Rapport de Stage de 2A

Détection de panneaux routiers

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Thomas Forgione
2INE

Tuteur : M. Ramon Morros

16 Juin 2014 — 26 Juillet 2014

Introduction

Ce projet a pour objectif de développer un programme capable de détecter, voire reconnaître les panneaux de signalisation sur une photo. Les applications de ce projet sont multiples : en équipant les voitures de caméra et de ce programme, on sera en mesure non seulement de proposer une aide au conducteur (il n'aura plus besoin de chercher les panneaux puisque ceux-ci s'afficheront sur un écran sur le tableau de bord), mais on pourrait aussi envisager un pilote automatique de voiture, qui aurait donc besoin de connaître les règles particulières aux endroits dans lesquels il roule, et donc de détecter et reconnaître les panneaux.

Évidemment, la principale difficulté à laquelle nous allons devoir faire face est la capacité que devra avoir notre algorithme à fonctionner dans n'importes quelles conditions, notamment météorologiques (brouillard, pluie, obscurité, forte luminosité ...).

Comme d'habitude en vision par odinateur, nous allons effectuer une chaîne de transformations qui conduirons petit à petit au résultat. Chaque étape sera décrite par une section dans ce rapport.

Remarque Il faudra garder à l'esprit que le but est d'avoir un algorithme qui soit relativement rapide, dans la mesure où il sera utilisé en temps réel. Trouver des panneaux sur une photo prise il y a une minute n'a aucun intérêt, et le choix des méthodes sur chaque étape prendra en compte son temps d'exécution.

Évaluation des algorithmes

Pour évaluer notre algorithme, une base de 669 photos a été fournie par le maître de stage. Ces photos sont accompagnées de données :

— un masque solution, c'est-à-dire une image binaire où les pixels correspondant aux panneaux sont à 1 et les autres à 0.
— une annotation, c'est-à-dire un fichier texte décrivant la position des panneaux, leur taille, et leur type.

Ces données nous permettrons de tester notre algorithme dans de nombreux cas. En effet, cette base d'image contient des images où les panneaux sont bien visibles, mais aussi des images où les panneaux sont moins visibles (forte inclinaison par rapport à la camera, luminosité trop faible, luminosité trop élevée, panneaux détérioriés, elle contient même certains panneaux qu'un homme aurait du mal à remarquer !).

Nous considérerons les éléments suivants :

— les *vrais positifs* (notés TP) : les endroits où l'algorithme dit qu'il y a un panneau, et qu'il y en a effectivement un.
— les *faux positifs* (notés TN) : les endroits où l'algorithme dit qu'il n'y a pas de panneaux, et qu'il n'y en a effectivement pas.
— les *faux positifs* (notés FP) : les endroits où l'algorithme croit qu'il y a un panneau, mais qu'il n'y en a en fait pas.
— les *faux négatifs* (notés FN) : les endroits où l'algorithme croit qu'il n'y a pas de panneaux mais qu'il y en a en fait un.

Nous considérerons ensuite :

— La *précision* : $TP/(TP + FP)$.
— L'*exactitude (accuracy)* : $(TP + TN)/(TP + TN + FP + FN)$.
— La *spécificité* : $TN/(TN + FP)$
— La *sensibilité* : $TP/(TP + FN)$.

Ces valeurs, ainsi que la durée d'exécution de chaque fonction nous permettrons comparer les différentes méthodes que nous allons tester, afin de choisir les meilleures, et de connaître leurs avantages et inconvéniants.

Les résultats des tests des meilleurs fonctions seront donnés en annexes.

Table des matières

1. Segmentation par couleur

La première étape de cette chaîne de détection et de reconnaissance de panneaux de signalisation est le seuillage de l'image de manière très simple. En effet, les panneaux sont souvent très colorés, la plupart du temps, ils sont rouge ou bleu, et on peut commencer par avoir une idée de la probabilité qu'un pixel appartiennent à un panneau en seuillant ses couleurs de manière intelligente.

L'objectif de cette étape est donc d'avoir une fonction qui prenne en paramètre une image RGB, et qui nous renvoie une image binaire dont les pixels appartenant aux panneaux sont blancs, et ceux n'y appartenant pas sont noirs.

1.1. Seuillage statique

Dans un premier temps, on va, pour seuiller les images, seuiller des fonctions élémentaires sur les images (composantes, ou bien rapports de composantes)

1.1.1. Première idée

La première idée que l'on peut naturellement avoir, est de dire que si un pixel est suffisament rouge ou bleu, alors, il a beaucoup de chance d'appartenir à un panneau. On commence notre seuillage de cette façon :

```
function binary = Threshold1(im)

threshold_high_value = 50;
threshold_low_value = 80;

binary = ...
             ... % Detect the red color ...
            (im(:,:,1) > threshold_high_value ...
            & im(:,:,2) < threshold_low_value ...
            & im(:,:,3) < threshold_low_value) ...
       | ...
           ... % Detect the blue color ...
           (im(:,:,1) < threshold_low_value ...
           & im(:,:,2) < threshold_low_value ...
           & im(:,:,3) > threshold_high_value);

end
```

La critique que l'on peut faire avec cet algorithme est qu'il dépend fortement de la luminosité de l'image : s'il fait nuit, et que l'éclairage est faible, un pixel à $(40, 1, 1)$ devra être considéré comme rouge, mais notre algorithme ne fonctionnera pas sur ce pixel. En voici un exemple :

(a) Image originale (b) Seuillage

FIGURE 1 – Exemple de panneau non détecté à cause de la luminosité

1.1.2. Rapports de canaux

En étudiant le cas du pixel $(40, 1, 1)$, on voit que ce n'est pas la couleur du pixel qu'il faut étudier, mais bel et bien le rapport entre les canaux. Au lieu de seuiller les composantes R, G, et B, on va maintenant tenter de seuiller $\frac{R}{G+B}$ pour détecter le rouge, et $\frac{B}{R+G}$ pour détecter le bleu. On a alors l'algorithme suivant :

```
function binary = Threshold9(im)

im = double(im);

red = im(:,:,1)./(im(:,:,2) + im(:,:,3));
blue = im(:,:,3)./(im(:,:,2) + im(:,:,1));

binary = red > 0.9 | blue > 0.9;

end
```

(a) Image originale (b) Seuillage

FIGURE 2 – Seuillage suivant les rapports de canaux

1.1.3. Amélioration

Après de nombreux tests, j'ai remarqué que la couleur bleue des panneaux ne correspondaient pas parfaitement à la couleur bleue $(0, 0, 255)$. C'est pourquoi le seuillage sur les pixels rouges fonctionne nettement mieux que le seuillage bleu. J'ai donc testé plusieurs techniques, et celle qui donne le meilleur résultat est la suivante :

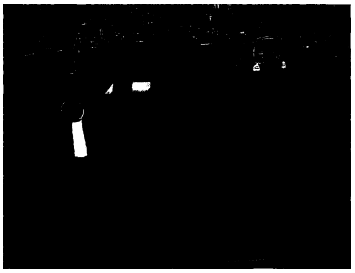
```
function [red, blue] = Threshold93(im, v1, v2)

% Compute the red threshold
red = im(:,:,1)./(im(:,:,2) + im(:,:,3)) > v1;

% Compute the blue threshold
blue = im(:,:,3) ./ im(:,:,1) > v2 & im(:,:,3) ./ im(:,:,2) > v2;

end
```

On va seuiller $\frac{R}{G+B}$ pour le rouge comme prévu, mais pour le bleu, nous allons considérer seulement les pixels ayant un bon rapport $\frac{B}{R}$ et $\frac{B}{G}$.[1] Elle donne ces résultats

1. Les constantes choisies pour seuiller ont étés choisies au départ comme permettant de détecter le plus de panneaux possibles, tout en évitant de répondre "tous les pixels sont susceptibles d'appartenir à un panneau", ce qui ne nous avancerait pas beaucoup.

(a) Image originale

(b) Seuil bleu



(c) Seuil rouge

Remarque On a pris le parti de séparer le seuillage bleu et rouge, pour se faciliter la tâche par la suite. Les algorithmes seront exécutés sur chacun des seuillages, ce qui sera plus long mais offira de meilleurs résultats.

1.1.4. Seuillage dynamique

On a remarqué que dans certaines images, des maisons rouges étaient présentes, ce qui avait pour résultat une image polluée par de nombreux faux positifs. Une première version de seuillage dynamique a été implémentée ainsi.

```
function [red, blue] = Threshold93D(im, v1, v2)

% Compute the red threshold
red = im(:,:,1)./(im(:,:,2) + im(:,:,3)) > v1;

% Compute the blue threshold
blue = im(:,:,3) ./ im(:,:,1) > v2 & im(:,:,3) ./ im(:,:,2) > v2;

% Dynamic threshold
if sum(red(:)) + sum(blue(:)) > numel(red(:))/25
    [red,blue] = Threshold93(im,v1+0.1,v2+0.1);
end;

end
```

Cependant, on remarquera que cette méthode mélange les couleurs rouges et bleues. En effet, si un mur rouge est decté, le seuil bleu va être augmenté, et la détection des panneaux bleus va être dégradée. On a donc fait une deuxième version du seuillage dynamique ainsi.

```
function [red, blue] = Threshold94D(im, v1, v2)

% Compute the thresholds
red = Red(im, v1);
blue = Blue(im, v2);

end

function red = Red(im, v1)
% Compute the red threshold of the image

    red = im(:,:,1)./(im(:,:,2) + im(:,:,3)) > v1;

    % If there is more than 5% pixels that are white,
    % increase the threshold
    if sum(red(:)) / numel(red(:)) > 0.05
        red = Red(im,v1+0.1);
```

```
    end
end

function blue = Blue(im,v2)
% Compute the blue threshold of the image
    blue = im(:,:,3) ./ im(:,:,1) > v2 & im(:,:,3) ./ im(:,:,2) > v2;

    % If there is more than 5% pixels that are white,
    % increase the threshold
    if sum(blue(:)) / numel(blue(:)) > 0.05
        blue = Blue(im, v2+0.1);
    end
end
```

Voici une comparaison des résultats du seuillage statique et du seuillage dynamique sur un exemple bien choisi.

(a) Image originale (b) Seuil rouge statique (c) Seuil rouge dynamique

FIGURE 4 – Avec la méthode `Threshold93D.m` / `Threshold94D.m`

Les résultats sont assez explicites, la quantité de faux positifs a été diminuée fortement grâce au seuillage dynamique.

1.1.5. Méthodes basées sur d'autres représentations des images

Une des idées suggérées par le maître de stage était de travailler dans d'autres espaces de couleur. J'ai donc essayé de passer dans l'espace HSV pour pouvoir seuiller la teinte des pixels sans prendre en compte la luminosité. On a donc essayé le code suivant :

```
function binary = Threshold10(im)

hsv = colorspace(RGB->HSV, im);

binary = (hsv(:,:,1) < 230 & hsv(:,:,1) > 200) ... % Detect blue color
       | (hsv(:,:,1) < 30 | hsv(:,:,3) > 330);     % Detect red color

end
```

Le problème de cette approche est que certains pixels correspondant à la route sur l'image ont parfois une teinte plus proche du bleu que certains panneaux un peu de biais et à l'ombre. On se retrouve donc face à un dilemme : soit on considère que les pixels du sol appartiennent à un panneau, et on aura une précision très faible, soit on ne les considère pas, et on va manquer quelques panneaux.

1.2. Seuillage de loi normale

Une méthode parallèle au seuillage brut de la couleur est de calculer la moyenne et la variance de chaque canal des panneaux d'une certaine catégorie, de modéliser ses couleurs par une loi normale,

et d'estimer la probabilité d'un pixel d'appartenir à un panneau comme

$$\frac{1}{\sigma\sqrt{2\pi}}e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2}$$

On a donc trois pseudo-probabilités (une par canal) que l'on peut combiner pour calculer un indicateur d'appartenance à un panneau. Seuiller cet indicateur est une façon de répondre au problème de cette étape.

Cependant, le temps de calcul de l'exponentielle étant relativement élevé, et les résultats n'étant pas nettement meilleurs qu'un seuillage brut, cette méthode n'a pas été retenue.

2. Fenêtre glissante

Remarque importante

Juste avant cette étape, nous allons utiliser la fonction matlab `imfill`, avec le paramètre `'holes'` qui permet de remplir les trous dans une image. Comme dit précédemment, nous allons appliquer cette fonction à chacune de nos images seuillées (rouge et bleue), et ensuite y appliquer l'algorithme des fenêtres glissantes. Cela nous permettra d'avoir des résultats nettement meilleurs puisque nous pourrons détecter aussi l'intérieur des panneaux d'interdiction (circulaires à bord rouge).

2.1. Principe

Pour cette deuxième étape, nous avons donc en entrée une image binaire représentant les pixels susceptibles d'appartenir à un panneau. Évidemment, cette image sera certainement bruitée, et il nous faut donc repérer les zones où de nombreux pixels sont blancs.

L'idée est de dessiner des carrés, de tailles variables, qui vont glisser sur l'image. Si dans l'un de ces carrés, les pixels blancs représentent plus d'un certain pourcentage, alors on va considérer que ce carré est probablement un panneau.

2.2. L'over-lapping

Un des problèmes de cette méthode est que lorsqu'un carré sera proche d'un panneau, il contiendra beaucoup de pixels blancs, et sera donc considéré comme un panneau. Au voisinage des panneaux, nous nous retrouverons donc avec plusieurs fenêtres, légèrement décalées les unes par rapport aux autres. Nous allons donc utiliser une fonction qui aura pour but de fusionner les carrés se superposant trop.

La première étape sera de supprimer les carrés qui sont inclus dans d'autres carrés (puisque l'on a des carrés de différentes tailles, il est probable que certains petits carrés se retrouvent dans des carrés plus grands). La deuxième étape consistera à faire fusionner les carrés qui se superposent suffisament.

FIGURE 5 – Illustration du problème d'over-lapping

2.3. Algorithme dérivé

2.3.1. Inconvéniants de l'algorithme précédent

Le problème de cette approche est que si l'on veut avoir une bonne fenêtre, il faut les autoriser à partir de taux de remplissage très faible. Pour le montrer, il suffit de considérer le cas d'un panneau triangulaire dans un carré. Dans le meilleur cas, on aura une figure comme celle ci :

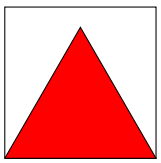


FIGURE 6 – En rouge, la partie détectée par le seuillage, et en noir, le bord de la fenêtre

Si le carré a pour côté a (et le triangle aussi), l'aire du triangle valant $\frac{\sqrt{3}}{4}a^2$, et celle du carré valant a^2, le rapport entre les deux vaut $\frac{\sqrt{3}}{4} \approx 43.3\%$ Cela signifie qu'il faut accepter au moins toutes les fenêtres dont le taux de remplissage dépasse les 43.3%, sinon, on risque de manquer des bonnes fenêtres.

2.3.2. Autre idée

L'autre idée est de chercher les panneaux comme étant des agrégations de plus petites fenêtres dont le taux de remplissage sera nettement plus important.

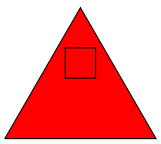


FIGURE 7 – Taux de remplissage de 100%

La technique de fusion des fenêtres sera donc nettement plus sensible : il faudra fusionner les fenêtres qui se chevauchent se serait-ce qu'un petit peu. C'est cependant ce qui donne les meilleurs résultats.

Un autre avantage de cette technique est qu'un très faible nombre de tailles différentes possibles pour les fenêtres est suffisant pour avoir de bons résultats. Il suffit d'avoir deux tailles relativement faibles pour détecter de nombreux panneaux, alors que la technique précédente nécessite d'avoir toutes les tailles de panneaux possibles (ou un peu moins si l'on accepte de baisser encore un peu plus le taux de rempissage, mais le nombre de faux positifs va alors croître).

2.4. Fenêtres intelligentes

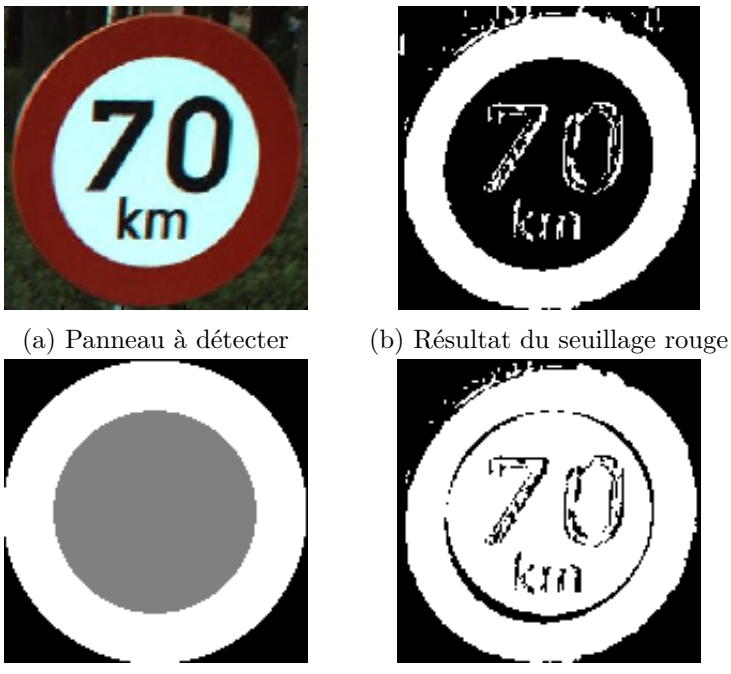
Une autre idée est celle d'utiliser des fenêtres intelligentes. Plutôt que de chercher un carré contenant beaucoup de pixels blancs, on va chercher une forme géométrique dans l'image en utilisant une fenêtre avec une certaine forme. Cette fenêtre sera donc une image (carrée) qui nous indiquera pour chaque pixel si :

— il devrait être allumé (pixel correspondant au rouge d'un panneau par exemple)
— il devrait être éteint (pixel correspondant au blanc d'un panneau par exemple)
— il n'a pas d'importance (pixel à l'extérieur du panneau par exemple)

Remarque Dans ce cas, on n'utilisera pas `imfill`, puisqu'on se servira du noir à l'intérieur du panneau pour le détecter.

On pourra ainsi calculer un coefficient de remplissage en calculant le rapport entre la somme du nombre de pixels blancs qui devraient être blancs et du nombre de pixels noirs qui devraient être noirs, et le nombre de pixels utiles dans la fenêtre.

Voyons ce fonctionnement avec un exemple. La fenêtre représente les pixels inutiles en noir, les pixels supposés être blancs en blanc, et les pixels supposés être noirs en gris.



(a) Panneau à détecter (b) Résultat du seuillage rouge

(c) Fenêtre intelligente (d) Application de la fenêtre

FIGURE 8 – Illustration du fonctionnement des fenêtres intelligentes

Cet algorithme présente tout de même quelques inconvéniants : il est notamment très lent (puisqu'il est nécessaire de tester toutes les formes, et il faut aussi remarquer que si la quantité de pixels supposés être noirs dans le masque est grande, on risque d'avoir de nombreux faux-positifs puisque dans une zone où aucun pixel n'est supposé appartenir à un panneau, on aura une grande quantité de pixels qui seront corrects par rapport à la fenêtre intelligente.

3. Recherche de formes particulières

Maintenant que nous disposons d'un nombre de fenêtres important, il nous faut chercher à éliminer les faux positifs. On va donc parcourir les fenêtres que nous avons trouvées, et chercher un critère permettant d'éliminer les fenêtres qui ne correspondent sûrement pas à des panneaux.

Remarque Puisque le nombre de fenêtres dont nous disposons représente une portion de l'image relativement faible, on va pouvoir utiliser des techniques plus performantes, qui coûtent un peu plus cher en temps d'exécution, puisqu'elles seront appellées sur des portions d'images.

Une idée est de chercher à détecter des formes dans ces fenêtres, et de tenter de détecter des triangles, rectangles, ou cercles. Pour faire ça, on va s'appuyer sur des transformées de Hough.

3.1. Recherche de triangles / carrés

3.1.1. Recherche de segments

La première étape pour trouver des triangles ou des carrés, est de chercher des segments. Pour cela, matlab propose des fonctions qui effectuent des transformées de Hough de sorte à trouver des segments (et d'ailleurs, la documentation y est très bien faite).

On va donc chercher une dizaine de segments que ces fonctions vont nous suggérer. Elles retourneront des `struct array` dont les attributs `point1` et `point2` pourrons nous permettre de retrouver ces segments.

(a) Panneau à détecter (b) Segments détectés

FIGURE 9 – Détection des segments

Une fois ces segments trouvés, on va essayer de fusionner les points qui ne sont pas trop loins les uns des autres : il est probable que la détection des segments soit de mauvais qualité, il faut donc s'attendre à ce que les segments détectant les bords des panneaux ne se touchent pas parfaitement.

La méthode qui a été choisie est de séparer l'information géométrique de l'information topologiue. On va construire un tableau contenant la localisation des points, et pour chaque segment, on testera la proximité de ses extrémités avec les points qui sont déjà dans le tableau. Si la distance est faible, on considèrera que les deux points sont les mêmes, sinon on ajoutera le point courant au tableau. On construira ensuite un tableau de couple d'entiers (c'est-à-dire une matrice de taille $2 \times n$) où deux indices sur la même colonne signifieront que ces indices forment un segment. On se retrouve donc à l'arrivée avec un graphe : ces nœuds et ces arcs.

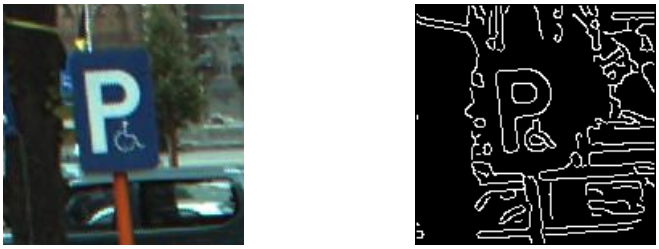
3.1.2. Recherche des triangles

Remarque : il est important de noter que la plupart des panneaux triangulaires sont soit des panneaux blancs a bords rouges, soit des panneaux carrés bleus contenant un triangle blanc et un pictogramme. Dans tous les cas, il y aura un fort contraste entre les couleurs aux frontières du triangles, ce qui ne serait pas le cas si, par exemple, nous avions un panneau rouge sur fond rouge.

Pour trouver les triangles dans une image, on va commencer par chercher les chemins de longueur 2 (c'est-à-dire contenant 3 points) dans le graphe. On va donc calculer les trois angles de ce triangle, et si chacun vaut 60° à 10° près, on retiendra le triangle. Si l'on fait l'hypothèse que les fenêtres sont bien placées, on peut supposer que dans une fenêtre, il n'y aura qu'un seul panneau triangulaire, son aire sera grande et le triangle sera vers le milieu de la fenêtre. En conséquence, on ne conservera que le triangle qui a la meilleure note (on pourra par exemple noter les triangles avec une combinaison linéaire entre l'aire et distance du centre de gravité du triangle au centre de la fenetre). On pourrait aussi supprimer les triangles qui n'ont aucun côté horizontal.

3.1.3. Recherche des rectangles

Contrairement au cas précédent, les panneaux rectangulaires n'ont pas de bords bien marqués, et par conséquent, la détection des segments qui sont les contours du panneaux est nettement moins efficaces. On peut facilement comprendre pourquoi en regardant ce cas particulier d'un panneau de stationnement :



(a) Image originale (b) Image de contours

FIGURE 10 – Cas difficile

On voit sur l'image des contours (calculée avec le filtre de canny) que les bords du panneau ne sont pas visibles. À partir de là, la fonction de détection des segments de matlab ne va rien donner de bon, et une méthode comme celle précédente ne fonctionnera pas. Elle a été quand même implémentée (dans la mesure où l'on s'est rendu compte de ce problème après les tests). Plusieurs techniques ont été essayées :

— recherche des segments parallèles et création du quadrilatère
— recherche des segments perpendiculaires et création du quadrilatère

Cependant, toutes ces méthodes se sont avérées infructueuses puisque dans la mesure où les bords des panneaux sont mal détectés, il a fallut être particulièrement généreux dans l'acceptation de ce que l'on considère comme étant des segments. À partir de là, un bruit trop important fait des quadrilatères sont détectés n'importe où et à chaque fois qu'on a une fenêtre quelque part, on peut être sûr que

cette méthode arrivera à trouver un quadrilatère. Celà ne va donc guère nous aider pour la suite. Une idée pour palier à ce problème est d'utiliser un détecteur de contours plus fin (CF UCM 4.3), mais ceci n'a pas été implémenté.

3.2. Recherche des cercles

Une fonction de détection de cercles basée sur une transformation de Hough a été fournie par le maître de stage. Elle a donc été utilisée sur une image de contours lissée (puisqu'un cercle vu en perspective n'est plus vraiment un cercle, il faut donc laisser un peu de marge à cette détection pour avoir de bons résultats). Elle fonctionne bien dans de nombreux cas, mais parfois, donne des résultats pour le moins étrange.

4. Problème principal rencontré

4.1. Problème du seuillage

Le principal problème rencontré dans le cadre de ce projet est la détection de certains panneaux bleus à l'ombre. L'image posant le plus de problème et représentant un bon exemple de cette difficulté est la suivante

FIGURE 11 – Cas difficile

Comme vous pouvez le constater, le panneau est très sombre, et sa couleur est la même que la portion de route à l'ombre. Par conséquent, une analyse fondée simplement sur la couleur de chaque pixel ne pourra pas donner un bon résultat.

4.2. Une proposition de solution

Une solution est de travailler sur une image de contours plutôt qu'une image renseignant sur les couleurs. L'idée était de suivre ces étapes :

1. Calculer une image de contours de l'image
2. Calculer une carte de distance des contours
3. Faire glisser une fenêtre (cette fois-ci avec une forme : triangulaire ou circulaire...)

On conservera les fenêtres qui minimisent la somme des pixels dans la carte des distances, l'idée étant que puisque un panneau est fermé, les pixels à l'intérieur du panneau seront proches des bords et donc de valeur faible dans la carte des distances.

Le résultat attendu est le suivant :

(a) Image originale

(b) Image de contours

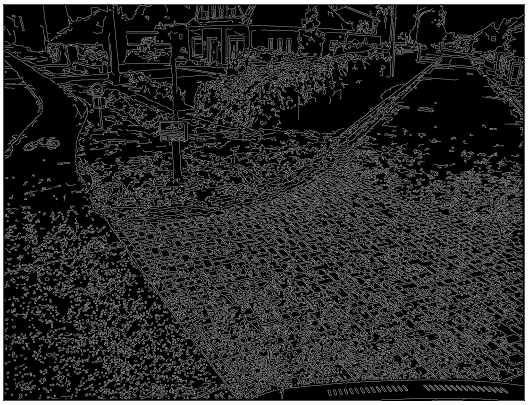
(c) Carte de distance

(d) Detection

FIGURE 12 – Cas attendu

Le problème est qu'en pratique, l'image des contours est de nettement moins bonne qualité. En voici un exemple

(a) Image originale



(b) Image de contours

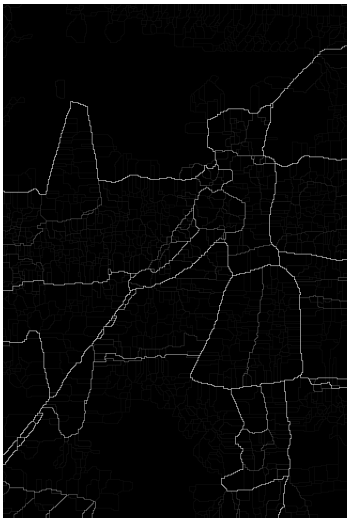
FIGURE 13 – Problème de détection des contours

Comme vous pouvez le constater sur la figure 13, l'image des contours montre la texture du sol (bords des pavés) et on voit à peine les contours du panneau. Il est clair qu'ici, la moitié inférieure de l'image aura une carte des distances très faibles, et on n'arrivera donc pas à trouver le panneau.

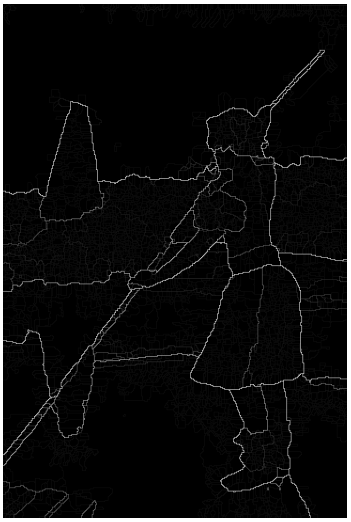
4.3. Le détecteur de contours UCM

Une idée donnée par le maître de stage est d'utiliser une technique nettement plus efficace dans la détection des contours. Voici un exemple des résultats produits par cette technique.

(a) Image originale



(b) Contours rapides



(c) Contours précis

FIGURE 14 – Détéction efficace des contours

Cette fonction renvoie une image, deux fois plus grande que l'originale (puisqu'un contour n'est pas sur un pixel mais entre deux pixels), et la valeur de chaque pixel correspond à l'intensité du contours. On peut donc l'utiliser pour éliminer les contours inintéressants de manière nettement plus efficace qu'avec un simple détecteur de canny.

Cette solution n'a pas été implementée par manque de temps.

5. Conclusion

Ce projet m'a permis d'une part, de me familiariser encore plus avec le langage matlab, et de revoir les techniques utilisées en traitement d'image et en vision par ordinateur. Il a aussi été un exemple très parlant des difficultés couramment rencontrées en vision par ordinateur, à savoir la recherche d'un algorithme qui soit efficace dans toutes les conditions. La vision par ordinateur en informatique est un thème plein de contraintes, et qui peut se révéler assez frustrant puisqu'il semble impossible d'avoir un algorithme qui offre de très bons résultats.

À côté de celà, il reste plusieurs choses possibles à faire sur ce projet

— utiliser le détecteur de contours UCM pour détecter les panneaux manqués, et éventuellement combiner les techniques orientées couleurs et contours
— corriger les problèmes des détections de rectangles et cercles
— combiner cet algorithme à un programme capable de prendre des photos à des moments bien choisis, voire de faire du suivi de panneaux
— mettre en place le système de reconnaissance des motifs à l'intérieur des panneaux, en sachant qu'on dispose déjà d'indications sur la forme des panneaux (triangles, cercles...)

A. Résultats du seuillage

Remarque ces résultats ont été calculés après le passage par `imfill`.

Technique	Temps moyen par image	Precision	Exactitude	Spécificité	Sensibilité
`Threshold93`	0.1465s	15.6851%	98.4325%	98.5236%	73.9049%
`Threshold94`	0.1627s	8.0297%	96.5064%	96.5651%	80.6966%

Le `threshold93` est le seuillage dynamique qui modifie simultanément le seuillage rouge et bleu (même valeur pour le seuil). Le `threshold94` est celui qui traite séparément le rouge et le bleu.

B. Résultats pour les fenêtres et formes

Nous obtenons ces résultats avec un temps moyen d'exécution de 2.0394s par image.

Formes	Precision	Exactitude	Sensibilité
Fenêtres	31.9978%	26.1959%	59.0956%
Triangles	62.5551%	40.3409%	53.1835%
Cercles	12.6706%	8.6782%	21.5947%

C. Images résultats

Les images résultats sont disponibles sur ce lien. Les boîtes rouges représentent les fenêtres élargies (on les a élargi pour être sûrs que l'intégralité du panneau soit dans la fenêtre). Les cercles rouges, et les triangles cyans représentent les formes détectées. Les rectangles cyans sont les plus petits rectangles contenant les formes détectées (pour les triangles, on les a encore élargi puisque l'on suppose avoir détecté la frontière intérieure du panneau).