

# Compte rendu TP Traitement d'image

Vasilis SKARLEAS, Rami ARIDI

## Partie 1 - Détection des voies

### Objectifs

Dans cette première partie, on souhaite détecter les voies de circulation à gauche et à droit. Ici on travaille sur un vidéo de la roue, mais le même principe peut être appliqué aux vidéos capture par les voitures autonomes pour détecter les voies de gauche et droit respectivement.

On essaye de ne pas se baser sur une manière de résolution qui est spécifique pour cette vidéo, mais d'appliquer des méthodes universelles ou on peut appliquer dans différents formats de vidéo tout en incluant des outils qui permettra de régler les seuils des différents filtres et méthodes de traitement d'image qui sont appliqués et qu'on va voir en détail sur la partie ci-dessous (par exemple les Trackbars nous permettent de changer les paramètres de base de notre application).

### Méthodes / Choix

	Approche 1	Approche 2
<b>Méthode</b>	Mettre chaque frame sur l'espace HSV et distinguer les voies selon la différence crucial de couleur entre les voies et les verts autour de la route.	Appliquer la transforme de Hough car les voies sont caractérisées par des lignes droites.
<b>Avantages</b>	Une bonne séparation entre les voies et la pelouse. C'est une méthode simple à mettre en œuvre, car elle ne nécessite pas de transformations mathématiques complexes.	Méthode plus universelle car il ne dépend pas à la différence des couleurs. C'est une méthode robuste en ce qui concerne le bruit et la différence d'intensité.
<b>Inconvénients</b>	Dépend sur la différence des couleurs qui est spécifique pour les frames en question. En plus, cette méthode est très sensible aux variations d'éclairage, aux ombres et aux autres objets colorés présents dans l'image.	Ce méthode ne peut pas être appliquée aux cas des voies tournantes si on ne fait pas un raisonnement.

Dans notre résolution, on a choisi de procéder avec la méthode de la transformation de Hough car elle peut être adaptée à différents types de marquages au sol et à différentes conditions routières. En outre, la transformée de Hough est plus robuste face aux variations d'éclairage et au bruit, quelque chose qui le rende universelle selon les cas d'application différents.

### Comment on appliqué ?

La vidéo est constituée d'une série d'images fixes, qui sont affichées les unes après les autres à une vitesse très rapide (30 FPS dans notre cas).

```
Camera::Camera()
{
    m_fps = 30;
}
```

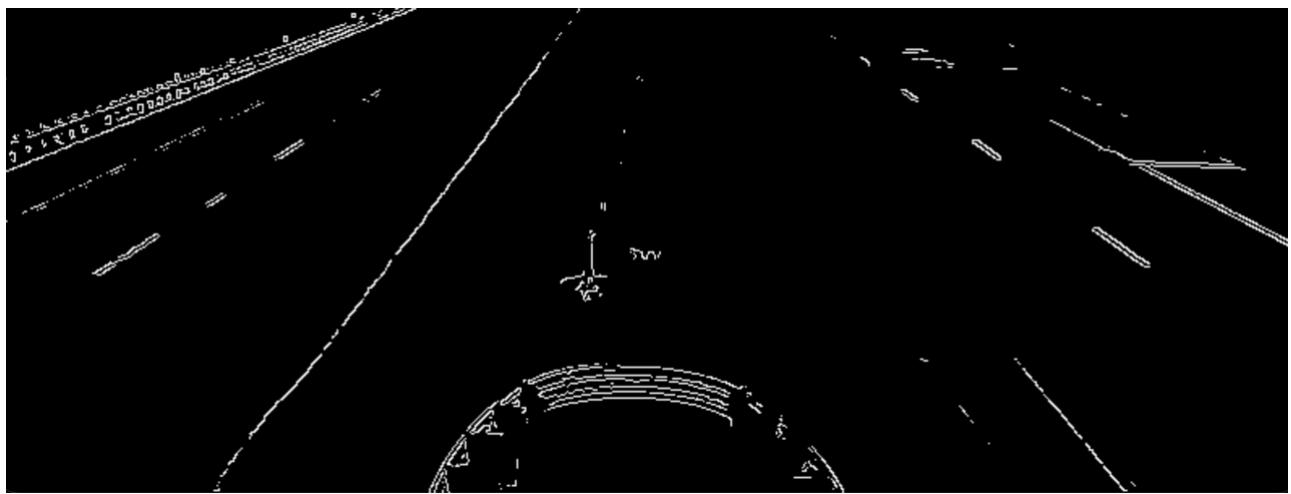
Donc, on peut appliquer toutes les différentes méthodes du cours sur chaque image qui compose la vidéo et avoir des résultats en temps réels si on considère la vitesse de la réalisation de calcul. À partir de ce moment, chaque fois qu'on est référencé à une image, en fait on est référencé à un frame de la vidéo.

## Étapes

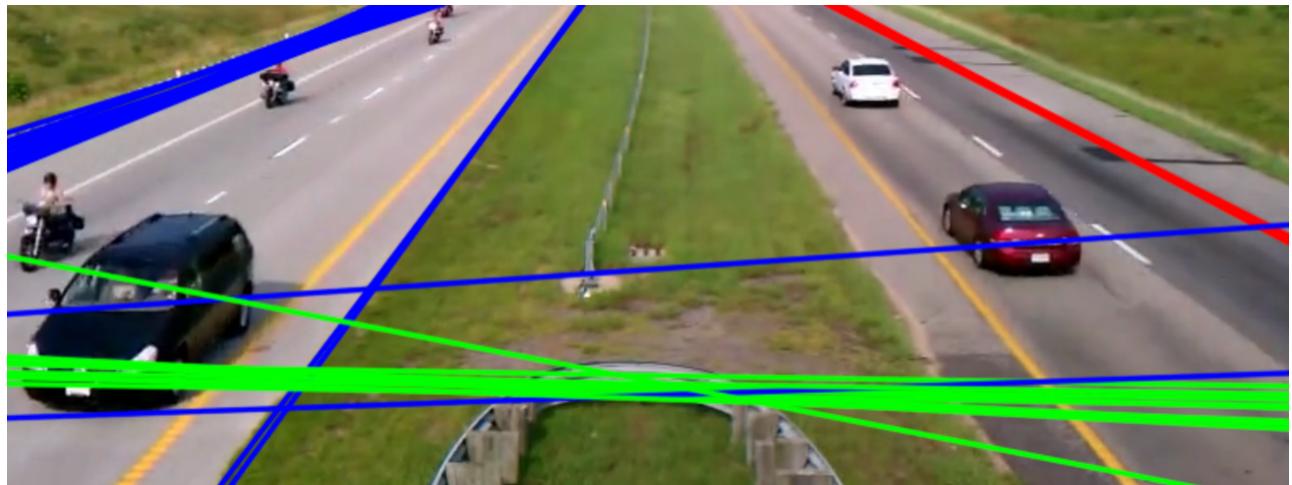
1. Obtenir l'image en niveau de gris (**gray**)



2. Détection des contours (**edges**).



3. Application de la transformée de Hough sur **edges** et sauvegarder les retours de la transformée de Hough sur un vecteur de dimension 2.



1.  $\rho$  : qui est à distance par le pixel principal
2.  $\theta$  : l'angle de la ligne selon le pixel en question
4. Trier les lignes sauvegardé
5. Fusionner les lignes qui sont très proches (`rho_threshold`) avec une différence d'inclinaison acceptable (`theta_threshold`). Voici une image après cette procédure:



## Applications

### Détection des contours

On a besoin de trouver les contours sur l'image car la transformée de Hough est applicable seulement sur une liste des contours. Forcément pour détecter les contours, il faut avoir l'image en niveau de gris. Il y a plusieurs méthodes d'obtenir les contours comme l'approche Laplacien ou l'approche gradient. Ici, on a procédé avec la méthode de Canny qu'on n'a pas vu en cours mais il est sensé d'être très efficace par rapport le calcul des points de contours.

### Transformée de Hough

Sur la librairie OpenCV, il a y a deux fonctions qui permet de récupérer les lignes de Hough :

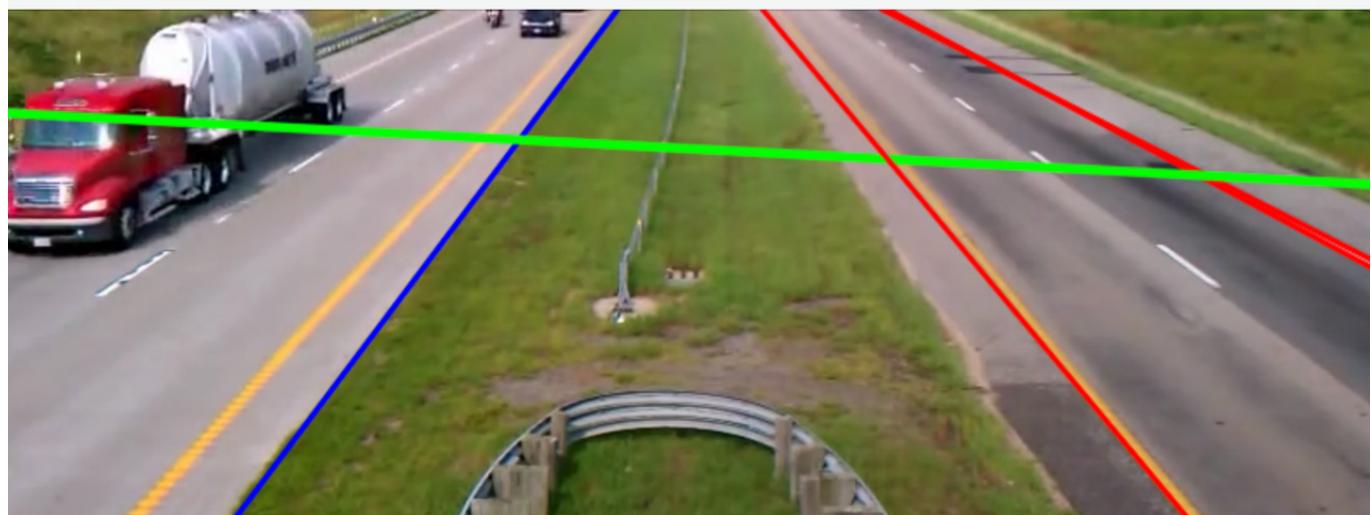
- `HoughLines` retourne le  $\rho$  et le  $\theta$
- `HoughLinesP` retourne les coordonnées de deux points qui constituent une ligne

```
cv::HoughLines(edges, lignes, 1, CV_PI / 180, thres_hough);
```

Nous on a utilisé **HoughLines** car on veut faire le tri qu'on va détailler ci-dessous

#### Trier les lignes de Hough

Une fois qu'on a sauvegardé toutes les lignes détectées par la transformé de Hough, il faut distinguer des différents cas et garder que les lignes qui sont en norme. Par exemple, dans l'image ci-dessous, la ligne verte ne peut pas être accepté car l'angle  $\phi$  est beaucoup plus grand que celui pour les lignes de voies.



Les différents tests qu'on effectue sont :

1. Enlever les lignes de Hough qui ont un angle inférieur à `thres_hough_theta` choisi pendant l'analyse

```
// Tri No 1
for (size_t j = 0; j < lignes.size();)
{
    int angle = lignes[j][1] * 180 / CV_PI; // recuperer l'angle en
    // degré car il est donnée en radians par HoughLines()
    if ((angle > thres_hough_theta) && (angle < 180 -
    thres_hough_theta))
    {
        lignes.erase(lignes.begin() + j); // Enlève l'élément à
        // l'indice j
    }
    else // passer à la prochaine
    {
        j++;
    }
}
```

2. Enlever les lignes restants qui n'ont pas la bonne inclinaison au correct partie de l'image.



Dans l'image ci-dessus, la ligne verte n'est pas acceptée même si elle est dans les normes de `thres_hough_theta` car elle est localisée à la fausse partie de l'image.

```
// Tri No 2
for (size_t i = 0; i < lignes.size() - 1; i++) // ATTENTION: Without
the -1, there was a segmentation Fault in Macos
{
    float rho = lignes[i][0];
    float theta = lignes[i][1];
    double a = cos(theta), b = sin(theta);

    /* Calcul des coordonnées cartésiennes des points qui compose une
ligne.
C'est la projection d'un vecteur ligne sur le repère de base */
    double x0 = a * rho;
    double y0 = b * rho;

    /* Calcul du point le plus bas sur la ligne */
    // Le point (x0, y0) s'agit d'un point le plus proche de l'origine
(selon l'équation de Hough).
    // Le point pt_low est choisi en s'éloignant de ce point le long
de la ligne dans une sens (vers le bas dans ce cas-là).*/
    cv::Point pt_low(cvRound(x0 + 1000 * (-b)), cvRound(y0 + 1000 *
(a)));
}

/* Corps du tri No 2 */
if (pt_low.x > hough_final.cols / 2 && (theta * 180 / CV_PI) < 90)
// À droite
{
    lignes.erase(lignes.begin() + i); // On enlève l'élément à
l'indice i
}
else
{
    i++; // On incrémente seulement si on n'a pas effacé l'élément
}
if (pt_low.x < hough_final.cols / 2 && (theta * 180 / CV_PI) > 90)
```

```
// À gauche
{
    lignes.erase(lignes.begin() + i); // On enlève l'élément à
l'indice i
}
else
{
    i++;
}
}
```

#### Fusionner les lignes

S'il y a plusieurs lignes qui sont très proche (+- rho\_threshold) avec un angle similaire (+- theta\_threshold), on veut faire une fusion et considérer une seule ligne dans ce cas.

```
lignes = merge_lignes(lignes, rho_threshold, theta_threshold); // On merge
les lignes

//Voici l'application de l'algorithme de fusion
// Si les lignes de Hough sont très proches l'un à l'autre, on veut garder
qu'une seule commune pour chaque groupe de lignes
// Remarque 1.2
std::vector<cv::Vec2f> merge_lignes(const std::vector<cv::Vec2f> &lines,
float rho_threshold, float theta_threshold)
{
    std::vector<cv::Vec2f> merged_lines;

    std::vector<bool> merged(lines.size(), false);

    for (size_t i = 0; i < lines.size(); ++i)
    {
        if (merged[i])
            continue; // Skip lines that have already been merged

        float rho1 = lines[i][0];
        float theta1 = lines[i][1];

        float rho_sum = rho1;
        float theta_sum = theta1;
        int count = 1;

        // Check for other lines that are close to the current one
        for (size_t j = i + 1; j < lines.size(); ++j)
        {
            float rho2 = lines[j][0];
            float theta2 = lines[j][1];

            // If the lines are close in rho and theta, consider them the
same and merge
            if (std::abs(rho1 - rho2) < rho_threshold && std::abs(theta1 -
```

```

theta2) < theta_threshold)
{
    rho_sum += rho2;
    theta_sum += theta2;
    count++;
    merged[j] = true; // Mark this line as merged
}
}

// Average the rho and theta values to create the merged line
float avg_rho = rho_sum / count;
float avg_theta = theta_sum / count;

merged_lines.push_back(cv::Vec2f(avg_rho, avg_theta));
}

return merged_lines;
}

```

Cette application parcourt les lignes détectées et fusionne celles qui sont proches l'une de l'autre en calculant leur moyenne. Cela permet de réduire le nombre de lignes détectées et d'améliorer la précision de la détection.

### Améliorations

Une fois que cette démarche était mis en complet sur le programme de lecture des images, on voudrait trouver une manière de isoler le calcul de lignes de transformation de Hough que pour le 10 premiers images, et afficher les lignes final dans le cours de la vidéo. Comme ça, une fois que les camions passent, on n'aura pas une perturbation sur les lignes et par extension avec cette méthode, les lignes des voies vont être stable au cours de la vidéo.

C'est pourquoi sur la logique de la boucle `while(isReading)` il y a un compteur des frames (`frame_id`) pendant laquelle on fait la fussion des lignes. Le vecteur final après les 10 iterations est toujours affiché via :

```

if (frame_id >= 10)
{
    for (const auto &line : fixed_lines)
    {
        afficher_lignes({line}, frame_with_fixed_lines);
    }
}

```

### Remarque sur la transformée de Hough

À partir les paramètres  $\theta$  et  $\rho$ , on peut calculer des points de la ligne en s'éloignant de point d'origine le long de la ligne dans les deux sens (haut et bas). Avant faire ça, il faut calculer le point d'origine qu'il suffit forcement d'une simple projection de la ligne de Hough sur le système cartésien  $x$  et  $y$  via:

```

float rho = lignes[i][0];
float theta = lignes[i][1];

double a = cos(theta);
double b = sin(theta);

/* Calcul des coordonnées cartésiennes des points qui compose une ligne */
double x0 = a * rho;
double y0 = b * rho;

```

Donc pour le calcul de deux points qui permettent de designer la ligne à la fin sur l'image on a:

```

cv::Point pt1(cvRound(x0 + 1000 * (-b)), cvRound(y0 + 1000 * (a)));
cv::Point pt2(cvRound(x0 - 1000 * (-b)), cvRound(y0 - 1000 * (a)));

```

## Partie 2 - Suivi des véhicules

En utilisant les techniques données dans le cours, ici on veut effectuer le suivi des véhicules au cours de la vidéo. La question principale est comment faire car il y a plusieurs manières de procéder ? Ci-dessous on va analyser seulement notre processus de résolution de la problématique introduit.

### Comment on a fait ?

Il faut faire un traitement d'image adapté qui est capable de détecter les objets qui sont en mouvement. La fonction `process_frame(frame_with_fixed_lines, edges, voiture_total_gauche, voiture_total_droite);` va inclure tous les étapes du traitement.

### Étapes

1. Substraction du frame du maintenant avec le frame précédent. Ca va nous donner une image avec seulement les objets qui sont en mouvement (image **Movement**).
2. Obtenir l'image Movement en niveau de gris
3. Segmentation/binarisation de l'image en utilisant **threshold**
4. On fait une fermeture pour fermer les trous, si possible, à l'image **edges** qui est l'image **Movement binarisé**. On obtiens l'image final **closed**.
5. On trouver les contours via **findContours** pour calculer leurs surfaces. Comme ça, on peut trier quels contours on veut garder au pas [même processus pour la plaque d'immatriculation qu'on a vu en TD machine]
6. Designer le rectangle autour de voitures
7. On applique l'algorithme de compter les voitures

### Applications

#### **Image Movement**

Sauvegarder le frame en copie pour réaliser la substraction avec le prochain frame. Comme ça on peut obtenir l'image Mouvement qui aura seulement les objets qui bougent seulement.

En C++ et OpenCv, il faut faire un `clone()` de l'image courant pour être sûr qu'on obtiens pas une référencé vers l'image courant mais vraiment un copie de l'image courant. C'est réalisé via

```
frame_prec = m_frame.clone();
```

#### Image Movement en niveau de gris

On utilise le LUT de OpenCV de base qui est `COLOR_BGR2GRAY`. Voici l'application :

```
/* Obtenir l'image Movement en niveau de grey */
cv::cvtColor(Movement, gray, cv::COLOR_BGR2GRAY);
```

#### Segmentation de l'image Movement

On utilise aussi la méthode prédéfini chez OpenCV qui est `THRESH_BINARY` via `cv::threshold(gray, edges, threshold_value, 255, cv::THRESH_BINARY);`

#### Application de la fermeture (érosion -> dilatation)

```
// Appliquer la dilatation
cv::dilate(edges, dilated, masque);

// Appliquer l'érosion
cv::erode(dilated, closed, masque);
```

La masque choisi est un rectangle de taille 15x15 qu'on définit à `cv::Mat masque = cv::getStructuringElement(cv::MORPH_RECT, cv::Size(15, 15));`

#### Calcul de la surface de contours & designer les rectangles

La librairie OpenCV viens avec le calcul de la surface de contours via une méthode intégré (`cv::contourArea(contour)`). En plus, il existe une méthode d'obtenir un rectangle approximatif qui s'appelle `boundingRect`. Pour nous donc il reste de trouver le limite qui nous permet d'accepter ou pas un contour et par extension un rectangle. Donc :

```
for (const auto &contour : contours)
{
    double area = cv::contourArea(contour); // calcul de la surface du
    contour

    if (area > 135 && area < 10000)
    {
```

```

        // Donner un rectangle approximatif du contour
        cv::Rect boundingBox = cv::boundingRect(contour); //
boundingbox: technique par matlab
        voitures_maintenant.push_back(boundingBox);           // Ajouter
le rectangle approximatif à la liste des voitures

        // Afficher le rectangle sur l'image
        cv::rectangle(frame, boundingBox, cv::Scalar(0, 255, 0), 2);
    }
}

```

Après plusieurs empirements, on a arrivé sur la condition `area > 135 && area < 10000`. Ainsi, si jamais il y a un rectangle approximatif qui passe le test, alors on va créer un objet rectangle pour le contour en question `boundingBox`.

#### Compter les voitures (algorithme)

1. \*\*Comparaison des rectangles:\*\* Pour chaque nouveau rectangle détecté (voiture actuelle), l'algorithme le compare avec les rectangles détectés précédemment.
2. \*\*Calcul de la distance:\*\* La distance euclidienne entre les centres des rectangles est calculée pour déterminer s'il s'agit du même véhicule.
3. \*\*Classification:\*\* Si le véhicule est considéré comme nouveau (distance supérieure au seuil), il est incrémenté dans le compteur de voitures à gauche ou à droite selon sa position.
4. \*\*Mise à jour:\*\* La liste des voitures précédentes est mise à jour avec la liste des voitures actuelles pour la prochaine itération.

```

// Compter les voitures selon les rectangles approximatifs donnees
void compter_voitures(std::vector<cv::Rect> &voitures_maintenant, int
&voitures_gauche, int &voitures_droite, cv::Mat &frame)
{
    const int distance_threshold = 100; // pour la distance entre les
centres des rectangles
    const int erreur = 550;           // pour la tolérance de l'erreur
sur la voie de gauche ou les voitures viens vers nous et on a un effet de
mal comptage quand les rectangles sont places de le debut

    for (const auto &voiture_maintenant : voitures_maintenant)
    {
        bool nouveau = true; // est que c'est nouveau le voiture ?

        for (const auto &voiture_prec : voitures_precedentes)
        {

            // Calcul de la distance euclidienne entre les centres des
deux rectangles
            cv::Point centre(voiture_maintenant.x +
voiture_maintenant.width / 2, voiture_maintenant.y +
voiture_maintenant.height / 2);
            cv::Point centre_prec(voiture_prec.x + voiture_prec.width / 2,

```

```

voiture_prec.y + voiture_prec.height / 2);

double distance = cv::norm(centre - centre_prec);

// If the distance is below the threshold, it's the same car
if (distance < distance_threshold)
{
    nouveau = false;
    break; // REALLY IMPORTANT: otherwise it doesn't stop at
the moment that we detected the new car
}

// Si c'est vraiment une nouvelle voiture, on incremente le
compteur global (passe en méthode pointeur)
if (nouveau)
{
    if ((voiture_maintenant.x) < (frame.cols / 2))
    {
        voitures_gauche++;
    }
    else
    {
        voitures_droite++;
    }
}
}

// Mis à jour de la liste voitures_precedentes avec la liste
voitures_maintenant
voitures_precedentes = voitures_maintenant;
}

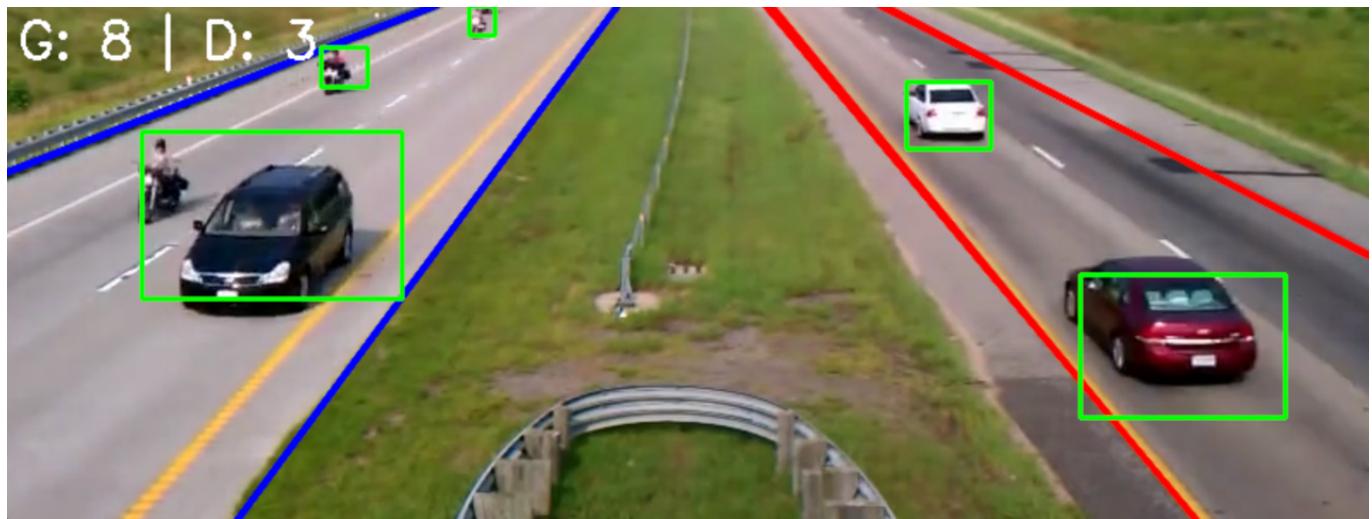
```

Le choix des paramètres et la gestion des cas limites sont cruciaux pour l'efficacité de comptage de véhicules. Le seuil de distance, détermine si deux rectangles correspondent au même véhicule. Un seuil trop faible risque de conduire à un double comptage, tandis qu'un seuil trop élevé pourrait entraîner des pertes de véhicules.

Par ailleurs il faut faire attention à la gestion des cas limites, tels que les véhicules qui sortent du champ de vision ou les erreurs de détection initiales. On a introduit la notion **erreur** pour la tolérance de l'erreur sur la voie de gauche ou les voitures viennent vers nous et on a un effet de mal comptage quand les rectangles sont placés au début.

Le principe de l'algorithme est la distance euclidienne entre les centres des deux rectangles qu'on compare avec **distance\_threshold**. Si c'est vraiment une nouvelle voiture, on incrémente le compteur global (passe en méthode pointeur). Il ne faut pas oublier de mettre à jour la liste **voitures\_precedentes** avec la liste **voitures\_maintenant**. C'est la liste avec les rectangles, donc ça veut dire les objets en mouvement qui sont détectés.

Voici un exemple d'application des différents étapes :



Pour aller plus loin...

Si jamais on est intéressé de garder qu'une seule ligne pour la voie gauche et une seule ligne pour la ligne droite, on peut appliquer une méthode de moyenne entre les deux lignes principales de la voie. Voici la logique de la fonction `std::vector<cv::Vec2f> keep_one_line(std::vector<cv::Vec2f> &lignes)`.

```
std::vector<cv::Vec2f> keep_one_line(std::vector<cv::Vec2f> &lignes)
{
    std::vector<cv::Vec2f> one_line;
    float rho_moyenne_gauche = 0;
    float theta_moyen_gauche = 0;
    float rho_moyenne_droite = 0;
    float theta_moyen_droite = 0;

    int nombre_droite = 0; // nombre de droite à droite
    int nombre_gauche = 0; // nombre de droite à gauche

    /* On va parcourir tous les lignes */
    for (size_t i = 0; i < lignes.size(); i++)
    {
        /* Obtenir les coordonées polaires de la ligne */
        float rho = lignes[i][0];
        float theta = lignes[i][1];

        /* Decision si la ligne est plus proche à gauche ou plus proche à
        droit selon theta et rho */
        if (rho < 0 && theta > CV_PI / 2) // est qu'on à gauche ? Si oui,
        alors theta est supérieur que pi/2
        {
            rho_moyenne_gauche += rho;
            theta_moyen_gauche += theta;
            nombre_gauche++;
        }
        else if (rho > 0 && theta < CV_PI / 2) // on à droite? Si oui,
        alors theta est inférieur que pi/2
        {
    }
```

```
    rho_moyenne_droite += rho;
    theta_moyen_droite += theta;
    nombre_droite++;
}
}

rho_moyenne_gauche = rho_moyenne_gauche / nombre_gauche;
theta_moyen_gauche = theta_moyen_gauche / nombre_gauche;

rho_moyenne_droite = rho_moyenne_droite / nombre_droite;
theta_moyen_droite = theta_moyen_droite / nombre_droite;

// Ajouter deux éléments au vecteur
one_line.push_back(cv::Vec2f(rho_moyenne_droite, theta_moyen_droite));
// Premier élément
one_line.push_back(cv::Vec2f(rho_moyenne_gauche, theta_moyen_gauche));
// Deuxième élément

return one_line; //il a une ligne pour la voie gauche et
}
```