Rapport Analyse d'image

Université Claude Bernard Lyon 1

Théo Bénard - p2007461
Théo Grillon - p1907354

Informatique graphique

Reconnaissance de formes avec la transformée de Hough

---

1 Introduction

Ce projet s'inscrit dans le cadre de l'UE de master 1 informatique *Mif17 - Analyse d'images*. Ce dernier se focalise principalement sur l'analyse de contours d'une image en utilisant la transformée de Hough comme outil principal. Ce rapport permet de présenter notre travail. Il est aussi ponctué de retour réflexif sur notre travail ainsi que la justification des choix techniques.

2 Transformée de Hough

La transformée de Hough est un outil permettant de reconnaître des formes simples, comme des droites et des cercles, dans le plan de l'image qui peut être décrit par un ensemble de n caractéristiques. Un cercle peut être décrit par 3 caractéristiques: la position de son centre (x, y) et son rayon r. Cette transformée nécessite en entrée une image binarisée contenant les contours présents dans l'image. Le paramètre d'entrée peut être calculé en fonction du gradient de l'image si celui-ci est initialement inconnu. L'algorithme donne en sortie sous la forme d'un accumulateur n dimensions (avec n = nombre de caractéristiques) l'occurrence de toutes les formes possibles dans l'image pour un type donné.

L'algorithme général de la transformée de Hough procède de la manière suivante:

Algorithm 1 Transformée de Hough (naif)

Entrée: image binaire
Sortie: accumulateur des occurences des formes paramétriques
Créer un accumulateur $\boldsymbol{\alpha}$ à n dimensions
Pour chaque pixel **p** de l'image étant un contour:
  Pour chaque case **c** de $\boldsymbol{\alpha}$:
    Considérons le n-uplet qui caractérise la forme **f** en **c**
    Si p est intersecté par **f alors** incrémenter **c** de un
  Fin Pour
Fin Pour
retourner $\boldsymbol{\alpha}$

Par la suite, nous verrons que cet algorithme peut être optimisé en faisant intervenir, par exemple, les directions précédemment obtenues lors du calcul du gradient de l'image.

La complexité en espace et en temps est d'autant plus importante que le nombre de caractéristiques de la forme l'est.

2.1 Hough Lines

Dans un premier temps, nous nous sommes intéressé à la détection de lignes. La transformée de Hough permet de détecter des droites sous la forme $(\boldsymbol{\theta}, \boldsymbol{\rho})$, correspondant aux coordonnées polaire de la droite dans le plan.

Pour rappel, il existe une relation entre le système cartésien et polaire:

$$\mathbf{a}x + \mathbf{b}y + \mathbf{c} \Rightarrow x\cos(\boldsymbol{\theta}) + y\sin(\boldsymbol{\theta}) = \boldsymbol{\rho} \qquad (1)$$

L'utilisation du système polaire permet de réduire le nombre de caractéristiques, donc, de dimensions de l'accumulateur et ainsi de réduire la complexité (d'un ordre de grandeur d'une puissance) comparée à un système cartésien (3 caractéristiques).

Pour la détection de ces lignes, on parcourt chaque pixel de l'image, puis pour chaque pixel **p** correspondant à un contour, on itère sur θ de $-\frac{\pi}{2}$ à π, permettant d'obtenir l'ensemble des droites possibles, en calculant à chaque fois la valeur de ρ, comprise entre 0 et $\sqrt{\text{width}^2 + \text{height}^2}$:

$$\boldsymbol{\rho} = \mathbf{p.x}\cos(\boldsymbol{\theta}) + \mathbf{p.y}\sin(\boldsymbol{\theta}) \qquad (2)$$

On obtient ainsi les paramètres de la droite qui intersecte **p**. On incrémente l'accumulateur de 1 aux indices $\boldsymbol{\rho}$ et $\boldsymbol{\theta}$ calculés.

2.2 Recherche de maximums locaux

Lors de l'application de l'algorithme des Hough pour la détection des lignes, le nombre de droites détecté est trop important. Le but est de faire un tri dans l'ensemble de ces droites et de garder les occurrences

les plus pertinentes et de regrouper les droites qui sont très similaires entre elles.

Nous avons opté pour l'utilisation d'un algorithme de recherche de maximums locaux dans l'accumulateur afin de regrouper les droites paramétriquement proches, c'est-à-dire, ayant un angle $\boldsymbol{\theta}$ et une distance $\boldsymbol{\rho}$ proche.

2.2.1 En quoi ça consiste ?

L'algorithme de recherche de maximums locaux consiste principalement à calculer le barycentre de chaque région détectée dans l'accumulateur. Une région est déterminé par un agglomérat de valeur supérieure à un certain seuil $\boldsymbol{\sigma}_1$. Un second seuil $\boldsymbol{\sigma}_2$ est appliqué lors de l'ajout des voisins. Ce dernier permet de diviser une région en sous-région et ainsi prendre plus de droites. Le barycentre est représenté par un point, avec deux valeurs : $\boldsymbol{x}$ et $\boldsymbol{y}$ correspondant respectivement à la distance $\boldsymbol{\rho}$ et à l'angle $\boldsymbol{\theta}$.

Cet algorithme retourne un vecteur de lignes. Une structure `Line` à été définie avec les trois attributs suivants :

- un point qui stocke la position correspondante dans l'accumulateur
- un flottant `theta` pour l'angle
- un flottant `rho` pour la distance

L'algorithme ci-dessus réalise un scan de l'accumulateur en recherche des points au-dessus d'un certain seuil $\boldsymbol{\sigma}_1$. Lorsque l'algorithme trouve une droite avec un nombre d'intersections suffisant, il va essayer de regrouper le plus de voisins possible dans une région. Le second seuil $\boldsymbol{\sigma}_2$ détermine ce regroupement. Plus $\boldsymbol{\sigma}_2$ est petit, plus les lignes de fortes proximitées paramétriques seront ajoutées à la même région.

Le barycentre est calculé au fur et à mesure que l'on parcourt la région. Une fois cette dernière entièrement parcourue, on calcule $\boldsymbol{\theta}$ et $\boldsymbol{\rho}$ et on stocke, via la structure `Line`, les informations sur la

Algorithm 2 Recherche de maximums locaux : **getLines**

Entrée : (accumulateur $\boldsymbol{\alpha}$, seuil de détection $\boldsymbol{\sigma}_1$, seuil de regroupement $\boldsymbol{\sigma}_2$)
Sortie : vecteur de lignes (`Line`) `v`
Début :

```
temp ← copier α dans un temporaire
v ← initialiser v
Pour chaque α_{ρ,θ} ≥ σ_1 :
  # initialiser une pile
  pile ← {ρ, θ}
  # initialiser le barycentre
  centre ← {0, 0}
  # initialiser un compteur
  cmpt ← 0
  Tant que la pile n'est pas vide :
    # récupérer l'élément en tête de pile
    (x, y) ← pile.top()
    # retirer l'élément en tête de pile
    pile.pop()
    # incrémenter les coordonnées du barycentre
    centre.x ← centre.x + x
    centre.y ← centre.y + y
    # appel à la procédure colorPixelRegion
    colorPixelRegion(temp, pile, σ_2, x, y)
    # incrémenter le compteur
    cmpt ← cmpt + 1
  Fin Tant que
  centre ← centre / cmpt
  # définir les attributs de la droite
  line.params ← (centre.x, centre.y)
  line.theta ← radians(centre.y)
  # ici MAX_RHO est une variable globale
  line.rho ← centre.x - MAX_RHO
  v.push(line)
Fin Pour
Retourner v
```

Fin

Algorithm 3 Procédure d'ajout des voisins dans la pile : **colorPixelRegion**

Entrée : ($\boldsymbol{\alpha}$, `pile`, $\boldsymbol{\sigma}$, `x`, `y`)
Sortie : ne retourne rien
Début :

```
# mettre à 0 l'élément (x,y) dans l'accumulateur
α(x,y) ← 0
# voisins en 4-connexités
Pour chaque voisin ν de α_{x,y}
  # si la valeur du voisin est supérieur au seuil
  Si ν ≥ σ Faire :
    α(ν_x,ν_y) ← 0
    # ajouter sa position dans la pile
    pile ← {ν_x, ν_y}
  Fin Si ν ≥ σ
Fin
```

droite dans le vecteur retourné par la fonction.

L'algorithme se termine car à chaque ajout de voisin, celui-ci est effacé dans l'accumulateur temporaire. Cela permet d'éviter qu'une droite appartienne à plusieurs régions.

Comme pour la détection de ligne, une recherche des maximums locaux est également effectuée pour la détection de cercle, suivant le même principe mais avec en entrée une matrice à trois dimensions.

2.2.2 Premiers résultat

Pour les premiers résultats, en utilisant notre implémentation du gradient, nous obtenons ceci :

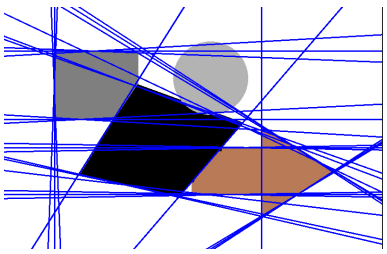

Figure 1: Détection de lignes avec notre gradient

L'implémentation de notre gradient n'étant pas parfaite et l'affinage des contours manquants, beaucoup trop de lignes sont détectées dans l'image. En comparaison, avec l'utilisation de la détection de contour fournie par OpenCV, le nombre de lignes détectées est plus précis même si certaines lignes, comme pour le cas précédent, restes manquantes :

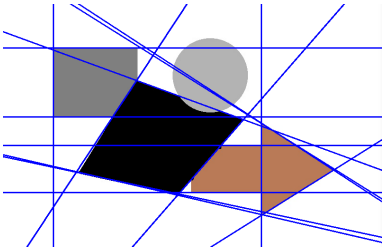

Figure 2: Détection de lignes avec la détection de contours d'OpenCV

2.2.3 Utilisation de la direction du gradient

Dans le cas où le gradient de l'image est connu, celui-ci permet de connaître directement la tangente de la droite au point $\mathbf{p}$ en utilisant la direction du gradient avec $\boldsymbol{\theta} = atan2(g_y, g_x)$. Cela permet d'éviter une itération sur le domaine de θ.

Dans le cas de l'utilisation du gradient en multidirectionnel, les angles de recherches sont limités à 0, $\frac{\pi}{4}$, $\frac{\pi}{2}$ et $\frac{3\pi}{4}$, ce qui implique que certaines lignes ne peuvent pas être détectées correctement :

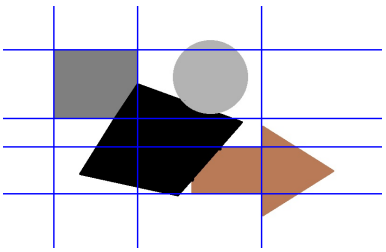

Figure 3: Utilisation de la direction du gradient en multidirectionnel

Dans le cas du bidirectionnel, notre implémentation ne doit pas être tout à fait correcte puisque le

résultat obtenu n'est pas satisfaisant :

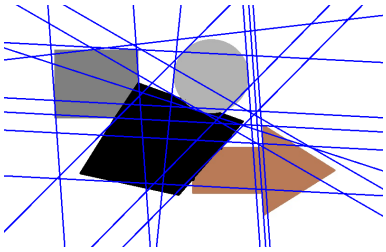

Figure 4: Utilisation de la direction du gradient en bidirectionnel

La recherche de maximums locaux permet d'avoir un meilleur contrôle sur le nombre de lignes détecté dans l'image. Notre implémentation du gradient n'est pas complète, il manque l'affinage des contours, rendant son utilisation moins efficace que l'implémentation d'OpenCV.

L'utilisation des directions du gradient peut être intéressante dans certains cas, par exemple si l'on souhaite détecter une grille dans une image, l'utilisation des directions du gradient en multidirectionnel peut permettre de réduire le temps de calcul en obtenant un résultat satisfaisant. Cependant, il semble y avoir une erreur dans l'utilisation des directions en bidirectionnelle puisque les lignes détectées ne correspondent pas vraiment au résultat souhaité.

2.3 Hough Circles

Un cercle peut-être défini par deux composantes simples : son centre **c** définit par ses coordonnées (**a**, **b**) et son rayon **r**. L'équation ci-dessous nous permet de calculer directement **r** en fonction des coordonnées du pixel considéré et des coordonnées de **c** :

$$(x - \mathbf{c}_x)^2 + (y - \mathbf{c}_y)^2 = \mathbf{r}^2$$
$$\Rightarrow \sqrt{(x - \mathbf{c}_x)^2 + (y - \mathbf{c}_y)^2} = \mathbf{r}$$

Pour détecter un cercle dans l'image, comme pour la détection de ligne, on parcourt chacun des pixels. Pour chaque pixel **p** correspondant à un contour, on cherche tous les cercles qui intersecte **p**. Le problème avec cette approche c'est que c'est extrêmement couteux en temps de calcul. En effet, pour chaque pixel **p**, on parcourt une seconde fois l'image. Dans ce second parcours, on calcule le rayon correspondant au cercle qui intersecte **p** en utilisant l'équation paramétrique d'un cercle définit précédemment avec (x, y) les coordonnées de **p** et **c** les coordonnées de l'image dans le second parcours.

Une optimisation peut être réalisée lorsque le calcul du gradient a été effectué auparavant. L'étape du gradient nous donne l'information sur la direction du contour au point **p**. On peut tracer une droite suivant cette direction. Elle représente les positions possibles pour les cercles en intersection avec **p** sachant la direction du gradient en ce point. Cette optimisation permet de gagner un facteur dix sur le temps d'exécution.

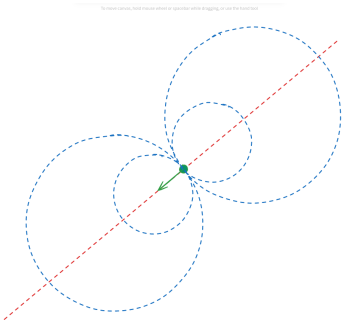

Figure 5: Cercles possibles en fonction de la direction du gradient en un point

Sachant ceci, il suffit de parcourir cette droite dans l'accumulateur. En effet, l'accumulateur à trois dimensions : les deux premières (**a**, **b**) correspondent à la position du centre du cercle dans l'image d'entrée et la troisième correspond au rayon **r**. Parcourir cette droite dans l'accumulateur revient à la parcourir dans l'image. On incrémente donc toutes les cases présentent sur cette droite. Le parcours s'arrête lorsque la case considérée se trouve hors de l'image.

2.3.1 Résultats

En utilisant le gradient, sa direction et en prenant uniquement les cercles ayant un nombre de votes supérieurs à 40%, 50% puis 90% du maximum global, dans le cas multidirectionnel, on obtient ceci :

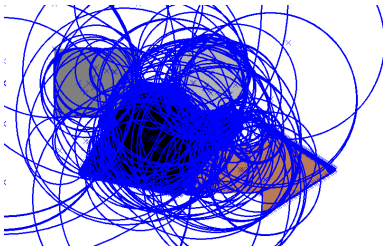

Figure 6: Détection de cercle en multidirectionnel à 40% du maximum

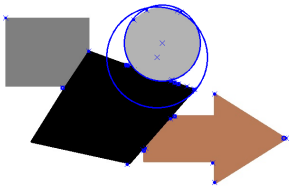

Figure 7: Détection de cercle en multidirectionnel à 50% du maximum

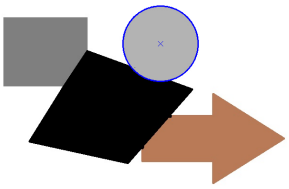

Figure 8: Détection de cercle en multidirectionnel à 90% du maximum

Le gradient bidirectionnel est plus intéressant que celui en multidirectionnel car les directions calculées sont plus précises, et donc le cercle que l'on doit détecter dans l'image aura plus de vote que les autres cercles détectés car les intersections seront plus nombreuses :

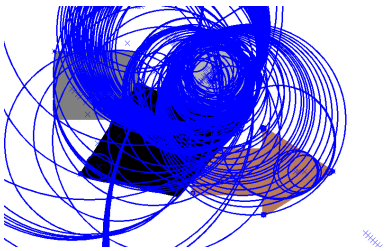

Figure 9: Détection de cercle en bidirectionnel à 40% du maximum

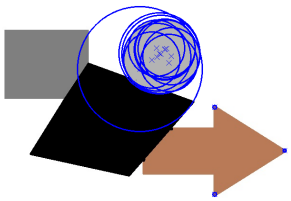

Figure 10: Détection de cercle en bidirectionnel à 50% du maximum

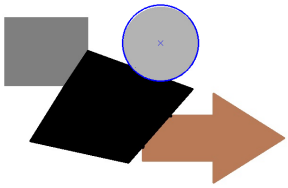

Figure 11: Détection de cercle en bidirectionnel à 60% du maximum

On peut voir qu'en prenant uniquement les cercles ayant un nombre de votes équivalant à 60% du maximum global, il ne reste que le cercle que l'on souhaite détecter là où en multidirectionnel il fallait

aller jusqu'à 90%.

Nous arrivons à un résultat similaire si l'on ne fait pas usage de la direction du gradient mais le temps de calcul est beaucoup plus important, supérieur à un facteur dix (2500 ms avec direction contre 30000 ms sans direction) :

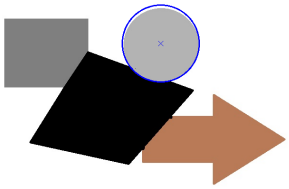

Figure 12: Détection de cercle sans utilisation des directions

3 Observations

3.1 Multithreading

L'algorithme de la transformée de Hough que nous avons implémenté dans le cas des cercles peut-être très lents. En utilisant les directions données par le gradient, on à en moyenne, quatre secondes d'exécution avant l'affichage de l'image résultat. En revanche, lorsque l'on ne les utilise pas, le temps de calcul augmente de manière exponentielle. Il est donc préférable d'utiliser le gradient dans le cas de la déctection de cercle. Nous avons tenté d'utiliser du multithreading pour ce cas spécifique pour améliorer les performances.

L'implémentation du multithreading est basique, n-thread se partage une partie de l'image. Nous cherchons à optimiser le calcul de l'accumulateur de cercle qui est en 3 dimensions. On associe chaque case du tableau avec un mutex dans un tableau séparé de même dimension. Si un thread cherche à incrémenter la case de l'accumulateur il doit alors prendre le mutex, s'il ne peut pas, il doit l'attendre.

Nous observons des performances à la baisse comparées à l'algorithme initial. L'espace mémoire consommé est gigantesque, jusqu'à 4Go de RAM pour l'image de la cathédrale pour contenir l'accumulateur et les mutexes. Nous avons décidé néanmoins de laisser le code disponible dans un fichier séparé.

3.2 Méthodes

Nous avons obtenu des résultats plus ou moins bons en fonction de la méthode utilisée. Voici les avantages / inconvénients de chaque méthode :

3.2.1 Méthode sans gradient

- **Avantages**
  - dans le cas d'une image déjà binarisé, application directe de la transformée de Hough
  - peut dans certains cas avoir de meilleur contours que par l'utilisation du gradient et permet donc d'obtenir un meilleur résultat
- **Inconvénient**
  - cas de la détection de cercle trop coûteux en temps

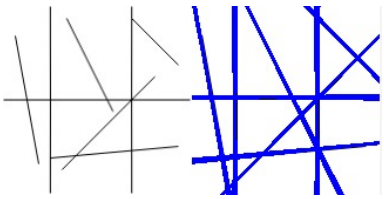

Figure 13: Détection de lignes sans gradient

3.2.2 Méthode avec gradient sans utiliser les directions

Si le gradient est simplement utilisé pour avoir les contours dans l'image et que les informations supplémentaires qu'il fournit ne sont pas utilisées, alors cela

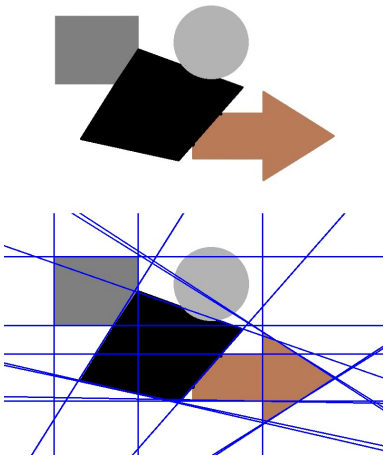

Figure 14: Détection de lignes sans gradient 2

reviens relativement au même que l'image binarisée si ce n'est que le résultat des contours sera différent ce qui peut impacter la qualité du résultat final.

3.2.3 Méthode avec gradient en utilisant les directions

En bidirectionnel

- **Avantages**
  - forme circulaire plus facilement détecté du fait de la convergence d'un plus grand nombre de direction en un point
  - temps de calcul fortement réduit pour la détection de cercle
- **Inconvénients**
  - résultat peu satisfaisant pour la détection de ligne
  - les directions sembles trop peu précisent

En multidirectionnel

- **Avantages**
  - bonne détection des lignes verticales/horizontales

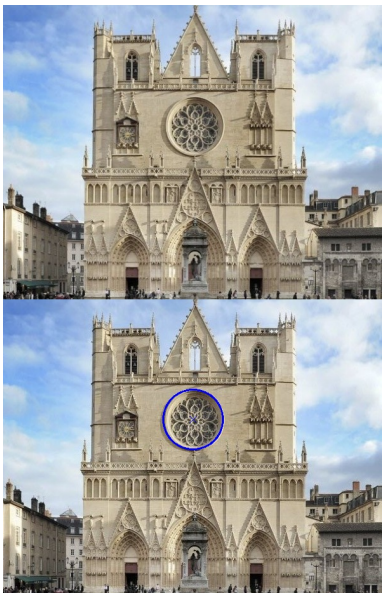

Figure 15: Détection de cercle en bidirectionnel

  - résultat de la détection de cercle correcte
  - temps de calcul fortement réduit pour la détection de cercle
- **Inconvénient**
  - nombre de directions limité

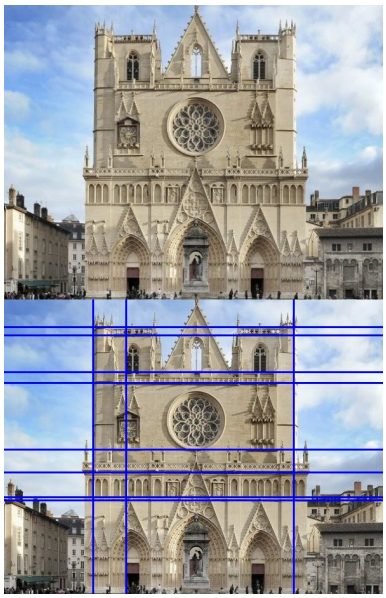

Figure 16: Détection de ligne en multidirectionnel

4 Conclusion

La transformée de Hough que nous avons implémenté dans le cadre de ce TP donne des résultats plus ou moins cohérents en fonction de la méthode utilisée. Des améliorations peuvent être apportées comme, par exemple, effectuer une pondération des votes en fonction de l'amplitude du gradient. Une découpe de l'image en plusieurs images sur lesquelles l'algorithme serait appliqué de manière indépendante pourrait être réalisé afin de mieux détecter certaines lignes. Par exemple, sur l'image de la cathédrale, les lignes correspondant aux bâtiments sur le côté ne sont pas détectées ou difficilement. Avec un calcul local du gradient sur une de ces portions de l'image, les contours de ces bâtiments pourraient être mieux détectés et ainsi avoir un meilleur résultat pour la transformée de Hough.