OEFENINGEN PYTHON – REEKS 5

Signaal- en beeldverwerking

In deze oefeningenreeks gaan we enkele eenvoudige toepassingen bestuderen in het domein van signaal- en beeldverwerking. In de eerste oefeningen beschouwen we een 1-dimensionaal signaal dat essentieel gewoon een lijst van getallen is. Daarna gaan we naar grijswaarde afbeeldingen, dit zijn matrices van getallen (in Python: een lijst in een lijst). In de laatste oefening gaan we op een kleuren afbeelding werken, dit zijn ook matrices maar elke pixel wordt nu beschreven met 3 getallen (in Python heb je dus een lijst van 3 getallen in een lijst in een lijst).

Alle bestanden die je nodig hebt, zitten in het zip-bestand (reeks5_data.zip) dat je kan downloaden van de website. Pak dit zip-bestand uit op je computer en zorg dat alle python code die je schrijft wordt opgeslagen in diezelfde map zodat je eenvoudig de juiste bestanden kan inlezen).

Vraag 1: Ruis verwijderen in een 1-dimensionaal signaal

Zet bovenaan je programma de volgende import:
from pylab **import** figure,title,subplot,plot,show

a) Lees de gegevens van het bestand sig1.txt als een lijst *sig1* van floats. Plot dit signaal met behulp van de volgende code:

```
figure()
plot(sig1)
show()
```

Dit signaal bevat ruis en we gaan proberen deze te verwijderen door het signaal te filteren. Formeel kunnen we de filter operatie schrijven als $y[n] = \sum_{i=0}^{h-1} x[n-i] * f[i]$
In deze vergelijking is y[n] de n'de sample in het gefilterde signaal, x[n-i] de (n-i)'de sample van het signaal en f[i] is de i'de waarde in het filter dat we toepassen.
Voor deze oefening kiezen we een filter-grootte h (bijvoorbeeld **h = 7**) en stellen f[i] = 1/h. Als h = 4, dan ziet f eruit als f = [0.25, 0.25, 0.25, 0.25].
Als we dan de filter toepassen dan is in het gefilterde signaal de waarde op plaats n gelijk aan het gemiddelde van de niet-gefilterde waarde op plaats n en de (h-1) voorgaande niet gefilterde waarden. Implementeer dit filter, plot het gefilterde signaal en test het voor verschillende waarden van h.

b) Lees de gegevens van het bestand sig2.txt weer als een lijst *sig2* van floats en plot dit signaal weer. Zoals je kan zien is ook hier weer ruis aanwezig, maar de ruis is hier van een ander type. Voer je code uit deel a weer uit en sla dit resultaat op in de variabele *filt1*. Schrijf nu een 2de filter dat de mediaan gebruikt in plaats van het gemiddelde en creëer zo een gefilterd signaal *filt2*. Met behulp van de volgende code en het originele ruisvrije signaal *noiseless* in noiseless.txt, vergelijk de resultaten van *filt1* en *filt2* voor **h = 7**. (Heb je enig idee waarom een van de filters beter zou werken dan het ander?)

```
figure()
plot(sig2)
title('Signaal met ruis')
figure()
subplot(2,1,1)
plot(filt1)
plot(noiseless,color='r')
title('Filtering met gemiddelde')
subplot(2,1,2)
plot(filt2)
plot(noiseless,color='r')
title('Filtering met mediaan')
show()
```

Vraag 2: Afgeleiden

In signaalverwerking en voornamelijk in beeldverwerking benaderen we vaak de afgeleide van een signaal door simpelweg het verschil tussen 2 opeenvolgende samples/pixels te berekenen. Herinner je dat de afgeleide van een functie wordt gedefinieerd als:

$$\frac{df(t)}{dt} = \lim_{h \to 0} \frac{f(t) - f(t-h)}{h}$$

De volgende code genereert 5 seconden van een sinus-signaal $\sin(2\pi f * t)$ en tegelijk ook de afgeleide van dit signaal $2\pi f * \cos(2\pi f * t)$ met een frequentie $f = 2Hz$. Beide signalen zijn continu, maar worden gesampled aan fs samples per seconde. Hoe groter deze sample-frequentie, hoe 'nauwkeuriger' we het signaal kunnen plotten.

```
from math import sin,pi,cos
from pylab import figure,subplot,plot,show,legend

t_end = 5
fs = 100
f = 2

sig = []
dsig = []
t_sig = []

t = 0
while t <= t_end:
    x = sin(2*pi*f*t)
    dx = 2*pi*f*cos(2*pi*f*t)
    sig.append(x)
    dsig.append(dx)
    t_sig.append(t)
    t += 1.0/fs
```

Bereken nu de afgeleide door het signaal sig te filteren met het filter d = [+1,-1] zoals in de vorige oefening. Het gefilterde sample op index n is dus gelijk aan het originele sample op index *n* min het originele sample op index *(n-1)*. Dit stuk geeft je de teller in de formule voor de afgeleide. De h in de noemer is de tijd die tussen 2 opeenvolgende samples zit en is dus gelijk aan $1/fs$.

Plot je resultaat met behulp van de volgende code en kijk hoe de waarde van fs de nauwkeurigheid van het resultaat beïnvloedt door ze kleiner te maken:

```
figure()
plot(t_sig,sig,label="f(t)")
plot(t_sig,dsig,color='g',label="f'(t)")
plot(t_sig,filt,color='r',label="gefilterd")
legend()
show()
```

Vraag 3: Grijswaarde afbeeldingen

Zet de volgende imports bovenaan je code:
from pylab **import** figure,imread,imshow,show
Lees de coins.png afbeelding gebruik makende van de *imread* funtie:
I = imread('coins.png')
Deze afbeelding is 300x246 pixels groot en bevat enkel grijswaarden (geen kleur). Deze grijswaarden worden normaal gezien opgeslagen in 8 bits en kunnen dus waarden aannnemen tussen 0 en 255. In de bibliotheek die wij gebruiken zijn deze 8 bit waarden herschaald naar **float** waarden tussen 0 en 1. Deze pixels zijn per rij opgeslagen in een geneste lijst.
len(I) zou 246 moeten teruggeven: onze afbeelding bevat 246 rijen.
len(I[0]) zou 300 moeten teruggeven: elke rij in de afbeelding bevat 300 pixels.
I[0][0] is dus de pixel helemaal links bovenaan in de afbeelding.

We kunnen deze afbeelding tonen via de *imshow* functie. Hier moeten we ook nog zeggen dat de afbeelding met grijswaarden getekend moet worden:
imshow(I,cmap='gray')
show()

We gaan nu een "edge detector" schrijven die automatisch de randen in een afbeelding kan zoeken. Een rand definiëren we als de posities in de afbeelding waar de grijswaarde plots sterk verandert. Herinner je dat de "mate van verandering" wiskundig wordt gegeven door de afgeleide. Voor een 2-dimensionale functie f(x,y), zoals onze afbeelding, door de gradiënt $\vec{\nabla}\mathrm{f} = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}\right)$. In de vorige oefening hebben we gezien hoe we 1-dimensionale afgeleiden konden berekenen.

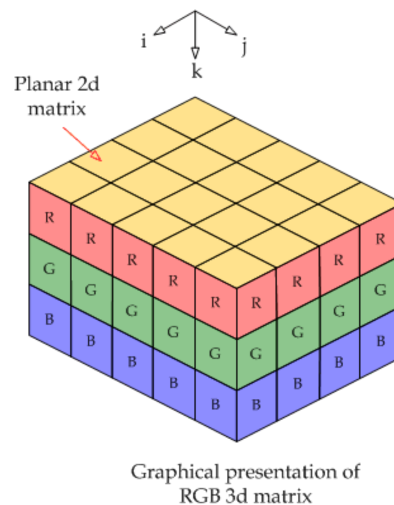
Schrijf een programma dat de randen in een afbeelding kan herkennen door de volgende stappen uit te voeren:

(Hint: Om snel een matrix van nullen met h rijen en w kolommen te creëren kan je gebruik maken volgend stukje code: nullen = [[0 for j in range(w)] for i in range(h)])

a) Bereken de afgeleide volgens x, door voor elke rij de afgeleide te berekenen en sla dit resultaat op in een nieuwe afbeelding *Ix*.
b) Doe hetzelfde per kolom om zo de afgeleide *Iy* volgens y te berekenen.
c) Je hebt nu 2 afbeeldingen Ix en Iy. De gradiënt voor een pixel op coordinaten (i,j) is nu de vector $\vec{\nabla}\mathrm{f} = (I_x, I_y)$. Bereken voor alle pixels $\left|\vec{\nabla}\mathrm{f}\right|^2 = I_x^2 + I_y^2$ en schrijf dit weg naar een nieuwe afbeelding I_*grad*.
d) Maak nu een nieuwe afbeelding edges die 0 bevat voor elke pixel waarvoor I_grad kleiner is dan een threshold van 0.05 en 1 voor de pixels waarvoor I_grad groter is.

RGB Images

Een RGB kleurenafbeelding kan je beschouwen als een matrix in 3 dimensies. De eerste twee dimensies zijn de hoogte en de breedte van de afbeelding. De derde dimensie zorgt er voor dat we een kleurenafbeelding hebben: elke pixel uit de afbeelding heeft een waarde 0-255 voor ROOD, een waarde 0-255 voor GROEN en een waarde 0-255 voor BLAUW. Zo kan elke pixel dus meer dan 16.000.000 (256x256x256) verschillende kleuren krijgen.



Graphical presentation of RGB 3d matrix

Vraag 4: Overlay & Mask

Open de afbeeldingen I1 = python.bmp en I2 = bugs.bmp. Maak nu een nieuwe afbeelding *I_out* waarin we deze twee afbeeldingen gaan overlappen:

We kiezen een “blending factor” alpha = 0.5 en we creëeren I_out door een gewogen gemiddelde te nemen van de twee input afbeeldingen:

I_out[i][j] = alpha * I1[i][j] + (1-alpha) * I2[i][j]

Toon het resultaat dat je verkrijgt door verschillende waarden voor alpha te gebruiken.

Feitelijk zou het beter zijn als we uit de afbeelding ‘bugs.bmp’ enkel het konijn zouden knippen en dit overzetten naar de andere afbeelding. Dit kunnen we doen door gebruik te maken van een masker. Als masker neem je de afbeelding ‘bugs_masker.bmp’.

Kopieer nu de pixels van *I2* naar *I1* maar enkel als de overeenkomende pixel in *masker* een zwarte kleur heeft.

Vraag 5: De 'toverstaf' uit Photoshop

We gaan een programma maken dat meet hoeveel pixels van een bepaalde kleur er in een afbeelding aanwezig zijn. Dit gebeurt in het algemeen als volgt:

1. Open de afbeelding en lees de pixels in.

2. Open één van de afbeeldingen met homogene kleur en lees de pixels in. Hiermee kan je de te zoeken kleur bepalen (= de pixelwaarde op plaats (0,0)).

3. Definieer een threshold-waarde die bepaalt hoe ver een pixel uit de originele afbeelding van de targetkleur mag afwijken om al dan niet geselecteerd te worden.

4. Maak een nieuwe image die voor elke pixel uit de originele afbeelding aantoont hoe ver deze van de targetkleur verwijderd is. Open hiervoor een nieuwe grijswaarden-afbeelding van dezelfde grootte als de originele afbeelding:

   ```
   newim = [[0 for j in range(w)] for i in range(h)]
   ```

   Om de pixelwaarden van deze nieuwe afbeelding te berekenen gebruik je de volgende regel: 'Hoe dichter de R, de G en de B waarden van de originele pixel bij de targetwaarden liggen, hoe lichter de kleur van de overeenkomstige pixel in de nieuwe afbeelding.' Een witte pixel betekent dus een perfecte match met de targetkleur.

5. Nu gaan we de grijswaarden-afbeelding die we hebben aangemaakt 'thresholden' zodat we enkel witte en zwarte pixels overhouden. Witte pixels betekenen dat er slechts een klein verschil bestond tussen de originele R, G en B waarden en de targetwaarden (dus de grijswaarde valt binnen de threshold). In het andere geval zijn de pixels zwart.

6. Bepaal het kleurpercentage door te kijken naar het aantal witte/zwarte pixels in het bekomen resultaat.

   Test je code door gebruik te maken van afbeelding 'mario.bmp' en targetkleuren 'kleur.bmp' en 'kleur2.bmp'. Varieer de threshold-waarde en vergelijk de bekomen resultaten.

KLEUR