Jean-Baptiste FRAU - Janvier 2019
Projet GE : Evaluation d'une méthode de planification de chemins par réseaux de neurones
OU Mise en place d'un protocole pour la démonstration d'un algorithme de planification de chemin

Note d'application

Commande du robot Khepera III et implémentation en C++

Généralités

Dans notre système, la position (x,y) et l'orientation du robot (alpha) dans le repère de la webcam (O, x, y) est connue en temps réel grâce au module ArUco de la librairie OpenCV.

Le robot Khepera III a deux roues fixes d'axe de rotation commun (**fig.1**). Les actionneurs sont donc les 2 groupes moteur+roue du robot que l'on commandera en vitesse.

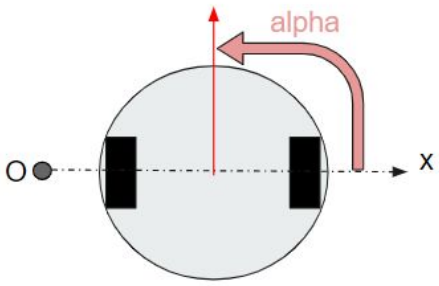

fig.1 Schéma du robot Khepera III

Le but de cette commande est de faire suivre au robot le chemin calculé par l'algorithme du client exprimé en un ensemble P ordonné de points du repère webcam (O, x, y) (**fig.2**).

ex : P = {(x0, y0) ; (x1, y1) ; … ; (xn, yn)}

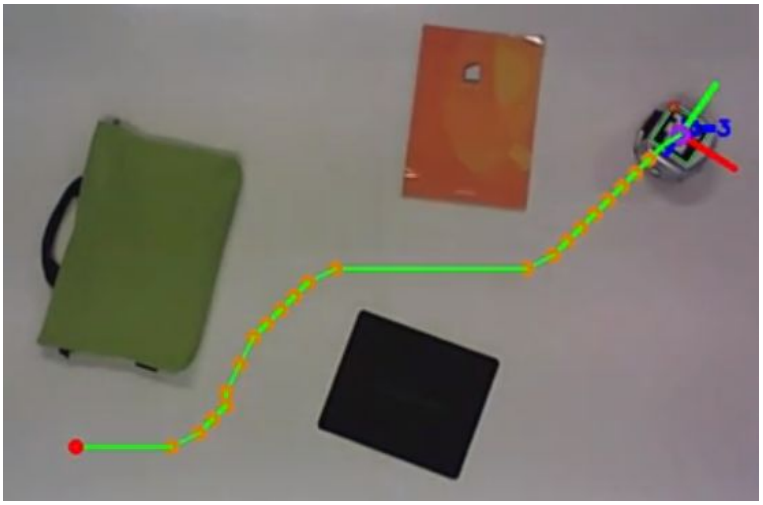

fig.2 Affichage du chemin dans l'interface graphique, avec les points de P en orange (en rouge pour le point d'arrivée)

Stratégie de commande

La stratégie de commande adoptée consiste à faire évoluer le robot d'un point à un autre jusqu'à ce qu'il atteigne le point d'arrivée. De cette manière, le problème est simplifié car on se concentre à faire rejoindre en ligne droite au robot un point (xi, yi) de P, noté dans le sous-problème (xP, yP) (**fig.3**). Lorsque le robot se trouve à un voisinage de ce point, on reproduit l'opération en considérant le point suivant de l'ensemble P : (xP, yP) devient (xi+1, yi+1).

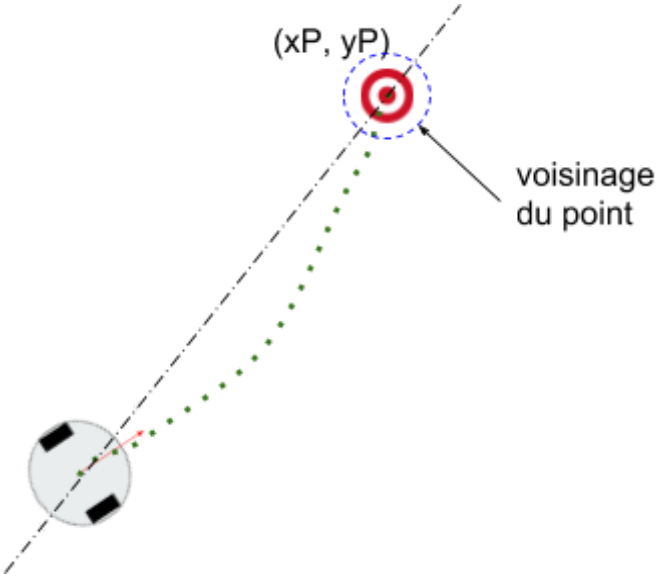

fig.3 Schéma du sous-problème : le robot doit rejoindre (xP, yP) si possible en ligne droite

Pour faire aller le robot en ligne droite jusqu'au point (xP, yP), l'idée est de commander l'orientation du robot de manière à ce qu'il se dirige droit vers la cible. On connaît déjà la position du robot et son orientation ainsi que la position de la cible dans (O, x, y). On peut donc calculer en temps réel le vecteur u qui relie le centre du robot à la cible et estimer l'orientation angulaire de u et du vecteur directeur d du robot dans (O, x, y) (**fig.4**).

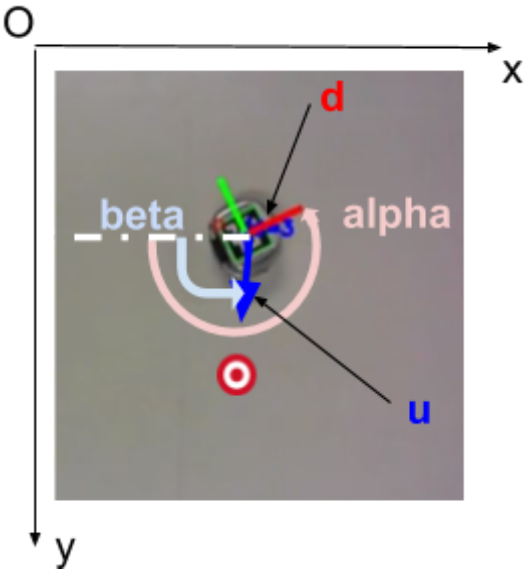

fig.4 Schéma montrant les vecteurs u et d et leur orientation angulaire (respectivement beta et alpha) dans (O, x, y)

Si on reformule le sous-problème, le but est de maintenir en permanence u et d sur le même axe en agissant sur l'orientation de d.

La démarche suivie est d'actualiser en permanence alpha et beta et d'appliquer le raisonnement suivant :

- **si alpha appartient à [beta - epsilon ; beta + epsilon]**, avec epsilon une petite erreur angulaire → *on applique la même vitesse aux deux roues du robot de manière à ce qu'il avance en ligne droite*
- **sinon alpha n'appartient pas à cet intervalle** → *on compare alpha et beta* :
  - **si alpha > beta** → *on applique une vitesse de rotation pure du robot vers sa droite*
  - **sinon alpha < beta** → *on applique une vitesse de rotation pure du robot vers sa gauche*

Implémentation en C++

Structure

La commande est concernée par deux fichiers :

- main.cpp (**fig.5**) : ce fichier coordonne l'ensemble du système, permet d'acquérir les données utiles à la réalisation de la commande (position et orientation du robot, coordonnées du prochain point cible (xP, yP)), et permet d'envoyer les vitesses des roues une fois calculées au robot.
- asservi.cpp : ce fichier contient la définition des vitesses des roues telle que vue dans la partie précédente.

```
asservissement(pos_x, pos_y, alpha, xP, yP, &Vd, &Vg);
sprintf(commande,"L%dR%d",Vg,Vd);
// Run
if(strcmp(commande,tmp) != 0) {
      algorithm_run(commande);
}
```

fig.5 extrait de main.cpp où se situe l'appel de la fonction asservissement()

Problème rencontré

Le principal problème rencontré dans l'implémentation de cette solution est que les angles alpha et beta sont définis dans le repère (O, i, j) et que la discontinuité dans la mesure des angles au passage de 360° à 0° entraîne des erreurs lorsqu'on se contente de faire la différence alpha-beta (première solution adoptée).

C'est pourquoi, cette comparaison entre alpha et bêta a été réalisée de manière un peu plus astucieuse dans l'idée de limiter voire supprimer les erreurs dues à la discontinuité.

La solution consiste à identifier dans quel cadran de la représentation des angles se trouve les vecteurs u et d pour éviter des erreurs de comparaison, et ensuite à reprendre le raisonnement initial, c'est à dire : vérifier si d se trouve orienté au voisinage de l'orientation de u.

On remarque dans la **fig.6**, qu'après avoir vérifié si on se trouve hors du voisinage de la cible (D>5), on vérifie si d se trouve dans le cadran]5° ; 90°[puis si beta se trouve dans le cadran]0° ; 90°[. Si oui, u et d sont dans le même cadran et on procède à la comparaison des angles. Sinon, si beta ne se trouve pas au point de discontinuité, on cherche à ramener u et d dans le même cadran par rotation. Et si alpha se trouve dans l'intervalle [360°=0° ; 5] et que beta est au point de discontinuité (alors alpha est au voisinage de beta), on fait avancer le robot tout droit. Le reste de la fonction relève de la même algorithmie.

```
if( D > 5){
      if(alpha_deg>5&&alpha_deg<90){
          if(beta_deg>0&&beta_deg<90){
              if(alpha_deg-beta_deg>ang_seuil){*Vg=v_rot; *Vd=-v_rot;}
              else if(alpha_deg-beta_deg<-ang_seuil){*Vg=-v_rot; *Vd=v_rot;}
              else{*Vg=v_dot; *Vd=v_dot;}
          }
          else if(beta_deg>=90&&beta_deg<180){
              *Vg=-v_rot; *Vd=v_rot;
          }
          else if(beta_deg>=180&&beta_deg<270){
              if(beta_deg-alpha_deg>180){*Vg=v_rot; *Vd=-v_rot;}
              else if(beta_deg-alpha_deg<180){*Vg=-v_rot; *Vd=v_rot;}
          }
          else if(beta_deg>=270&&beta_deg<360){
              *Vg=v_rot; *Vd=-v_rot;
          }
          else if(beta_deg==0||beta_deg==360){*Vg=4000; *Vd=4000;}
      }
```

fig.6 extrait de asservi.cpp montrant une partie de la nouvelle solution