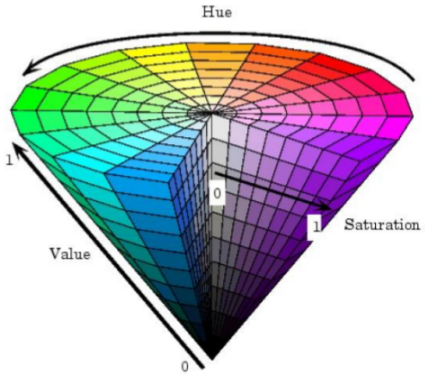
Labo 4: Bewerkingen op kleurenbeelden en detectie van randen en lijnen

Gianni Allebosch, Martin Dimitrievski



Figuur 1. De HSV-kleurenruimte.

1 Bewerkingen op kleurenbeelden

Kleurenbeelden in een **RGB**-formaat bestaan uit drie lagen die met een bepaalde bitdiepte (meestal 8 per kleurlaag) de rode, de groene en de blauwe component voorstellen. Er bestaan ook enkele andere representaties voor kleurenbeelden, waarvan er één veelgebruikte kort zal voorgesteld worden.

De **HSV**-ruimte (*Hue*, *Saturation* en *Value*) is een kleurensysteem waarbij kleuren gekozen worden uit een kleurenwiel of van een palet. Dit systeem staat dichter bij de menselijke ervaring en beschrijving van kleur dan het RGB-systeem. Als de hue varieert van 0 tot 1, dan variëren de kleuren van rood over geel, groen, cyaan, blauw en magenta terug naar rood, dus circulair zoals je kan zien op Figuur 1 (er is dus rood in de buurt van zowel 0 als 1). De saturatie kan ook variëren van 0 tot 1, en de corresponderende kleuren (of hue) variëren mee van ongesatureerd (wit en lichtgrijze schakeringen) tot volledig gesatureerd (geen witte component meer aanwezig). De value of 'brightness' varieert ook van 0 tot 1, en de kleuren worden lichter naarmate de value stijgt. De conversie gebeurt door mapping van cartesische RGB-coördinaten naar cilindrische HSV-coördinaten. OpenCV in Python voorziet de conversie tussen RGB en verschillende kleurenruimtes met de functie `cv2.cvtColor`.

(a) (b)

Figuur 2. Afbeeldingen van verkeerssituaties; (a) `traffic1.jpg` , (b) `traffic2.jpg`

1.1 Opdracht

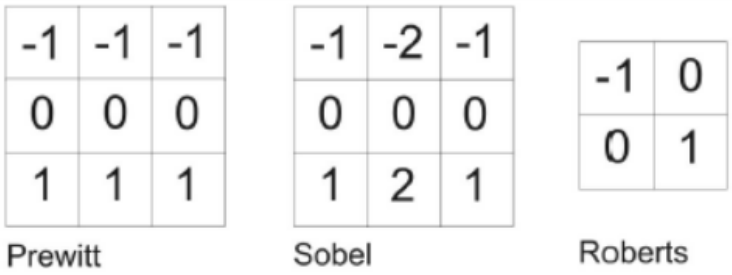
1. **Kleursegmentatie**
   De beelden op Figuur 2 zijn genomen met een digitale scanlijncamera vanuit een rijdende auto, het beeldformaat van de camera is YCbCr, maar is al geconverteerd naar RGB. De opdracht is een kleurenfilter te implementeren om de belangrijkste kleuren van verkeersborden te detecteren: rood, blauw, wit en zwart. Je kan dit in de RGB-ruimte proberen, maar intuïtief kan je al aanvoelen dat dit beter zal gaan in de HSV-ruimte: blauw en rood zijn gemakkelijk te onderscheiden in de hue-component van deze ruimte. Eventueel kan je ook op saturatiewaarden filteren, omdat die vrij hoog is voor de kleuren van een verkeersbord.
   Noem de functie `segment_color`. Leg minimale en maximale drempelwaardes op voor de waardes van de hue/saturatie. Denk er ook aan dat rood zowel rond 0 als 1 voorkomt, dus houd daarmee rekening voor de grenzen van de minimale en maximale hue-waarde. Zoek zelf uit hoe je best voor wit en zwart filtert: welke kleurenruimte, welke laag ... Als output van de filters wordt een zwart/witbeeld (= een logische matrix of masker) verwacht waarin de enen duiden op de aanwezigheid van de gefilterde kleur in het beeld. Beeld deze maskers ook af voor elk gefilterd kleur.

2 Randdetectie

Randen in een beeld worden gekenmerkt door discontinuïteiten in het intensiteitsoppervlak van een beeld. Deze discontinuïteiten kunnen gedetecteerd worden door een eenvoudig spatiaal filtermasker over het beeld te laten lopen. De gewichten in het masker worden zo gekozen dat de randen in een bepaalde richting een hoge respons geven. Bij voldoende hoge respons wordt een pixel dan geclassificeerd als een randpixel. Een voorbeeld van een randenbeeld vind je in Figuur 3.

Enkele detectoren zijn gebaseerd op discrete benaderingen van de eerste afgeleide zoals de randdetectoren van **Roberts**, **Prewitt** en **Sobel**. De maskers

Figuur 3. Voorbeeld van randdetectie; (a) `clown.jpg`, (b) een randenbeeld ervan.



Figuur 4. Enkele veelgebruikte randfilters.

worden met een bepaalde directionaliteit opgesteld: voor horizontale, verticale of diagonale randen. In de Figuur 4 wordt de horizontale versie van de maskers getoond. Door de maskers te draaien kan je ook diagonale en verticale lijnen detecteren. De som van de gewichten in de maskers sommeren tot 0, zodat in gebieden met een constante intensiteit de respons ook 0 is. Sobel en Prewitt zijn de meest gebruikte, waarbij Sobel een betere ruisonderdrukking heeft. **Vraag:** ***Waarom geeft Sobel een betere ruisonderdrukking dan Prewitt?*** Bekijk hiervoor aandachtig het filtermasker in Figuur 4.

Ook de **Laplaciaan** (zie labo 2) valt trouwens onder de categorie van randenfilters. Deze is echter gebaseerd op de tweede afgeleide in een beeld. De **LoG** of ***Laplacian-of-Gaussian*** filter is een randfilter dat eerst het beeld smoother maakt (m.b.v. de Gaussiaan) en dan de Laplaciaan berekent. Daaruit kan de randinformatie bepaald worden als de nuldoorgang van de Laplaciaan. De ***zero-crossings detector*** (nuldoorgangdetector) is gebaseerd op hetzelfde concept als de LoG-methode, maar de convolutie gebeurt met een gespecificeerde filterfunctie. Hoe smoother het beeld wordt, hoe minder detailranden er gevonden worden.

De ***Canny edge detector*** is één van de krachtigste methodes om randen te detecteren. Dit is een algoritme dat tevens *edge linking* uitvoert. De methode kan als volgt beschreven worden:

1. Convolueer een beeld met een Gaussiaan van schaal σ, met σ de standaarddeviatie van de Gaussiaan.
2. Bereken de lokale grootte (en richting) van de gradiënt voor elke pixel (bijvoorbeeld via Sobelmaskers). Hoge waarden duiden op overgangen in intensiteit.
3. Zoek de locatie van randpixels met de hoogste lokale gradiëntmagnitude door lokaal de maxima te bepalen (*non maximum suppression*), deze stap zorgt ervoor dat je een rand van 1 pixel breedte bekomt, i.p.v. een band van randpixels.
4. De overblijvende randpixels worden dan verdeeld volgens 2 drempelwaardes: T_1 en T_2 met $T_1 > T_2$, in twee verzamelingen van sterke ($> T_1$) en zwakke ($> T_2$ en $< T_1$) randpixels. Randpixels met een magnitude boven T_1 worden sowieso als randpixels geclassificeerd. Het linken van randen gebeurt vanuit deze sterke randpixels. Aangrenzende randpixels boven T_2 worden telkens weer toegevoegd tot de magnitude van de volgende randpixel onder T_2 zakt. Deze vorm van hysteresis zorgt ervoor dat ruizige randen niet in meerdere delen uiteenvallen.

Merk hierbij op dat het Canny-algoritme een binair beeld teruggeeft, in tegenstelling tot de andere methoden. Elke pixel wordt bij het Canny-algoritme met andere woorden geclassificeerd als 'rand' of 'geen rand', waar de andere methoden eerder een maat voor de sterkte van de randen teruggeven. Al deze filters kunnen geïmplementeerd worden via `scipy.ndimage.filters` of `cv2` (`cv2.Sobel`, `cv2.Canny` ...). in Python.

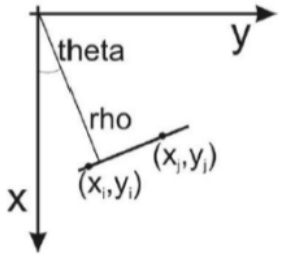
3 Lijndetectie

Eens de randpixels onderscheiden zijn in de originele afbeelding, kunnen we deze samennemen tot volledige lijnen. Hiervoor bestaan verschillende technieken, waarvan we er hieronder 2 bespreken.

3.1 Houghtransformatie

De Houghtransformatie berekent de rechte lijnen in een beeld door te kijken welke lijnen door een punt (x_i, y_i) van het beeld lopen. Er kunnen oneindig veel lijnen $y_i = ax_i + b$ door (x_i, y_i) lopen met telkens verschillende waarden voor a en b. In de parameterruimte ab kunnen al deze lijnen voorgesteld worden door verschillende punten (a, b). Als één lijn, dus met dezelfde parameters a en b, zowel door het punt (x_i, y_i) als (x_j, y_j) loopt, kan je deze lijn weergeven door hetzelfde punt in de parameterruimte (a, b).

Het nadeel hiervan is dat deze lijnvoorstelling geen verticale rechten kan voorstellen (want dan zou $a = \infty$), dus stappen we over naar de representatie



Figuur 5. De (θ, ρ)-parametrisatie van een lijn.

$\rho = x_i \cos\theta + y_i \sin\theta$, waarbij θ de hoek is die de loodrechte op de lijn maakt met de x-as terwijl ρ de afstand tot de lijn uitdrukt (zie Figuur 5). De Houghtransformatie verdeelt de parameterruimte in accumulatorcellen (θ_i, ρ_i) met een bepaalde grootte. De cellen liggen verdeeld tussen minimale en maximale waarden voor θ en ρ ($\theta \in [-90°, 90°]$, terwijl ρ kleiner is in absolute waarde dan de verste afstand van de oorsprong tot de hoeken van het beeld).

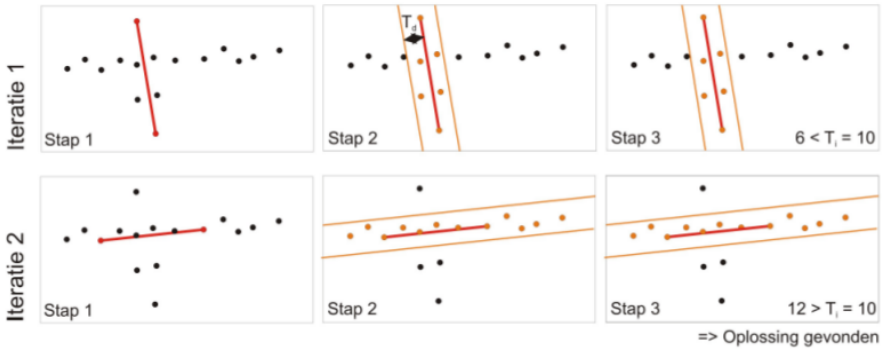
De Houghtransformatie gaat nu voor elke mogelijke combinatie (i, j) van θ en ρ alle mogelijke lijnen in het beeld bekijken. Het aantal randpixels dat op deze lijn (met een bepaalde (θ_i, ρ_i)) ligt in het beeld, wordt opgeteld bij de cel voor die (θ_i, ρ_i) in de parameterruimte. De accumulatorcellen in de parameterruimte waarvoor een hoge waarde bereikt wordt, zullen dan overeenkomen met rechte lijnen in een beeld. OpenCV voorziet de nodige commando's om de Houghtransformatie van een beeld te berekenen: `cv2.HoughLines`, dewelke een lijst van (ρ, θ)-paren (in deze volgorde!) teruggeeft en `cv2.HoughLinesP`, dewelke een lijst van verzamelingen van beeldpunten teruggeeft, waarbij elke verzameling alle punten bevat die tot één lijn behoren.

3.2 RANSAC

Het doel van RANSAC(*RANdom SAmple Consensus*) is een robuuste fit van een model aan een verzameling van punten die outliers bevat. Outliers zijn punten die niet aan het model voldoen, in tegenstelling tot inliers die wel, hetzij binnen bepaalde grenzen, aan het model voldoen.

Hier is het model een lijn die we proberen fitten aan een verzameling van randpixels uit een beeld. Het RANSAC-algoritme kan als volgt beschreven worden (zie ook Figuur 6):

1. Selecteer random een sample van s punten uit de verzameling S van alle punten en construeer een model voor de lijn op basis van deze punten (met $s = 2$ voor een lijn).
2. Zoek de subset S_i van punten die binnen een drempelwaarde T_d van de gemodelleerde lijn liggen. De consensus van de samplepunten en de punten uit S_i zijn de inliers voor het huidige model.
3. Vergelijk het aantal inliers r met drempelwaarde T_i.



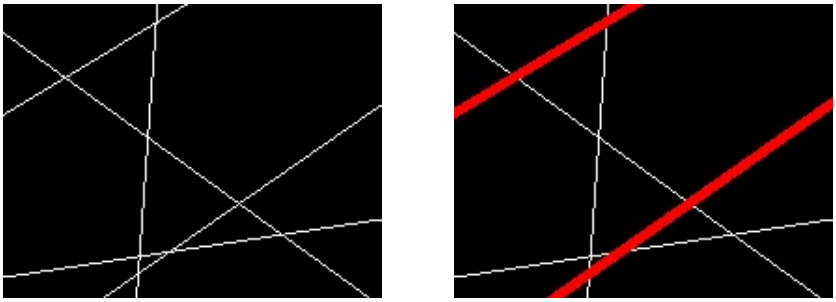
Figuur 6. Illustratie van de verschillende stappen in het RANSAC-algoritme voor lijndetectie: (1) Kies eerst een random sample van $s = 2$ pixels uit de verzameling S van alle randpixels. Stel dan een vergelijking op voor de lijn gevormd door deze 2 punten. (2) Bereken de afstand van alle andere punten tot deze lijn en kijk welke punten op een afstand kleiner dan T_d van deze lijn liggen. (3) Als er genoeg randpixels binnen deze grenzen liggen, kan je deze lijn echt als een volwaardige lijn in het beeld beschouwen, anders moet je een nieuw sample kiezen. Als de lijn voldoet, kan je ze eventueel nog hermodelleren en verfijnen op basis van de informatie uit alle pixels uit de consensusset S_i door bijvoorbeeld *least squares fitting*.

   - Als $r > T_i$, verfijn dan het model met alle punten in S_i , en beëindig het algoritme.
   - Als $r < T_i$, herhaal de bovenstaande stappen 1-3.
4. Als er na het selecteren van N verschillende samples, geen voldoende betrouwbaar model is gevonden, selecteer dan de grootste consensusset S_i en verfijn het model met alle punten in S_i.

Merk op dat RANSAC in de praktijk ook voor veel andere applicaties gebruikt wordt, waarbij outliers kunnen voorkomen. Het basisprincipe blijft telkens hetzelfde: er wordt telkens een model gefit aan een random gekozen, minimale set van punten (of waarnemingen). Als er voldoende andere punten aan dit model voldoen (binnen bepaalde marges), wordt het model behouden en eventueel verder geöptimaliseerd. **Vraag:** ***Wat is de minimale waarde van de s-parameter als je het RANSAC-algoritme zou gebruiken om cirkels in een beeld te detecteren?***

3.3 Opdrachten

2. **Lijndetectie met de Houghtransformatie**
   In `lines.jpg` zie je een binaire afbeelding met een aantal lijnen. Twee lijnen daarvan lopen *bijna* parallel met elkaar. Schrijf een functie `hough` die gebruik maakt van de houghtransformatie om deze twee lijnen te detecteren. Beeld deze vervolgens af in een andere kleur af bovenop de originele afbeelding (zie



(a) (b)

Figuur 7. Voorbeeldoplossing opdracht Houghtransformatie; (a) `lines.jpg` , (b) parallele lijnen aangeduid.

(a) (b)

Figuur 8. Voorbeeldoplossing opdracht RANSAC; (a) `wires.jpg` , (b) gedetecteerde lijnen aangeduid.

Figuur 7 voor een voorbeeld). Je mag ervan uitgaan dat er *exact* 2 parallele lijnen voorkomen in de afbeelding.

Let op: Als je gebruik maakt van `cv2.HoughLines` zul je hiervoor de snijpunten van deze lijnen met de randen van de afbeelding moeten bepalen, of anders een aantal startpunten moeten kiezen die ver genoeg buiten de afbeelding liggen. (zie `https://docs.opencv.org/4.1.1/d9/db0/tutorial_hough_lines.html`) Werk je met `cv2.HoughLinesP`, dan moet je de oriëntatie van de lijnen controleren door zelf hun vergelijking op te stellen. Voor het tekenen van de gekleurde lijnen mag je de originele afbeelding eerst omzetten naar een kleurenbeeld en vervolgens de functie `cv2.line` gebruiken.

3. **Lijndetectie met RANSAC**

   Implementeer het bovenstaande RANSAC-algoritme voor lijndetectie en noem deze functie `ransac`. Test dit algoritme op het beeld `wires.jpg`, waarbij we enkel de lange elektrische draden wensen te detecteren. Beeld net zoals bij de vorige opdracht de gedetecteerde lijnen af op de originele afbeelding, maar

gebruik deze keer voor elke gedetecteerde lijn een andere kleur (zie Figuur 8).

Enkele praktische tips voor de implementatie in Python:

- Voer de bovenstaande beschrijving van het algoritme evenveel keer uit als het aantal lijnen dat gezocht wordt (bv. in een for-lus).
- Het RANSAC-algoritme moet op het randenbeeld uitgevoerd worden: kies een methode zoals bv. Canny en bereken het randenbeeld. De coördinaten van de randpixels kan je bijvoorbeeld bekomen met `numpy.nonzero`. De indices worden teruggegeven als vectors waarin respectievelijk de rij- en kolomcoördinaten van de randpixels staan.
- Random getallen kan je genereren met de module `numpy.random`. Deze random getallen kan je als index in de rij- en kolomcoördinaten gebruiken. Let er wel op dat je niet 2 dezelfde punten kiest.
- Stel dat een lijn door 2 punten (x_1, y_1) en (x_2, x_2) gegeven is. Om de afstand van die lijn tot een punt (x_k, y_k) te bepalen, kun je volgende formule gebruiken:

$$d = \frac{|(x_1 - x_2)(y_2 - y_k) - (x_2 - x_k)(y_1 - y_2)|}{\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}}$$

  Denk eraan dat je in Python bewerkingen ook op vector- en matrixniveau kunt uitvoeren.
- Punten die al tot een lijn behoren kan je beter uit de verzameling randpixels verwijderen voordat je het algoritme opnieuw uitvoert om een volgende lijn te detecteren.
- Gebruik een schatting voor de parameters T_d en T_i op basis van de informatie over het beeld dat je gebruikt. Bv. hoe lang moet een lijn in het beeld zijn t.o.v. de grootte van het beeld voor ze aanvaardbaar is? (Je mag dit schatten voor dit specifieke inputbeeld).
- De verfijning van de gevonden puntenset gebeurt meestal door lineaire regressie en het *least squares algoritme*. Voor dit labo is het echter niet nodig dat jullie dit implementeren. Weet wel dat er verschillende Pythonimplementaties bestaan die dit algoritme uitvoeren (zie bijvoorbeeld `cv2.Solve` en `numpy.linalg`).