

GÉPI LÁTÁS:

Pénzérme számláló alkalmazás

Gombás Gergely

(GKNB\_INTM038)

Tartalom

[Bevezetés 3](#_Toc122380388)

[Alapötlet: 3](#_Toc122380389)

[Feladatkifejtés: 3](#_Toc122380390)

[Fejlesztői leírás 4](#_Toc122380391)

[Szükséges elméleti háttér: 4](#_Toc122380392)

[Első lépés: 5](#_Toc122380393)

[Második lépés: 10](#_Toc122380394)

[Sugár alapján való meghatározás: 10](#_Toc122380395)

[Szín alapján való meghatározás: 11](#_Toc122380396)

[ORB feature extraction, jellemző pontok alapján való meghatározás: 12](#_Toc122380397)

[Felhasználói kézikönyv: 16](#_Toc122380398)

[Források: 17](#_Toc122380399)

# Bevezetés

## Alapötlet:

Manapság egyre többször alkalmazunk fizetésnél bankkártyát és sokkal kevesebb esetben fizetünk készpénzzel, de akkor is leggyakrabban inkább a papírpénzt választjuk, így összegyűlik mindig az „apró”, és sosem tudjuk, hogy éppen mennyi az érmék értéke. Erre a problémára fejlesztenék ki egy alkalmazást, amely segítene a felhasználónak egy képről megmondani, hogy hány forintnak megfelelő érme található a képen. Más, nagyobb országok pénzérméihez léteznek már hasonló programok ( mint például az euróhoz vagy a dollárhoz), de kifejezetten forintra specializálódó programot nem találtam. Ezért döntöttem úgy, hogy projektemben a forinttal mint pénznemmel foglalkozom, és az ahhoz fontos paramétereket és bemeneti adatokat használom fel. Természetesen más pénznemekkel is működőképes lehet az alkalmazás, viszont akkor az adott valuta érméinek paramétereit és azokhoz specifikált forrásokat kell alkalmazni.

## Feladatkifejtés:

A terv egy olyan pénzérme-számláló alkalmazás készítése, mely előre definiált adatok alapján megadja, hogy egy képen milyen érmék láthatóak, azoknak mennyi az értékük és a végén a program összeszámolja majd megmondja, hogy összesen mekkora értéknek megfelelő pénz látható a képen. Alapvetően készítettem hozzá egy fejlesztői leírást és egy rövidebb felhasználói leírást. A fejlesztői részben ismertetem a megoldásomat több fejezetre lebontva. A feladat megoldásának több módon, különböző ötletekkel is nekiálltam, több módszert is alkalmaztam és alapjában véve két nagyobb részre lehet felosztani: az első az érmék megtalálása a képen, a második a már megtalált érmék tulajdonságai alapján azonosítani az érméket és összeszámolni az értékeiket. Később ezeket a lépéseket fogom részleibe menően kifejteni, valamint saját tapasztalataimat bemutatni.

# Fejlesztői leírás

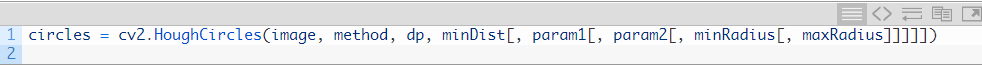
## Szükséges elméleti háttér:

A programot „python” nyelven írtam a „Pycharm” alkalmazás „community edition” változatában, a Python 3.10-es verzióját használtam. „Opencv” kiegészítőt telepítettem, hogy tudjak foglalkozni a képek manipulációjával. Az „opencv” letöltésével települ a „numpy” bővítmény is, így az „opencv” és a „numpy” függvényeit egyaránt fel tudtam használni. Ezenkívül importáltam az „os” kiegészítőt is, hogy tudjak foglalkozni a fájlok beolvasásával és az azokba való írással is. Ajánlott a választott programnyelv alapszintű tudása, mivel annak hiányában jelentősen megnehezedik a feladat elvégzése. Több nélkülözhetetlen, de annál egyszerűbb függvényt használtam fel, ugyanis ezeknek a megértése és használata nem tart sokáig és nem is bonyolult. A használt függvényeket részletesen bemutatom a későbbiekben.

## Első lépés:

A képen szöveg, különböző, különféle látható

Automatikusan generált leírásKezdésként szükségem volt egy bemeneti képre, amin különféle pénzérmék találhatóak. Itt fontos megjegyezni, hogy minél jobb a bemeneti kép minősége, annál könnyebb vele dolgozni, és annál jobb eredményt tudunk elérni a végén. A legegyszerűbb, ha az érméket egyszínű háttér elé helyezzük, talán a fehér vagy a fekete a legelőnyösebb. Egy előre beállított, tényleg ideális környezetben, fehér háttér előtt készített képpel dolgoztam a tesztelések elején, hogy az esetleg felmerülő összes hiba és probléma a program pontatlanságát jelezze, ne a kép minőségbeli hibáit. A továbbiakban végig ezen a képen mutatom be a lépéseket. Ahhoz, hogy megtaláljuk a képen az érméket, először létre kell hoznunk a kép szürkeárnyalatos verzióját, ez azt eredményezi, hogy minden egyes képponthoz egy színkódérték társítható és nem 3, mint a színes képeknél. Ezáltal könnyebb lesz észrevenni a különbségeket az egyes pixelek között. Itt az „opencv” saját metódusát alkalmaztam, a ***cv2.cvtColor()*** -t. A soron következő lépés a kép elhomályosítása, hogy a nagyon kiugró értékek se rontsák el az érmék megtalálását, ehhez szintén „opencv” függvényt használtam, ***cv2.blur()*** -t. 3\*3-as kernellel mentem végig a képen és simítottam ki a részleteket. Ezek után érkezünk el ahhoz a ponthoz, ahonnan több irányba is tovább lehetett lépni. Próbáltam megtalálni a tökéletes megoldást az érmék körüli körök létrehozására, eleinte saját megoldásokkal, de ez sehová sem vezetett, mert nagyon sok időt igényelt, és az eredmények sem lettek elfogadhatóak. Pontatlanul dolgozott a program és innen sajnos nem tudtam előrébb jutni.

Ami viszont szinte tökéletesen működött, az a ***Hough Circle Transformation***. Nagyon szépen megtalálja az „opencv” saját függvénye a ***cv2.HoughCircles()*** az érmék körüli köröket és minden körhöz hozzárendel 3 értéket. Az egész metódus úgy működik, hogy egy képre lefuttatva számtalan eltérő és egymáshoz nagyon közel eső kört detektál és ezekből kiválasztja a legpontosabbnak tűnőket, ezt egy „szavazással” dönti el. Ebben tudunk segíteni a programnak a paraméterek pontos megszabásával.

* **image**: első paraméternek a képet kell megadnunk, amin szeretnénk alkalmazni a függvényt. Ide az eddig használt, már előkészített, szürkeárnyalatos és elhomályosított képünket adjuk meg.
* **method**: második paraméterként a körök észlelésének módját kell meghatároznunk, viszont jelenleg egyetlen megvalósított technika létezik az „opencv”-ben, ez a ***cv2.HOUGH\_GRADIENT***, végig ezzel a módszerrel dolgoztam.
* **dp**: harmadik paraméterünk a ***dp***, mely az akkumulátor-felbontás és a kép felbontásának fordított aránya, ha ez az érték minél nagyobb, annál kisebb lesz az akkumulátortömb. Hogyha az érték 1, akkor megegyezik a bemeneti kép felbontásával, de például, ha 2, akkor az akkumulátortömb fele olyan magas és széles lesz.
* A képen szöveg, kültéri látható

  Automatikusan generált leírásA képen szöveg látható

  Automatikusan generált leírás**minDist**: következőnek azt kell megadnunk, hogy kettő létrehozott kör középpontja között mekkora lehet a legkisebb távolság. Itt sokat kellett játszanom az érték változgatásával, mivel ha túl alacsony, készülhetnek felesleges körök, ha viszont túl nagy, akkor nem veszi figyelembe az összes érménket.

Itt látható egy eset, mikor túl alacsonyra van állítva ez az érték és egy érmén belül több másikat is talál és egy másik, amikor olyan magasra lett beállítva, hogy az egymáshoz közeli érméket már nem találja meg.

Innentől olyan paramétereket mutatok be, amik nélkül is működőképes tud lenni a metódus, viszont a pontos és precíz megoldáshoz elengedhetetlen.

* **param1**: módszer-specifikus paraméter, ha a ***cv2.HOUGH\_GRADIENT*** választjuk megoldási módszernek és egyelőre csak ez az egy opciónk van. Ebben az esetben „Canny” éldetektorral dolgozik a függvény és az itt megadott érték lesz a felső, legmagasabb küszöb. Ezt az értéket 50-nek határoztam meg a tökéletes működéshez.
* **param2:** egy olyan módszer-specifikus paraméter, küszöbérték, ami kör középpontjának észlelésekor használatos. Minél kisebbre választjuk a küszöböt, annál több kört detektálunk, viszont sok lesz a hamis is. Hogyha ezt az értéket túl magasra választjuk, akkor a megtalált körökből többet vet el, ezáltal kevesebbet választ ki és ad vissza ténylegesen. A paraméter értékét 30-nak választottam, és ezáltal jól működött a metódus.
* A képen szöveg látható

  Automatikusan generált leírás**minRadius:** ez a megengedett legkisebb sugárnak az értéke, amelyet még elfogadunk az érmék körüli körök készítésénél, ez is sokat segít abban, hogy ne készüljenek felesleges körök a képen.

Itt kipróbáltam, hogy mi történne, ha a minimális kör sugarat nullának választanám. A képen is jól látható, hogy készülnek olyan körök, amelyek sokkal kisebbek, mint bármelyik pénzérme, és más hibákat is felvetnek, mint például két érme közé is került egy kör, ahol egyébként egy érme sem található. Ez a hiba a minimális kör középpontok közötti távolság nem jól megválasztott érteke miatt is felmerülhet. A ***minRadius*** értékét én alapvetően 25 környékére állítottam és így hibátlanul működött.

* A képen szöveg látható

  Automatikusan generált leírás**maxRadius:** az utolsó paraméterünk a készítendő körök sugarának maximális értéke, erre is azért van szükség, hogy kiküszöböljük az olyan eseteket, amikor a számunkra hasznos köröknél nagyobb helytelenül detektált köröket is figyelembe venné a program.

A képen szöveg látható

Automatikusan generált leírásEbben az esetben 80-ra állítottam a maximális sugár értékét és egyből megjelentek olyan körök, amelyek hamisak és megtévesztőek. A tökéletes paraméter-értéket 60 körül maximalizáltam.

És végül így néz ki, ha az összes paramétert jól tudjuk meghatározni, ezeket az értékeket úgy sikerült megtalálnom, hogy minél több bemeneti képen teszteltem, és azt választottam ki, amely a legtöbbhöz passzolt. Minden pénznemhez más paramétereket érdemes beállítani és forintnál is a kép felbontásától függ, hogy milyen értékekre van szükségünk. A forrásképeket ugyanazzal az eszközzel készítettem és ugyanolyan felbontásban használtam fel. Fontos megjegyeznünk, hogy a ***cv2.HoughCircles()*** futtatás után 3 paramétert ad vissza, az egyik a detektált körök sugara a másik kettő az x és y koordinátája. Ezen értékek segítségével folytattam tovább a megoldásomat.

A ***Hough Circles*** transzformáció után a vissza adott értékek nem kerek számok, viszont a későbbiekben a pontos számoláshoz arra lenne szükségünk, ezért ***numpy.around()*** függvénnyel egészekre kerekítettem, majd átkonvertáltam integer típusba. Ezt követően ***cv2.circle()*** metódussal minden kör középpont koordinátájába rajzoltam egy 1 sugarú kört, ez jelzi, hogy hol helyezkedik el a kör középpontja és az előre meghatározott sugár értékével is készítettem egy kört, ami az érme körül rajzolódik.

## Második lépés:

Miután meghatároztuk, hogy egy képen pontosan hol találhatóak érmék, következő lépésünk, hogy megadjuk egy-egy érme értékét és a végén összeszámláljuk őket.

### Sugár alapján való meghatározás:

Eleinte egy olyan megoldással próbálkoztam, hogy minden eltérő pénzérmének közel egyforma sugárral kell rendelkeznie, viszont hogyha a kép egy másik szögből készül, akkor ez a sugárérték más lesz. Innen támadt egy ötlet, hogy egy fix tárgyat, akár egy *sim-kártyát* vagy egy igazolvány kártyát elhelyezek a képen és mindig ezt a tárgyat tekintem kiindulópontnak. Miután ezt a tárgyat megtaláljuk, ehhez hasonlítva minden érmének lesz egy előre megadott értéke és csak a fix tárgy eredeti állapotától való eltérés arányával kell vizsgálnunk az érmék jelenlegi sugarát. Mindazonáltal ez nem egy jó megoldás, mivel ebben az esetben minden képen szerepelni kell ennek a tárgynak és mindig ugyanannak a tárgynak. Ehhez hasonlóan volt egy másik ötletem is, mikor minden képre elhelyezünk minimum egy ötforintost és ehhez ugyanúgy definiálunk minden egyes érméhez egy sugárarányt, a vizsgálatnál pedig a legkisebb pénzérméhez hasonlítjuk a többit. Ez sem jó megoldás sajnos, mert lehetnek olyan esetek, mikor nem használunk ötforintos érmét.

### Szín alapján való meghatározás:

Ezek után nagyon sok megoldási lehetőséget kerestem, de egyik sem bizonyult pontosnak és hasznosnak. Egy következő saját megoldással próbálkoztam, mikor az érmék körüli körök középpontjában megvizsgálom a színértékek átlagát. Minden középpontnál egy 3\*3-as mátrixban vettem figyelembe a színkódokat. Itt olyan problémákba ütköztem, mint például megcsillanó fény vagy fekete foltok az érméken, ezért ezeket a szélsőséges színkódokat ki kellett szűrnöm, mivel nagyban befolyásolnák az átlagok értékét. Minden érméhez készítettem több kép alapján egy fájlt, amibe kiírtam a képekről származó átlagolt színkódokat érmék szerint. Később ehhez hasonlítottam a bemeneti képek érméit. Voltak olyanok, ahol meglehetősen jól működött, de mivel ez saját megoldás és nem tökéletesen pontos, ezért sok esetben hibásan határozta meg a keresendő értékeket. Amellett, hogy a képre a pénzérmékhez kiíratom az értéküket, meghatároztam egy egyezési százalékot is, melyet szintén megjelenítek.

A képen szöveg látható

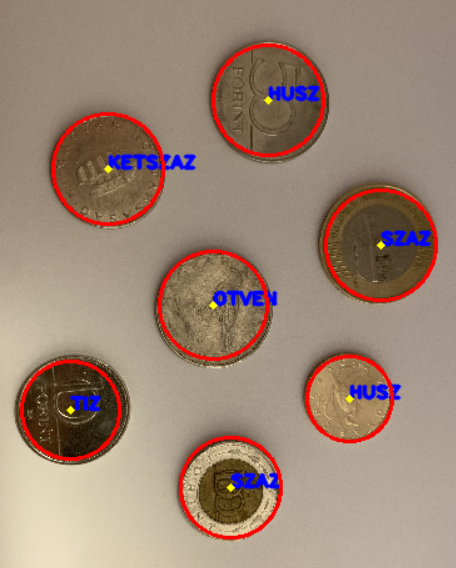
Automatikusan generált leírásA képen szöveg látható

Automatikusan generált leírás

A két példa-képen is jól látszik, hogy bármennyire próbálkoztam, az ideális környezetet nem egyszerű megtalálni és egy-egy beárnyékolódás vagy éppen fénycsillanás mennyire félre tudja vezetni a programot. Azt is fontos megjegyeznünk, hogy több érménk is hasonló színű és ez sok esetben megtévesztő lehet. Következtetésként, ez egy működőképes megoldás, viszont nem túl pontos.

### ORB feature extraction, jellemző pontok alapján való meghatározás:

Végső megoldásnak az ORB algoritmust találtam. Ez már sokkal pontosabban és jobban optimalizálva működött, mint az ezelőtti megoldási próbálkozások. Az algoritmus a képen a fontosnak és meghatározónak tűnő részleteket keresi és találja meg, ami lehet egy él vagy egy szimbólum. Ezeket az adatokat letárolja a ***descriptorok***ban és minden ilyen fontos jellemzők megtalálásához elment egy kulcspontot, ami a megjelenítésben fontos. Az ORB hasonlóan működik, mint a SIFT vagy SURF, amiket ugyanerre a probléma megoldására fejlesztettek ki, viszont azok ***license*** kötelesek és fizetni kellene értük. Az ORB az „opencv” saját fejlesztése és ezáltal ingyenes. Két másik algoritmust ötvöz egybe, az egyik a ***FAST***, aminek segítségével nyeri ki a szükséges kulcspontokat a másik a ***BRIEF***, melynek a támogatásával tudja hozzárendelni a meghatározott jellemzőkhöz a ***descriptorok***at. Első lépésnek létrehozunk egy ORB objektumot, amihez a ***cv2.ORB\_create()*** függvényt alkalmaztam. Majd egy változóba kimentettem a kulcspontokat az ***ORB\_object.detect()*** metódus segítségével, itt paraméternek meg kell adni vizsgálandó képet. Mivel az előre detektált pénzérmék meghatározó részleteire volt szükségem, ezért a már kész körök tanulmányozásánál futtattam és minden kör középpontjától a körsugarával készítettem egy négyzetet, a képnek ezen kivágott részét adtam meg a függvény paraméterének. Ezt követően a ***ORB\_object.compute()*** metódust alkalmazva határoztam meg a ***descrpitorok***at egy-egy körnél. Két paramétert kell megadnunk, az egyik az előbb meghatározott kulcspontok a másik a kép, vagy esetünkben a képrészlet, amit vizsgálunk. Több előre elkészített, és csak egy érmét tartalmazó képet vettem alapnak, minden különböző érmét tartalmazó képhez elkészített ***descriptorok***at kimentettem egy fájlba és érménként mappákba rendeztem őket. Később ezeket hasonlítottam, az éppen vizsgálandó képhez, kép részlethez. A hasonlítás a ***descriptorok*** segítségével tudott létrejönni a ***cv2.BFMatcher()*** metódust alkalmaztam és vizsgáltam az azonosságokat, melyeket kimentettem egy tömbbe és minden alkalommal megnéztem, hogy mennyi a jelenlegi egyezés, ha találtam az addigi legnagyobbnál nagyobbat, akkor ő lett a maximum és a legvégén annak a mappának a nevét írattam ki az érme középpontjába, amelyikből származik a legnagyobb egyezés. A ***BFMatcher*** függvény úgy működik, hogy felveszi az első kép egyik jellemzőjének ***descriptorát,*** és a második kép összes többi jellemzőjével párosítja valamilyen távolságszámítás segítségével. És a végén legközelebbit adja vissza. Mikor futtatjuk a metódust, először létre kell hoznunk a ***BFMatcher*** objektumot, erre a ***cv2.BFMatchert()*** alkalmaztam, kettő paramétert vár a függvény. Az első a ***normType,*** amely meghatározza a használandó távolságmérés módját, itt a ***cv2.NORM\_HAMMING***-et érdemes választani, mert az ORB-hez ez passzol legjobban, ilyenkor a ***Hamming*** távolságot használja fel a méréshez. A második paraméter egy logikai változó, a ***crossCheck***, aminek az alapértéke hamis, ha igazra állítjuk, akkor két egymáshoz hasonlított kép legpontosabb egyezéseit keresi, amik szinte ugyanúgy megtalálhatóak mindkét képen. Miután létrehoztuk az objektumot 2 nagyon fontos metódust tudunk használni, a ***BFMatcher.match()***, a legjobb egyezést adja vissza és a ***BFMatcher.knnMatch()***, ami a „k” darab legjobb egyezést adja vissza, itt a „k” értékét mi szabhatjuk meg. Sokkal pontosabb végeredményeket sikerült kinyernem, amikor csak a legjobb egyezést vizsgáltam, ezért én a saját megoldásomnál végig ezzel dolgoztam.



A 2 példaként használt képen látszik az eredmény, ez már pontosabb, de még mindig nem tökéletes. Nagy igazság, hogy egy programot sosem lehet teljesen pontosra és tökéletesre elkészíteni, mivel mindig lehetnek apró hibák, amelyek befolyásolják a teljesítményüket. Lehet azonban egyre jobbra és jobbra fejleszteni. Ahhoz, hogy szemlétesebben bemutassam az ORB és ***BFMatcher*** működését a meghatározott érme után kiírattam az egyezések számát.

A képen szöveg látható

Automatikusan generált leírásA képen szöveg látható

Automatikusan generált leírás

Végezetül kiszámoltam az értéket, amelyet a program meghatározott és megjelenítettem minden kép bal felső sarkában. Ez az érték sajnos nem pontos, mivel a program sem tudja precízen visszaadni egyenként az érmék értékét.

A képen szöveg látható

Automatikusan generált leírás

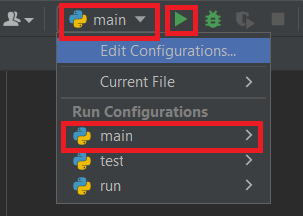
A program legutolsó fázisához készítettem 100 teszt képet saját kezűleg, mivel forinthoz nem találtam egy előre elkészített datasetet, ezeket a képeket egy külön testimages mappában helyeztem el a teszteléshez.

# Felhasználói kézikönyv:

A következőkben bemutatom, hogyan tudjuk mi is letölteni és beüzemelni az általam készített pénzérme-számláló alkalmazást. Először is kattintsunk a következő linkre: <https://github.com/ggombas36/Gepilatas_penzerme> . Itt meg tudjuk tekinteni az egész projektet, az összes verziójával. Következő lépés, hogy letöltjük a ***Pycharm*** alkalmazást és létrehozunk egy új projektet. Telepítjük a ***python packages*** résznél az ***opencv-python*** bővítményt és ezáltal a ***numpy*** bővítmény is automatikusan letöltődik.

A képen szöveg látható

Automatikusan generált leírás

Ezután a ***github-ról*** a *main.py* fájlból átmásoljuk a kódot a sajátunkba. Ezt követően lementjük az „***src”*** mappát és bemásoljuk a programunk mappájába. Az ott levő teszt képek segítségével tudunk akár tesztelni, ha meghívjuk a ***detectionORB()*** metódust és paraméternek megadjuk a tesztkép elérési útvonalát. Ha saját képet szeretnénk tesztelni, akkor azt a projekt alatt futó egyik mappába kell feltöltenünk, és szintén a függvény paraméterének kell megadnunk az elérési útvonalát. Most már csak futtatnunk kell, és kész is van a program. A futtatásnál a jobb oldalon fent ki kell választani, hogy a ***main.py*** fusson és mellette a kis zöld *play* gombra kell nyomni. Ezek után már láthatjuk is az eredményt jobb felső sarokban az összes képen látható pénzérme összege és minden érmén a középpontban az értéke, és hogy mekkora egyezés tartozik hozzá, ezen kívül minden érme körül látható egy kör, amely azt mutatja, hogy a program sikeresen felfedezte az érme elhelyezkedését.

# Források:

* <https://numpy.org>
* <https://opencv.org/>
* <https://medium.com/@isinsuarici/hough-circle-transform-in-opencv-d74bdf5161ed>
* <https://pyimagesearch.com/2014/07/21/detecting-circles-images-using-opencv-hough-circles/>
* <https://www.educba.com/opencv-orb/>
* <https://www.geeksforgeeks.org/feature-matching-using-orb-algorithm-in-python-opencv/>
* <https://opencv24-python-tutorials.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_matcher/py_matcher.html>