MODAL Robots
Détection et suivi d'objets colorés avec évitement d'obstacles

Eloïse Berthier, Hugo Touvron

5 juin 2017

Introduction

L'objectif de ce projet est de permettre à un robot de détecter puis de suivre un objet coloré, tout en évitant un obstacle. Notre système doit pouvoir s'adapter sur la plateforme robotique TurtleBot.

Pour cela, nous avons utilisé le langage C++, la bibliothèque OpenCV, ainsi que la plateforme ROS.

Nous exposons d'abord les différentes méthodes de traitement d'image et de navigation que nous avons utilisées, puis nous précisons le fonctionnement global du système.

1 Méthodes de détection

1.1 Détection d'objets colorés

En premier lieu, nous avons réalisé un système de détection d'objets colorés. Voici le détail de son fonctionnement.

Construction du masque de couleur

Dans un premier temps, l'utilisateur sélectionne dynamiquement la couleur recherchée en cliquant sur une zone de l'image.

L'image est ensuite convertie en hsv (*Hue Saturation Value*).

On construit ensuite un masque, c'est-à-dire une image de la même taille que l'image d'entrée, sur laquelle apparaîtront en blanc (255) les pixels de la couleur recherchée, en noir (0) les autres. Nous gardons une tolérance (t) autour des valeurs cibles (h_0, s_0, v_0) pour h (teinte) et s (saturation). En revanche, le paramètre v (valeur) n'est pas considéré, ce qui permet de mieux détecter des objets dont la luminosité n'est pas uniforme.

Les pixels correspondants sur le masque auront donc pour valeur :

$$\begin{cases} 255 & \text{si } h_0 - t \leq h < h_0 + t \text{ et } s_0 - t \leq s < s_0 + t \\ 0 & \text{sinon} \end{cases}$$

Pour la construction de ce masque, nous avons utilisé une fonction OpenCV qui permet d'appliquer cette règle logique à l'image entière :

```
cvInRangeS(hsv, cvScalar(h - tolerance -1, s - tolerance, 0),
          cvScalar(h + tolerance -1, s + tolerance, 255), mask);
```

La valeur de t doit être réglée pour qu'on sélectionne la couleur choisie : si t est trop faible, seule une partie de l'objet sera sélectionnée en raison des différences d'éclairage, si t est trop élevé, le masque sélectionnera d'autres objets de couleurs proches.

Élimination du bruit

Le masque que nous obtenons détecte bien l'objet, mais il est bruité. Pour éliminer ce bruit, nous avons d'abord utilisé un flou gaussien (gaussian blur). La fonction OpenCV suivante permet d'effectuer cette opération :

```
GaussianBlur(img,img,Size(5,5),2, 2);
```

Cela s'est avéré insuffisant, car pour éliminer tout le bruit, il faut appliquer un paramètre de flou important, ce qui réduit fortement la taille de l'objet détecté.

Nous avons donc utilisé la méthode d'érosion / dilatation. Cette méthode a l'avantage de conserver la taille des objets qui ne sont pas supprimés par l'érosion. Nous avons utilisé les fonctions OpenCV erode et dilate.

```
int erosion_size=2;
Mat element=getStructuringElement(cv::MORPH_CROSS,
           cv::Size(2*erosion_size+1,2*erosion_size+1),
           cv::Point(erosion_size,erosion_size));
erode(img,img,element);
dilate(img,img,element);
```

Estimation de la taille et de la position de l'objet

Nous calculons ensuite le centre de gravité des pixels blancs de ce masque, ainsi que leur surface. Cela nous permet d'avoir une estimation de la position de l'objet par rapport à la caméra, ainsi que de sa taille apparente, donc de sa distance à la caméra.

Enfin, le centre de gravité de l'objet est affiché sur l'image de départ (point rouge).

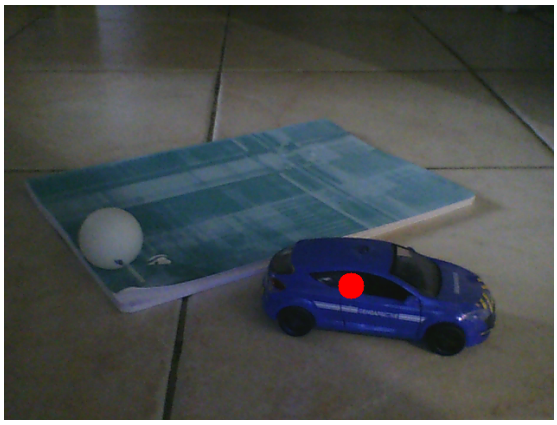


FIGURE 1 – Image acquise par la caméra.

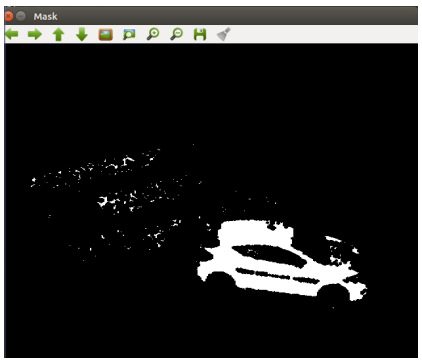


FIGURE 2 – Masque de détection de couleur.

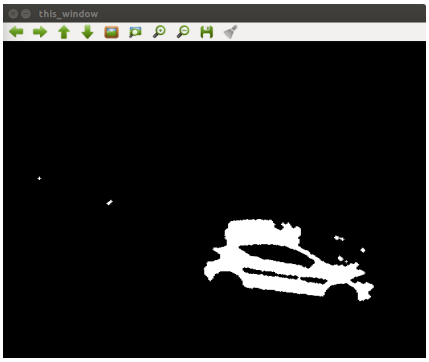


FIGURE 3 – Masque de détection de couleur après débruitage.

1.2 Détection de formes

Pour compléter la détection d'un objet par sa couleur, nous avons essayé de reconnaître également sa forme.

La première solution que nous avons envisagée était de détecter des contours sur le masque de couleur. Pour cela, nous avons utilisé la fonction OpenCV qui implémente l'algorithme de détection de contours créé par John F. Canny en 1986 :

```
Canny(src_gray, detected_edges, lowThreshold,
     lowThreshold*ratio, kernel_size );
```

Or les contours détectés sont très instables au cours du temps. De plus, les temps de calcul sont assez importants pour des images arrivant en temps réel. Il

est donc difficile d'obtenir une estimation fiable et continue de la position d'un objet.

Nous avons donc supposé que nous connaissons la forme de l'objet détecté (une balle). Cela nous permet de chercher à détecter une forme précise : un cercle. Nous avons pour cela utilisé la transformée de Hough, qui permet de détecter des lignes ou des courbes.

```
HoughCircles(src_gray, circles, CV_HOUGH_GRADIENT, 2,
             src_gray.rows/4, 150, 50,0,120);
```

Nous avons par ailleurs constaté que la détection de formes directement sur le masque crée précédemment donnait de très mauvais résultats. C'est pourquoi nous procédons à cette détection sur l'image de départ, convertie en nuances de gris, puis floutée par un flou gaussien. Finalement, les cercles détectés sont affichés sur l'image en nuances de gris.

FIGURE 4 – Détection de cercles par transformée de Hough.

1.3 Détection d'obstacles

Dans un premier temps, nous avons envisagé de faire de la détection d'obstacles grâce au flot optique. Mais en raison des mouvements possibles de la cible, nous avons décidé de faire de la détection d'obstacles avec la caméra de profondeur.

FIGURE 5 – Image obtenue avec la caméra de profondeur. Les zones proches apparaissent en foncé.

Nous exposons ici le principe de notre méthode de détection d'obstacles avec la caméra de profondeur. Tout d'abord, à partir de cette caméra, nous récupérons une matrice contenant la distance caméra/objet en mm, pour chacun des pixels représentés.

```
Mat depthImage=cv_ptr2depth->image;
Mat depthSeuil(depthImage.rows, depthImage.cols, CV_8UC1,
               Scalar::all(0));
```

Ensuite nous créons une nouvelle image où nous mettons en blanc les points qui sont assez proches de nous (80 cm de distance). Nous les considérons ainsi comme un obstacle.

```
float v;
for(int i=0;i<depthImage.rows;i++){
    for(int j=0; j<depthImage.cols;j++){
        v=depthImage.at<float>(i, j);
        if(v!=0 && v<=thresh)
            depthSeuil.at<uchar>(i,j)=255;
        if(initialized && InitDepth.at<uchar>(i,j)==255)
            depthSeuil.at<uchar>(i,j)=0;
    }
}
```

Toutefois, en procédant ainsi, le sol apparaît comme un obstacle. Pour remédier à ce problème, nous enregistrons la première image que récupère le robot, nous considérons que sur cette image il n'y a pas d'obstacles et nous retranchons la valeur des pixels de la matrice de cette image aux pixels correspondants dans les matrices des autres images. Ainsi, la caméra de profondeur est calibrée et le sol n'apparaît plus comme un obstacle.

```
if(initialized==false){
    InitDepth=depthSeuil.clone();
    initialized=true;
}
```

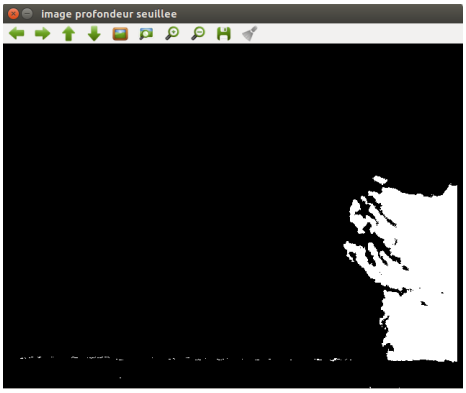


FIGURE 6 – Image de profondeur seuillée avec correction du sol.

Une fois que nous avons cette image en noir et blanc où apparaissent en blanc les obstacles, nous calculons le centre de gravité de l'obstacle.

```
for(int x = 0; x < depthSeuil.rows; x++) {
    for(int y = 0; y < depthSeuil.cols; y++) {
        // If its a tracked pixel, count it to the center of
        // gravity's calcul.
        if(depthSeuil.at<uchar>(y,x)==255) {
            sommeX += x;
            sommeY += y;
            (nbPixels)++;
        }
    }
}

CvPoint centreObst;
// If there is no pixel, we return a center outside the image,
// else we return the center of gravity.
if(nbPixels > 0)
    centreObst=cvPoint((int)(sommeX / (nbPixels)),
    (int)(sommeY / (nbPixels)));
else
    centreObst=cvPoint(-1, -1);
if(obst>MintailleObst)
    PresObsta=true;
```

```
else
    PresObsta=false;
centreXObs=centreObst.x;
```

Enfin, nous considérons qu'il y a un obstacle si sa surface est supérieure à un seuil, et nous mettons à jour une variable booléenne globale indiquant la présence ou non de cet obstacle.

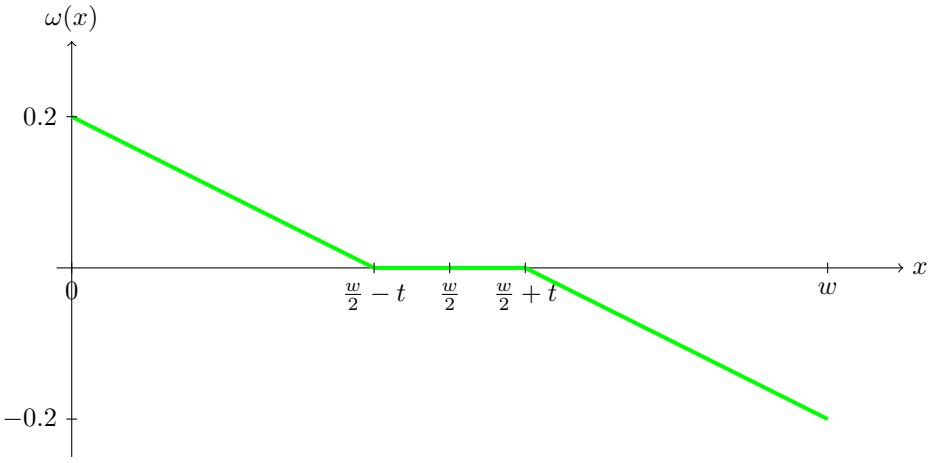
2 Méthodes de navigation

La détection de l'objet suivi ainsi que celle des obstacles nous permet de calculer le déplacement du robot.

2.1 Suivi d'objet

La détection d'objet coloré permet d'estimer deux paramètres : x, l'abscisse du centre de l'objet suivi, et r sa surface apparente.

Le paramètre x permet de calculer la vitesse angulaire du robot : si x est à gauche de l'image, on tourne à gauche, s'il est à droite, on tourne à droite. La vitesse est linéaire en fonction de la distance au centre, pour rendre les mouvements plus fluides.




```
//angular velocity
if (x > (width/2)+horizontalTolerance)
        angularVel = -0.2/( (width/2)-horizontalTolerance)*x-0.2
        +(0.2*width / (width/2 - horizontalTolerance));
else if (x < (width/2)-horizontalTolerance)
        angularVel = -0.2/( (width/2)-horizontalTolerance)*x+0.2;
```

Avec le paramètre r, nous estimons la vitesse linéaire. Si r est inférieur à un seuil de détection, il n'avance pas, car l'objet n'est pas détecté. Si r est

supérieur à ce seuil, le robot avance avec une vitesse affine en fonction de r. Enfin, si r est supérieur à la valeur cible (c'est-à-dire que le robot a atteint la cible), il s'arrête. Cela permet également d'arrêter manuellement le robot en sélectionnant une zone de couleur étendue (un mur par exemple).

```
//linear velocity
if (r > rGoal + rTolerance)
    linearVel = 0;
else if (r < rGoal - rTolerance)
    linearVel = 0.1/(rGoal-rTolerance-rmin)*r +0.1
                -0.1*rmin/(rGoal-rTolerance-rmin);
if(r<rmin || x<0 || y<0){
    linearVel = 0;
    angularVel = 0;
}
```

Le comportement du robot est résumé par le schéma ci-dessous :

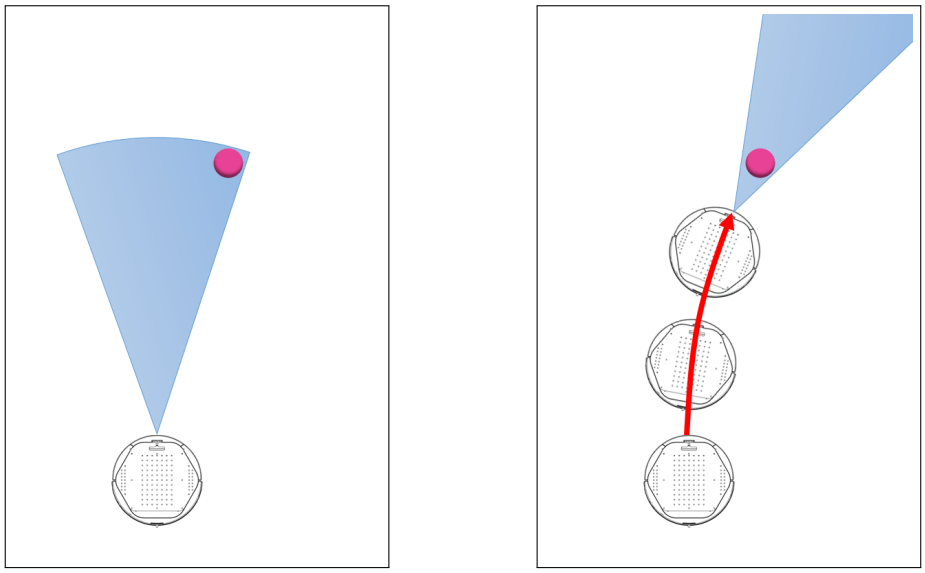


FIGURE 7 – Déplacement du robot en l'absence d'obstacle.

Nous faisons intervenir la détection ou non du cercle comme une modulation des vitesses du robot : il accélère quand il détecte la forme, et ralentit sinon. Cette détection de cercle intervient peu, car elle reste assez instable.

```
if(!ball){
    linearVel = 0.8*linearVel;
    angularVel = 0.8*angularVel;
} else{
```

```
    linearVel = 1.2*linearVel;
    angularVel = 1.2*angularVel;
}
```

2.2 Évitement d'obstacles

La détection d'un obstacle permet de modifier la trajectoire du robot pour l'éviter. La trajectoire du robot avec évitement d'obstacle est résumée par la figure 8.

En fonction de sa position, nous faisons varier la vitesse angulaire linéairement pour pouvoir éviter l'obstacle. Si l'obstacle est sur sa gauche, le robot tourne à droite, et inversement. De plus, si l'obstacle est en position centrale, le robot tourne préférentiellement dans la direction de l'objet suivi (représentée par x). Cela permet d'éviter que le robot ne perde l'objet de vue en évitant l'obstacle.

```
if(PresObsta){
    if (centreXObs>(width/2)+(width/8))
        angularVel=0.15;
    else if (centreXObs < (width/2)-(width/8))
        angularVel =-0.15;
    else
        angularVel=-sign(x-width/2)*0.15;
}
```

Nous avons rencontré une difficulté principale lors des tests. La caméra de profondeur ne peut pas percevoir les obstacles proches (moins de 50cm environ). Ainsi, en utilisant notre algorithme, le robot évite l'obstacle, mais à tendance à "se rabattre" trop tôt, et à percuter l'obstacle, car il recommence à suivre la balle.

Pour remédier à ce problème, nous avons envisagé plusieurs solutions. La première est de modifier, une fois l'obstacle détecté, la position de la cible à suivre. Cela permet au robot de s'écarter davantage de l'obstacle.

La seconde est de donner une consigne persistante dans le temps : lorsqu'un obstacle est détecté, le robot effectue la manœuvre d'évitement pendant une durée prédéfinie. Pour obtenir de meilleurs résultats, il faudrait ajuster cette durée en fonction de la position estimée de l'obstacle, et de sa distance à la caméra.

Ces méthodes ont permis d'améliorer la trajectoire du robot, mais ne résolvent pas totalement le problème. Pour obtenir des résultats plus satisfaisants, il faudrait établir une cartographie locale, construite à partir des observations, pour calculer des consignes plus précises et plus adaptées.

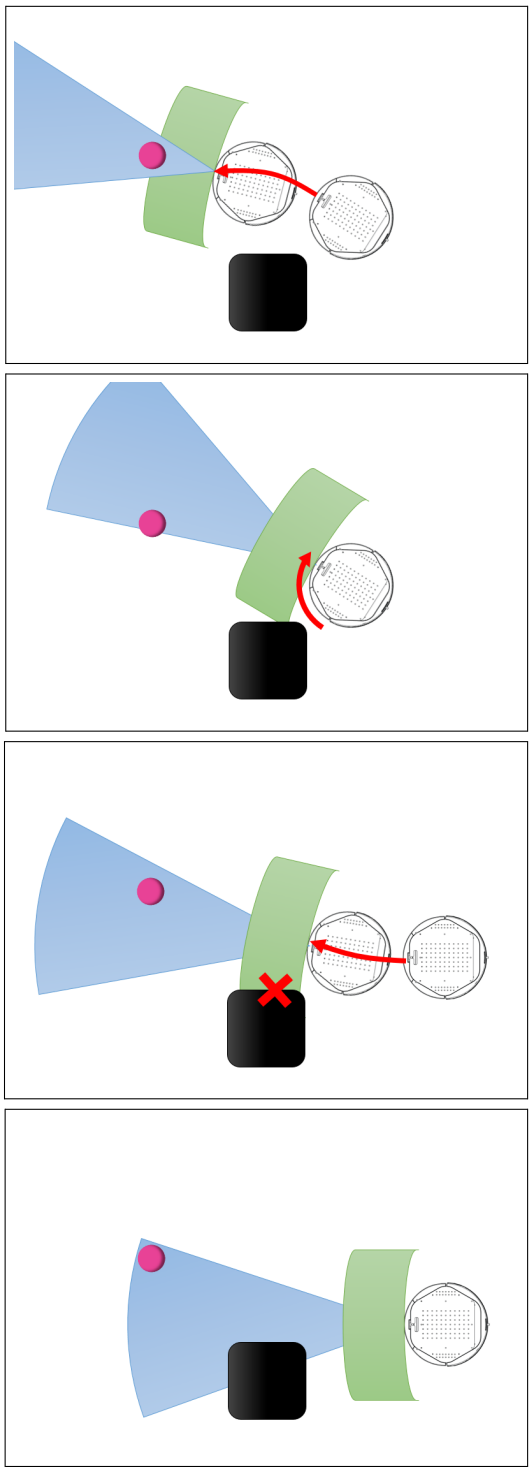


FIGURE 8 – Déplacement du robot avec un obstacle.

2.3 Filtrage global des vitesses

Nous avons constaté lors de nos tests que les mouvements du robot étaient très saccadés. En effet, dès que la détection de couleur, de forme ou d'obstacle varie, la vitesse du robot est modifiée. Or ces détections peuvent varier brutalement, pour des raisons exogènes (variations de l'éclairage, bruit...)

Pour rendre le mouvement plus fluide, nous avons procédé à un filtrage élémentaire. Il s'agit d'une version très simplifiée d'un filtre de Kalman. Nous procédons en deux phases :

Prédiction : Nous connaissons les valeurs de la vitesse à l'état précédent : $v(t-1)$ et $\omega(t-1)$. Nous faisons la prédiction suivante : l'état actuel des vitesses sera le même que l'état précédent.

Mise à jour : Les observations à l'état courant t permettent de calculer les valeurs de vitesse linéaire et angulaire définies précédemment $v_0(t)$ et $\omega_0(t)$. Nous donnons donc une consigne de vitesse qui tient compte de ces deux étapes :

$$\begin{cases} v(t) = (1-\rho)\ v(t-1) + \rho\ v_0(t) \\ \omega(t) = (1-\rho)\ \omega(t-1) + \rho\ \omega_0(t) \end{cases}$$

Le paramètre $\rho \in (0,1)$ représente l'importance relative que nous donnons à l'observation à l'état présent. Plus ρ est important, plus nous considérons l'observation comme fiable, et plus nous modifions la vitesse.

En pratique, nous avons utilisé le paramètre $\rho = 0.33$, qui donne un bon compromis entre fluidité et réactivité du robot.

3 Intégration à la plateforme robotique

Pour intégrer notre travail à la plateforme TurtleBot, nous avons utilisé l'interface ROS, qui nous a permis :

— de communiquer avec le robot pour recevoir des images et envoyer des consignes de vitesse,
— de gérer des flux d'information en temps réel.

L'architecture que nous avons choisie se décompose en trois nœuds :

1. Le nœud **cam_tracking_node** qui reçoit l'image rgb, et génère le masque de couleur et la détection de cercles. Il publie le topic **centre** sous forme d'un vecteur contenant la position du centre de l'objet, sa taille, et la présence ou non d'un cercle.
2. Le nœud **depth_node** qui reçoit l'image de profondeur. Il publie le topic **obstacle** contenant la présence ou non d'un obstacle, et sa position.
3. Le nœud **follower_node**, qui souscrit aux topics **centre** et **obstacle**, et calcule le mouvement du robot. Il publie sur le topic **cmd_vel_mux**.

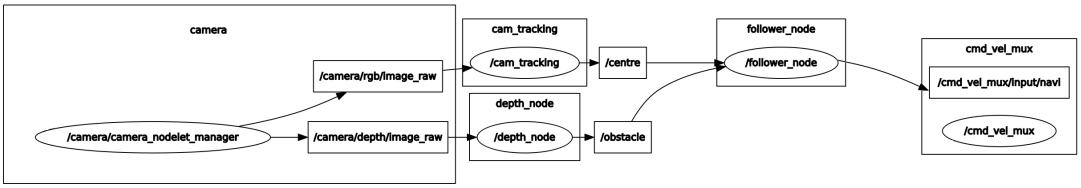


FIGURE 9 – Plan des nodes et topics produit par rqt_rosgraph.

Le but de cette architecture est de minimiser le nombre de transferts d'images pour gagner en rapidité, tout en séparant les différentes fonctions de façon modulaire. Ainsi, il est possible de changer une solution technique, sans modifier les autres composantes.

D'autre part, les calculs effectués par les nœuds sont assez rapides, et les déplacements du robot assez lents pour qu'il n'y ait pas besoin synchroniser les flux d'information.

Conclusion

Ce projet a permis de combiner des méthodes de traitement d'images et des méthodes de navigation. Les résultats des tests sur le robot sont satisfaisants pour la fonction de détection et suivi d'objet coloré.

En revanche, notre méthode de navigation est insuffisante pour permettre un évitement d'obstacle efficace. Elle pourrait être améliorée en établissant une cartographie locale des obstacles. L'architecture de notre module ROS permet de modifier facilement le nœud qui calcule les consignes de navigation.