search_params = dict(checks=50)
    flann = cv.FlannBasedMatcher(index_params,search_params)
    knn_matches = flann.knnMatch(d1, d2, 2)

    ratio_thresh = 0.7
    good_matches = []
    for m,n in knn_matches:
        if m.distance < ratio_thresh * n.distance:
            good_matches.append(m)

    return good_matches
```

A függvény bemenete a két kép kulcspontjainak leírói, és a kimenete a helyes párosítások. Először létre kell hozni egy ***FlannBasedMatcher*** objektumot, az ***index_params*** és a ***search_params*** paraméterekkel. Az index paraméterekkel lehet megadni az algoritmus típusát, és a fák számát, az 1-es algoritmus jelöli a FLANN alapút. A keresési paraméterekkel az ellenőrzések számát lehet megadni. Ezután a ***knnMatch*** függvény elvégzi a két kép leíróinak az összehasonlítását. Az összehasonlítás alatt ki kell szűrni, mely értékek nem felelnek a küszöbértéknek, ami jelen esetben 0.7 lesz.

```
        value = value / k
        log.write("Average of {}: {}\n\n".format(forint,value))
        matching[i][1].append(value)
        j+=1
```

Az értékeket ki kell átlagolni, ugyanis az összehasonlítást egy érme típusának egy adott képére végzi el, de a kiértékeléshez az adott érme értéke kell. Ez az érték eltárolásra kerül a logban, és beleíródik a ***matching*** listába is

```
    largest_value=max(matching[i][1])
    indexof_largest =forints[matching[i][1].index(largest_value)]
    count+=int(indexof_largest)

    log.write("The value of the coin is: {}\n\n".format(indexof_largest))

    cv.putText(target,indexof_largest,(circle[0],circle[1]),cv.FONT_HERSHEY_SIMPLEX,1,(255, 0, 0, 0),2)
    cv.circle(target,(circle[0],circle[1]),circle[2],(0,255,0),2)
    print(i+1,". circle done.")
    i+=1
```

Hogy megkapjuk melyik érme van a képen, a ***matching*** tömb ***i***-edik elemének legnagyobb egyezést vesszük, majd ezt egyeztet a ***forints*** tömbbel, megkapjuk az érme típusát. A típusokat futás közben adja hozzá a ***count*** változóhoz a program, beleírja a logba, a képre pedig rárajzolja a kör vonalát, a kör közepébe beleírja a címlet nagyságát.

```
log.write("\nThe overall value of the coins is: {}\n\n".format(count))

cv.putText(target,"The value of the coins is: {}".format(count),(10,30),cv.FONT_HERSHEY_SIMPLEX,1,(255, 0, 0, 0),2)
cv.imwrite(output+"endresult.jpg", target)

log.close()

cv.imshow("Coin counting done",target)
cv.waitKey()
```

A program végén a végső értéket beleírja a logba, ráírja a képre, bezárja a logot, és a végső eredményt egy külön ablakban megjeleníti, ami egy gomb lenyomása után bezárul.

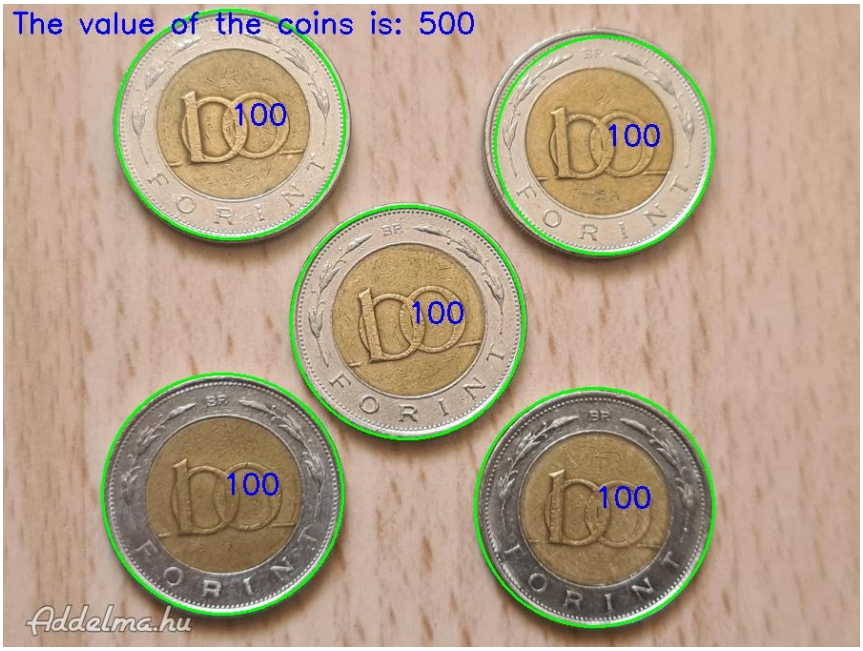
4. Tesztelés

Program lefuttatása a ***test*** mappában lévő képekkel.

test1.jpg
Körök minimum sugara: 50
Gauss kernel mérete: 3

test2.jpg
Körök minimum sugara: 100
Gauss kernel mérete: 5

test3.jpg
Körök minimum sugara: 100
Gauss kernel mérete: 5



test4.jpg
Körök minimum sugara: 100
Gauss kernel mérete: 5

test5.jpg
Körök minimum sugara: 50
Gauss kernel mérete: 3

test6.jpg
Körök minimum sugara: 75
Gauss kernel mérete: 3

5. Felhasználói dokumentáció

A program indításához be kell lépni a könyvtárába, és lefuttatni a ***coindetect.py*** nevű Python scriptet. A futtatáshoz három paramétert kell megadni, a vizsgálni kívánt kép elérési útvonalát, a körök minimum sugarát, és a Gauss kernel méretét. A ***test*** mappában érhetőek el a tesztkészlet képei. A ***db*** mappában lehet megtekinteni az összehasonlításhoz használt érméket.

```
python coindetect.py test/test1.jpg 50 3
```

Példa a program futtatására

A program futása közben üzeneteket fog küldeni a parancssorra. Sikeres futásnál ki kell írnia mennyi érmét detektált, illetve ha egy érmével végzett.

```
6  coins detected.
1 . circle done.
2 . circle done.
3 . circle done.
4 . circle done.
5 . circle done.
6 . circle done.
```

A program üzenetei futás közben

A program végeztével megjelenik egy ablak, ahol a végeredmény van ábrázolva. Ezt az ablakot bármilyen billentyűgomb lenyomásával be lehet zárni. Az eredményeket a program könyvtárán belül létrejött ***results*** mappában lehet megtekinteni, ahol az indítási idő időbélyegével ellátott mappába kell lépni. Itt lesz a logfájl, a Canny éldetektálás eredmény, a végeredmény, továbbá a ***matches*** mappában a kulcspontokkal ellátott összehasonlítások eredményei, a ***cuts*** mappában a kivágott érmék, és a ***samples*** mappában az éldetektált minták.

Irodalomjegyzék

- **https://docs.opencv.org**
- **https://www.inf.u-szeged.hu/~tanacs/pyocv/index.html**
- **https://www.inf.u-szeged.hu/~gnemeth/kurzusok/igazsagugyi_kepelemzes/kepijellemzok/a_sift_detektor.html**
- **https://vik.wiki/images/f/fc/IKM_2014_morfologia.pdf**
- **https://towardsdatascience.com/canny-edge-detection-step-by-step-in-python-computer-vision-b49c3a2d8123**
- **https://docs.python.org/3/**