2. feladat

Előfordul, hogy egy fényképen nehezen kivehetőek egyes részletek, mert sok hasonló árnyalatú képpont van egymás közelében. Ilyenkor jobban láthatóvá tehetjük a részleteket az ún. “hisztogram kiegyenlítés" módszerrel.

A módszer lényege, hogy úgy transzformáljuk a képpontok intenzitásértékeit, hogy a gyakorisági hisztogram, (azaz a lehetséges intenzitás értékek gyakorisága az érték függényében nézve) közel konstanssá váljon, ugyanakkor azonban megtartjuk az intenzitás értékek sorrendjét. Ez utóbbi úgy értendő, hogy ha egy pixel intenzitása a másikénál nagyobb, akkor a transzformáció után is nagyobb vagy egyenlő lesz. (A pixelek értékét egy monoton függvénnyel transzformáljuk, lásd még pl. a Wikipedia Histogram_equalization című cikkét.)

Írjunk C programot, ami az a) és b) pontbeli előkészítés után kipróbálja ezt a módszert, egymás után végrehajtva az alább részletezett lépéseket.

Az első parancssori argumentum legyen az átalakítandó képfájl neve. Második és harmadik a kép vízszintes és függőleges mérete pixelben.

A feladathoz tartozó képfájlok a /v/courses/halnum.public/feladat2/ találhatóak.

Fekete-fehér képfájlok: photo1c1280x720.dat, photo2c1280x720.dat, secret1280x720.dat

Színes képfájlok: Auronzo1280x720rgb.dat, Braies1280x720rgb.dat.

Összehasonlító képek: Auronzo.jpg, Braies.jpg

Ezeket fájlokat ne másolják át a saját mappájukba, hanem a megfelelő elérési útvonal megadásával használják őket.

a)

A program töltse be a képpontok intenzitását az első parancssori argumentumban megadott nevű fájlból. Tesztelésre használjunk saját próbafájlt és a weboldalon e feladatnál megadott .dat kiterjesztésű fájlokat. Utóbbiaknál a kép méreteit a képfájl nevéből lehet leolvasni.

E képfájlok formátuma a következő: a képpontokat sorfolytonosan rendezve, az intenzitásuk szerepel szöveges formátumban egymás után.

Elegendő az adatokat egy vektorba olvasni, mert az alábbi transzformációknál nem számít, hogy hány pontonként van sorokra tagolva az eredeti képben. Mivel az eredeti képfájlban 1 byte ír le egy pixelt, a számértékekek 0-tól 255-ig terjedő egész számok, ezt felhasználhatjuk a beolvasásnál.

Ezután készítsünk az adatsorból hisztogramot! Ez úgy értendő, hogy kijelölünk egy kiindulási értékkészlet-intervallumot, és azt valahány egyforma részre osztjuk. Majd megszámláljuk, hogy az egyes intervallumokba hány adat esik. Ehhez írjunk olyan függvényt, aminek meg lehet adni a kiindulási intervallumot, és hogy hány részre osztjuk. A függvény pedig adja vissza egy tömbben az egyes intervallumokhoz tartozó adatszámokat, és azokat írassuk ki!

Javasolt, hogy ne minden intervallumhoz olvassuk végig az adatsort. Egyszer menjünk végig az adatsoron, és mindegyik adatnál eldöntve, hogy hanyadik intervallumba esik, az annak megfelelő darabszámot növeljük meg! (Egy kis segítség: tekintsük az adatnak a kiindulási intervallum bal végétől vett távolságát és ennek viszonyát az osztásintervallumok hosszához)

A függvényt teszteljük kis próbafájlokkal! Ha a függvény jól működik, akkor a képfájlt elemezve futtassuk le úgy, hogy a 0-256 intervallum 16 vagy 32 részre osztását kérjük tőle, és az eredményt írassuk egy hist.dat nevű fájlba! A fájl első oszlopában az intervallumok közepei legyenek, második oszlopában a hozzájuk tartozó darabszámok!

Ezután jupyter notebookban olvassuk be és ábrázoljuk oszlopdiagramként, pl. az alábbi módon:

```
%pylab inline
hist = loadtxt("hist.dat")
bar( hist[:,0], hist[:,1], width=hist[1,0]-hist[0,0] )
```

b)

Számoljuk ki a hisztogram függvényt 256 intervallumra osztással és ennek segítségével határozzuk meg a számértékek kumulatív eloszlását!

Ebben az esetben tehát a hisztogram azt adja meg, hogy az egyes $i = 0, 1, \ldots 255$ számértékek hányszor fordulnak elő. Ezt fogjuk n_i-vel jelölni. A kumulatív eloszlás azt adja meg, hogy az i érték alatt összesen hány adat van, és ezt az alábbi képlettel kapjuk meg:

$$c_i = \sum_{j=0}^{i} n_j$$

(A wikipedia oldalon a sűrűség függvény szerepel, azaz a pixelek számával le van osztva. Itt azonban a fenti egyszerűbb leírás javasolt.)

Írjuk ki az eredményt egy cumul.dat nevű fájlba, most az i és c_i értékek legyenek két oszlopban! Ezután a már megkezdett jupyter notebookban a plot() függvénnyel rajzoljuk is ki a kumulatív eloszlást! Pl. így:

```
cumul=loadtxt("cumul.dat")
plot(cumul[:,0],cumul[:,1])
```

c)

A következő lépésben egy új tömböt szeretnénk létrehozni, amiben az értékek eloszlása közel egyenletes. Ez akkor fog teljesülni, ha az új tömb kumulatív eloszlása közel lineáris. Ezt pedig úgy tudjuk elérni, ha gondolatban sorba rakjuk a pixeleket értékük szerint, és mindegyiknek olyan új értéket adunk, ami e sorrendben való sorszámával arányos. Egymás utáni azonos értékek esetén a sorrend nem egyértelmű, ilyenkor ezek közül az utolsónak a sorszámát tekintjük. Ez a sorszám pedig egy i intenzitású pixel esetében éppen az addigiak darabszáma, azaz c_i értéke. A maximális 255 értéket figyelembe véve ez úgy valósítható meg, ha minden pixel i intenzitás értékét a következő összefüggés szerint transzformáljuk:

$$k = \mathrm{floor}(\, 255 \cdot c_i \,/\, n \,)$$

Itt n pedig a képpontok száma. Láthatjuk, hogy a c tömb indexébe kell az adott pixel i értékét tenni, és az eredményt az új tömb megfelelő helyére eltárolni.

Írjuk ki a pixelek új értékeit a beolvasottal azonos formátumban egy equalized.dat nevű fájlba! Egész számokként írjuk ki, különben feleslegesen nagy fájl keletkezik!

A jupyter notebookban pedig olvassuk be az eredeti képfájlt a loadtxt() függvénnyel, és jelenítsük meg a következő utasításokkal:

```
mx = loadtxt( 'fájlnév', dtype=int )
figsize( 12, 7)
axis( 'off' )
imshow( -resize( mx, (720,1280) ), cmap='Greys' )
show()
```

Ismételjük meg ezt a transzformált képpel, közvetlenül az előbbi alatt, hogy jól összehasonlítható legyen! (A)

d)

Készítsük el a transzformált képre is a gyakorisági és a kumulatív hisztogramokat! A gyakorisági hisztogramnál 16 vagy 32 intervallumot hozzunk létre. A kumulatív eloszlás számolásához viszont egységnyi osztásintervallumot használjunk! Használjunk új fájlneveket, pl. histeq.dat és cumuleq.dat! Állapítsuk meg, hogy megfelelnek-e az elvárásoknak! (A)

e)

Próbáljuk elvégezni a hisztogram-kiegyenlítést színes képen!

Leggyakrabban a pixelek színeit R (red), G (green) és B (blue) komponenseire bontva tárolják, lásd az "rgb.dat" végződésű fájlokat. Ezekből kell egyet választani a megoldáshoz. Ezekben a szokásosabb tömör formátum helyett az a) részben leírthoz hasonlóan szöveges formátumban szerepelnek az értékek, hogy az eddig tanult utasításokkal be tudjuk őket olvasni. Egy pixel 3 komponensének értéke egy sorban szerepel, R,G,B sorrendben, 0--255 értékű egész számként.

Próbáljuk ki, hogy milyen eredményt kapunk, ha a hisztogram kiegyenlítés módszerét a három komponensben egymástól függetlenül végezzük el! Az eredményt a beolvasotthoz hasonló formátumban fájlba írva ábrázoljuk pythonban az eredeti és a kiegyenlített képet! Mindkét képet így tudjuk megjeleníteni:

```
im = resize(loadtxt( 'fájlnév',dtype=uint8),(720,1280,3))
figsize(12,7)
axis('off')
imshow(im)
show()
```

Vajon ez a módszer elrontja a színeket? Összehasonlításul a képek jobb minőségű változata a weboldalon található jpg formátumban. (T)

f) Szorgalmi

Próbáljuk a kiegyenlítést úgy végezni, hogy jobb színeket kapjunk! Pl. használhatunk ehhez hsv vagy hls kódolást. E kódolások lényege, hogy a pixelek színeit rgb értékeik helyett a következő értékekkel jellemzik: h=hue, azaz a szivárvány színei mentén mért színárnyalat; s=saturation, azaz színtelítettség; v=value, ami lényegében intenzitás. A hls kódolás esetén pedig l=lightness egy másképp mért világosság. Ha hsv vagy hls értékhármasra átszámoljuk minden pixel rgb értékeit, és a v ill. l értékben végzünk csak hisztogram kiegyenlítést, akkor várhatóan jobb eredményt kapunk. (A végén persze vissza kell transformálni rgb értékekre, majd megjeleníteni.) Ezt a feladatrészt megengedett (és egyszerűbb) pythonban oldani meg, pl. a colorsys vagy más csomagok függvényeinek használatával.