Projet de perception pour la robotique

Andrea Cherubini

12 novembre 2020

Table des matières

Chapitre 1

Etalonnage de caméra

1.1 Objectifs

Vous allez étalonner la caméra du robot e-puck en utilisant un modèle de projection centrale et un objet d'étalonnage 3D (cube). Vous déterminerez ainsi les paramètre intrinsèques et extrinsèques (position par rapport au repère robot) de la caméra. Vous exploiterez ces résultats pour la suite du projet avec le e-puck (Travaux pratiques à venir).

1.2 Matériel

— un robot e-puck avec sa caméra,
— un cube d'étalonnage dont les carrés sont 1cm de côté,
— une machine ubuntu pour enregistrer les images acquises par le robot,
— une machine avec Matlab pour traiter une image du e-puck afin de déterminer les paramètres intrinsèques et extrinsèques de la caméra.

1.3 Etalonnage avec modèle sténopé

L'étalonnage se fera avec le modèle linéaire de caméra, appelé également modèle sténopé ou encore modèle à projection centrale car il est basé sur le principe de projection illustré en Fig. 1.1.

1.3.1 Modèle extrinsèque

Cette transformation exprime le passage du repère monde $\mathcal{R}_W$ au repère de la caméra $\mathcal{R}_C$ par une translation du repère monde vers le centre du repère

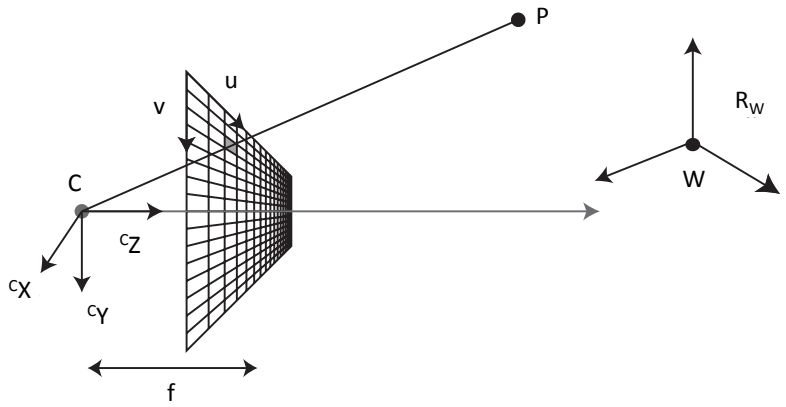

FIGURE 1.1 – Principe de formation de l'image par projection centrale.

caméra suivie d'une rotation permettant de faire correspondre les axes des deux repères (voir Fig. 1.1.).

Ainsi, les coordonnées $\left[{}^{W}X\ {}^{W}Y\ {}^{W}Z\right]^{\top}$ d'un point dans le repère monde s'expriment dans le repère de la caméra $\left[{}^{C}X\ {}^{C}Y\ {}^{C}Z\right]^{\top}$ à l'aide du formalisme des matrices homogènes :

$$\begin{bmatrix} {}^{C}X \\ {}^{C}Y \\ {}^{C}Z \\ 1 \end{bmatrix} = {}^{C}\mathbf{T}_{W} \begin{bmatrix} {}^{W}X \\ {}^{W}Y \\ {}^{W}Z \\ 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & p_x \\ r_{21} & r_{22} & r_{23} & p_y \\ r_{31} & r_{32} & r_{33} & p_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} {}^{W}X \\ {}^{W}Y \\ {}^{W}Z \\ 1 \end{bmatrix}, \qquad (1.1)$$

avec : $[p_x\ p_y\ p_z]^{\top}$ coordonnées de W dans $\mathcal{R}_C$, $[r_{11}\ r_{21}\ r_{31}]^{\top}$ coordonnées de ${}^{W}\overrightarrow{X}$ dans $\mathcal{R}_C$, $[r_{12}\ r_{22}\ r_{32}]^{\top}$ coordonnées de ${}^{W}\overrightarrow{Y}$ dans $\mathcal{R}_C$ et $[r_{13}\ r_{23}\ r_{33}]^{\top}$ coordonnées de ${}^{W}\overrightarrow{Z}$ dans $\mathcal{R}_C$.

1.3.2 Modèle intrinsèque

Dans un premier temps, on effectue une transformation du repère de la caméra $\mathcal{R}_C$ au repère du plan image situé à la distance focale f de $\mathcal{R}_C$. Puis on passe du plan image aux coordonnées pixel (u, v), par un recalage et une mise à l'échelle (Fig. 1.1). Pour cela, on effectue une translation qui tient compte du centre du plan image dans le repère de visualisation (u_0, v_0) puis d'une mise à l'échelle qui permet d'exprimer les coordonnées métriques en pixels (u, v). Comme il est impossible de dissocier la focale f des facteurs de mise à l'échelle (k_u, k_v), on effectue le changement de variables : $\alpha_u = k_u f$, $\alpha_v = k_v f$. En posant aussi $U = uS$ et $V = vS$ (avec λ un facteur d'échelle

scalaire et $S = \lambda^C Z$), on obtient l'équation du modèle sténopé :

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \lambda \begin{bmatrix} \alpha_u & 0 & u_0 & 0 \\ 0 & \alpha_v & v_0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} {}^C X \\ {}^C Y \\ {}^C Z \\ 1 \end{bmatrix}. \tag{1.2}$$

1.3.3 Modèle complet

En remplaçant (1.2) dans (1.1), on obtient le modèle complet :

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \lambda \begin{bmatrix} (\alpha_u r_{11} + r_{31} u_0) & (\alpha_u r_{12} + r_{32} u_0) & (\alpha_u r_{13} + r_{33} u_0) & (\alpha_u p_x + p_z u_0) \\ (\alpha_v r_{21} + r_{31} v_0) & (\alpha_v r_{22} + r_{32} v_0) & (\alpha_v r_{23} + r_{33} v_0) & (\alpha_v p_y + p_z v_0) \\ r_{31} & r_{32} & r_{33} & p_z \end{bmatrix} \begin{bmatrix} {}^W X \\ {}^W Y \\ {}^W Z \\ 1 \end{bmatrix} \tag{1.3}$$

que l'on note aussi :

$$\begin{bmatrix} U \\ V \\ S \end{bmatrix} = \lambda \begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \end{bmatrix} \begin{bmatrix} {}^W X \\ {}^W Y \\ {}^W Z \\ 1 \end{bmatrix}. \tag{1.4}$$

On note :

$$\begin{cases} \mathbf{c_1} & = & \begin{bmatrix} c_{11} & c_{12} & c_{13} \end{bmatrix} \\ \mathbf{c_2} & = & \begin{bmatrix} c_{21} & c_{22} & c_{23} \end{bmatrix} \\ \mathbf{c_3} & = & \begin{bmatrix} c_{31} & c_{32} & c_{33} \end{bmatrix} \end{cases}. \tag{1.5}$$

A partir de (1.3), (1.4) et (1.5) on peut déduire les relations suivantes. Pour les paramètre intrinsèques :

$$\begin{cases} \lambda & = & \|\mathbf{c_3}\| \\ u_0 & = & \mathbf{c_1}\mathbf{c_3}^\top / \lambda^2 \\ v_0 & = & \mathbf{c_2}\mathbf{c_3}^\top / \lambda^2 \\ \alpha_u & = & \sqrt{\mathbf{c_1}\mathbf{c_1}^\top / \lambda^2 - u_0^2} \\ \alpha_v & = & \sqrt{\mathbf{c_2}\mathbf{c_2}^\top / \lambda^2 - v_0^2} \end{cases}. \tag{1.6}$$

Pour les paramètre extrinsèques :

$$\begin{cases} \tilde{\mathbf{r}}_1 & = & (\mathbf{c_1} - u_0 * \mathbf{c_3}) / (\lambda \alpha_u) \\ \tilde{\mathbf{r}}_2 & = & (\mathbf{c_2} - v_0 * \mathbf{c_3}) / (\lambda \alpha_v) \\ \mathbf{r_1} & = & \tilde{\mathbf{r}}_1 / \|\tilde{\mathbf{r}}_1\| \\ \mathbf{r_2} & = & \tilde{\mathbf{r}}_2 / \|\tilde{\mathbf{r}}_2\| \\ \mathbf{r_3} & = & \mathbf{c_3} / \lambda \\ p_x & = & (c_{14} - u_0 c_{34}) / (\lambda * \alpha_u) \\ p_y & = & (c_{24} - v_0 c_{34}) / (\lambda * \alpha_v) \\ p_z & = & c_{34} / \lambda \end{cases} \tag{1.7}$$

On pourra enfin compléter la matrice ${}^{C}\mathbf{T}_{W}$ en sachant que :

$$\left\{ \begin{array}{rcl} \mathbf{r_1} & = & \left[\begin{array}{ccc} r_{11} & r_{12} & r_{13} \end{array} \right] \\ \mathbf{r_2} & = & \left[\begin{array}{ccc} r_{21} & r_{22} & r_{23} \end{array} \right] \\ \mathbf{r_3} & = & \left[\begin{array}{ccc} r_{31} & r_{32} & r_{33} \end{array} \right] \end{array} \right. . \tag{1.8}$$

A noter que en général la matrice ${}^{C}\mathbf{R}_{W}$ ainsi obtenue n'est pas orthonormée.

L'objectif de l'étalonnage sera d'estimer les valeurs $c_{11}, \dots c_{34}$ et d'en déduire les paramètres intrinsèques et extrinsèques grâce aux équations (1.6), (1.7) et (1.8).

1.3.4 Méthode des moindres carrés

Pour tout point P, la projection 2D dans l'image (u, v) peut être associée à sa position 3D dans le monde $\left({}^{W}X, {}^{W}Y, {}^{W}Z\right)$ par deux équations linéaires indépendantes [1] où les 12 inconnues sont les paramètres $c_{11} \dots c_{34}$:

$$\left\{ \begin{array}{rcl} (c_{11} - c_{31}u)\,{}^{W}X + (c_{12} - c_{32}u)\,{}^{W}Y + (c_{13} - c_{33}u)\,{}^{W}Z + (c_{14} - c_{34}u) & = & 0 \\ (c_{21} - c_{31}v)\,{}^{W}X + (c_{22} - c_{32}v)\,{}^{W}Y + (c_{23} - c_{33}v)\,{}^{W}Z + (c_{24} - c_{34}v) & = & 0 \end{array} \right. \tag{1.9}$$

En normalisant c_{34} à une valeur de 1 (ou en divisant les deux équations par c_{34}, le nombre de paramètres à déterminer n'est plus que de 11. Le système peut alors s'exprimer sous la forme matricielle

$$\mathbf{AX} = \mathbf{B}, \tag{1.10}$$

avec le vecteur des 11 paramètres recherchés :

$$\mathbf{X} = \left[c_{11}\ c_{12}\ c_{13}\ c_{14}\ c_{21}\ c_{22}\ c_{23}\ c_{24}\ c_{31}\ c_{32}\ c_{33}\right]^{\top}, \tag{1.11}$$

la matrice $\mathbf{A}$ indexée par rapport aux n points d'étalonnage :

$$\mathbf{A} = \left[\begin{array}{ccccccccccc} {}^{W}X_1 & {}^{W}Y_1 & {}^{W}Z_1 & 1 & 0 & 0 & 0 & 0 & -u_1{}^{W}X_1 & -u_1{}^{W}Y_1 & -u_1{}^{W}Z_1 \\ 0 & 0 & 0 & 0 & {}^{W}X_1 & {}^{W}Y_1 & {}^{W}Z_1 & 1 & -v_1{}^{W}X_1 & -v_1{}^{W}Y_1 & -v_1{}^{W}Z_1 \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ {}^{W}X_i & {}^{W}Y_i & {}^{W}Z_i & 1 & 0 & 0 & 0 & 0 & -u_i{}^{W}X_i & -u_i{}^{W}Y_i & -u_i{}^{W}Z_i \\ 0 & 0 & 0 & 0 & {}^{W}X_i & {}^{W}Y_i & {}^{W}Z_i & 1 & -v_i{}^{W}X_i & -v_i{}^{W}Y_i & -v_i{}^{W}Z_i \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots & \dots \\ {}^{W}X_n & {}^{W}Y_n & {}^{W}Z_n & 1 & 0 & 0 & 0 & 0 & -u_n{}^{W}X_n & -u_n{}^{W}Y_n & -u_n{}^{W}Z_n \\ 0 & 0 & 0 & 0 & {}^{W}X_n & {}^{W}Y_n & {}^{W}Z_n & 1 & -v_n{}^{W}X_n & -v_n{}^{W}Y_n & -v_n{}^{W}Z_n \end{array} \right] \tag{1.12}$$

et

$$\mathbf{B} = \left[u_1\ v_1\ \dots\ u_i\ v_i\ \dots\ u_n\ v_n\right]^{\top}. \tag{1.13}$$

1. Obtenues en remplaçant $S = \lambda^{C}Z$ dans (1.4).

Il faut donc un minimum de 6 points d' étalonnage afin de pouvoir estimer les paramètres du modèle complet. Pour des raisons de non dégénérescence du système, les points ne doivent pas être tous définis dans le même plan ou se trouver sur le même rayon visuel. La résolution se fait par une technique minimisant un écart quadratique des distances de projection qui n'est autre qu'un calcul de pseudo inverse :

$$\mathbf{X} = \left(\mathbf{A}^\top \mathbf{A}\right)^{-1} \mathbf{A}^\top \mathbf{B} = \mathbf{A}^\dagger \mathbf{B}. \tag{1.14}$$

Une fois les paramètres du modèle global déterminés, il est possible de dissocier les paramètres du modèle interne et ceux du modèle externe. Pour que cette méthode soit suffisamment stable, n doit être assez grand, et l'incertitude sur la position des points assez faible.

1.4 Expérimentation

Ce travail se fera en deux étapes :

1. Acquisition d'une image avec le robot e-puck
2. Traitement de l'image pour étalonnage sous Matlab.

1.4.1 Acquisition d'une image avec le robot e-puck

Récupérez (avec **git clone**) le code nécessaire à contrôler le e-puck, depuis internet : **umrob** > **epuck-client**.

Placez l'e-puck et le cube d'étalonnage comme dans la Fig. 1.2. Suivez les instructions du document **mode_emploi_epuck_etudiants.pdf** qui est dans le repertoire `epuck-client`. Par exemple, si vous utilisez le robot avec adresse ip 192.168.1.2 :

```
$ ping 192.168.1.2
$ cd epuck-client/build
$ cmake ..
$ make
$./apps/epuck-app setWheelCmd 192.168.1.2 0 0
```

Ce programme permet de sauvegarder les images acquises par l'e-puck dans le répertoire `epuck-client/logs/<dateetheure>`. Pendant que le programme tourne, déplacez le cube avec la pince jusqu'à'à avoir une vue correcte des trois plans dont vous aurez besoin pour l'étalonnage (comme l'image de la Fig. 1.3). A ce moment là, appuyez sur `ctrl+c` pour arrêter le programme. Attention à ne plus déplacer ni le robot ni le cube par la suite !

Mesurez à l'aide du papier millimétré et éventuellement d'une règle la position ${}^{C}\mathbf{p}_W$ du cube dans le repère caméra, ainsi que la position ${}^{R}\mathbf{p}_W$ du cube dans le repère robot $\mathcal{R}_R$. Le repère robot a origine R et axes ${}^{R}X$ et ${}^{R}Y$ au sol, et ${}^{R}Z$ perpendiculaire au sol. Le point R se situe au milieu de deux roues du robot, ${}^{R}X$ pointe à droite et ${}^{R}Y$ vers l'avant.

Notez sur papier ces 6 valeurs (en mètres). Récupérez sur une clé USB depuis `epuck-client/logs/<dateetheure>` la meilleure image du cube (correspondante à ces positions), par exemple `image0093.png`, pour continuer le travail sur un PC avec Matlab.

1.4.2 Traitement de l'image et étalonnage sous Matlab

La suite se fait sous Matlab avec l'image du cube acquise auparavant par l'e-puck. Vous utiliserez 3 programmes : `saisie_2D_3D.m`, `calcul_modele.m` et `etalonnage.m`. Vous modifierez ces programmes aux lignes avec le commentaire `%MODIFIER`

1. Commencez par lancer le programme `saisie_2D_3D.m`. Vous n'avez pas besoin de modifier ce programme.
2. Le programme propose de charger une image (fonction `imread()`).

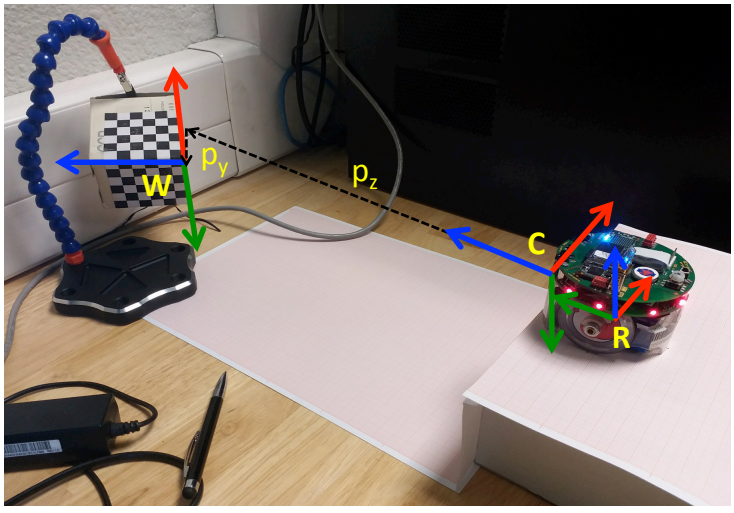

FIGURE 1.2 – Plateforme experimentale pour l'étallonage, avec les repères caméra, robot et monde (ou cube). Dans cette figure et dans les suivantes, les repères sont representés avec la convention couleur RGB (axe X en rouge, Y en vert et Z en bleu) ; p_y et p_z sont les translations permettant de passer de l'origine du repère caméra à l'origine du repère monde (p_x est négligeable dans la configuration de cette figure).

Choisissez l'image acquise par le e-puck (par exemple, `image0093.png`).

3. Sélectionnez en cliquant sur l'image (fonction `ginput()`) 12 points sur le cube, en suivant l'ordre de la Fig. 1.3 (gauche). Ensuite le programme affiche les points (croix rouges). Fermez la figure Matlab.
4. Puisque les coordonnées 3D des 12 points sont connues (et fournie dans la matrice `xyz`), le programme construit un fichier d'étalonnage appelé `uv_xyz.dat` qui contient les données de chacun des 12 points au format :

u (pixels)	v (pixels)	${}^W X$ (m)	${}^W Y$ (m)	${}^W Z$ (m)

5. Lancez `calcul_modele.m`. L'objectif de ce programme (à compléter) est de charger le fichier `uv_xyz.dat` et de l'utiliser pour estimer les 11 paramètres $c_{11} \dots c_{33}$ de la caméra. L'estimation se fera grâce à la méthode des moindres carrés décrite dans la Section 1.3.4.
6. Modifiez le programme pour construire les matrices $\mathbf{A}$ et $\mathbf{B}$ à partir des équations (1.12) et (1.13). Attention : ces matrices devront avoir respectivement dimension 24×11 et 24×1, ce qui n'est pas le cas dans la version initiale du programme.
7. Une fois que vous avez défini les bonnes matrices (vérifiez avec l'impression à la console ou dans le workspace de Matlab), le programme estime (grâce à la fonction `pinv()`) et affiche les 11 paramètres $c_{11} \dots c_{33}$ du modèle complet (1.4). Le modèle est sauvé dans le workspace (matrice $\mathbf{C}$).
8. Lancez depuis la console Matlab, la fonction `etalonnage(C)`. Cette fonction doit, à partir de la matrice $\mathbf{C}$, estimer les paramètres intrinsèques et extrinsèques du système et afficher les repères $\mathcal{R}_\mathcal{C}$ et $\mathcal{R}_\mathcal{W}$. Modifiez le programme afin de calculer ces paramètres en utilisant les équations (1.6) et (1.7).

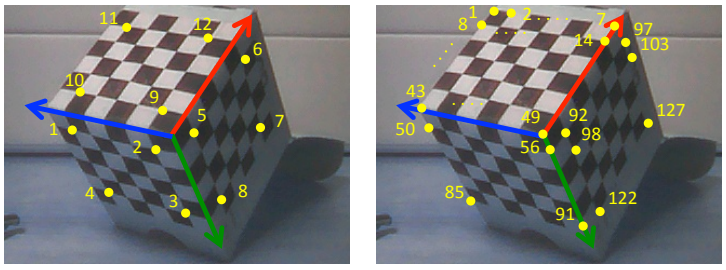

FIGURE 1.3 – Cube d'étalonnage avec le répère monde et l'ordre de saisie des points. Il va falloir cliquer sur les points indiqués, en respectant l'ordre. Gauche : on utilise 12 points, droite : on utilise 127 points.

9. A partir de la translation ${}^{C}\mathbf{p}_{W} = [p_x\ p_y\ p_z]^{\top}$ entre $\mathcal{R}_{\mathcal{C}}$ et $\mathcal{R}_{\mathcal{W}}$ (trouvée après cette procédure d'étalonnage) et de la translation ${}^{R}\mathbf{p}_{W}$ (mesurée en Sect. 2.3.1), calculez la translation ${}^{C}\mathbf{p}_{R}$ entre $\mathcal{R}_{\mathcal{R}}$ et $\mathcal{R}_{\mathcal{C}}$ (coordonnées de la caméra dans le repère robot). Notez ces valeurs, qui (avec les intrinsèques u_0, v_0, α_u et α_v) seront nécessaires pour la suite.

Vérification des résultats :

— Paramètres intrinsèques : puisque la résolution de la caméra est de 320×240 pixels, les valeurs de u_0 et de v_0 seront de l'ordre de grandeur d'une centaine de pixel ; α_u et α_v ont aussi ordre de grandeur 10^2 pixels.

— Paramètres extrinsèques : pour ce qui est des composantes de la translation ${}^{C}\mathbf{p}_{W} = [p_x\ p_y\ p_z]^{\top}$ entre $\mathcal{R}_{\mathcal{C}}$ et $\mathcal{R}_{\mathcal{W}}$ vous devriez retrouver des valeurs proches de celles mesurées et notées en Sect. 2.3.1. Pour les rotations, vous devriez voir s'afficher (après avoir changé le point de vue de la figure Matlab) des repères comme dans la figure du cube (voir Fig. 1.4, droite).

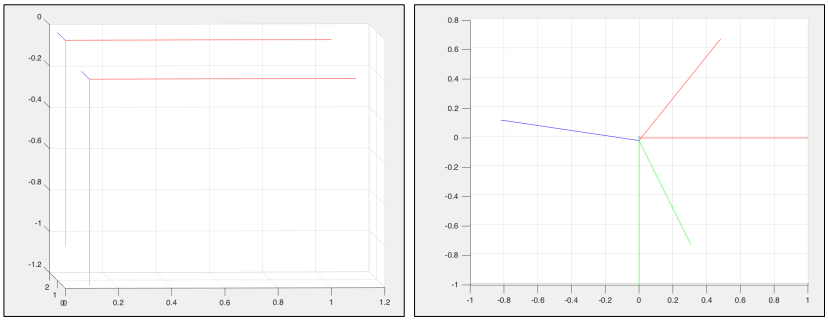

FIGURE 1.4 – Repères $\mathcal{R}_{\mathcal{C}}$ et $\mathcal{R}_{\mathcal{W}}$ affichés par la fonction `etalonnage(C)` de Matlab. Gauche : affichage initial (avant que vous ayez modifié la fonction). Droite : affichage après avoir bien modifié la fonction : notez que le repère $\mathcal{R}_{\mathcal{W}}$ est orienté comme dans la Fig. 1.3.

Questions bonus

1. Répétez le tout avec tous les 127 coins entre carrés noirs et blancs. D'abord relancez `saisie_2D_3D.m` en posant `manyPoints=true`. Ensuite, il faudra cliquer sur les 127 points en respectant l'ordre de la Fig. 1.3(droite). La précision de l'étalonnage devrait augmenter en utilisant plus de points.

2. Re-projeter les 12 (ou 127) points et les dessiner dans l'image du cube, en utilisant (1.4). Vous pourrez par exemple les dessiner en bleu pour vérifier qu'ils soient proches des points rouges sur lesquels vous avez initialement cliqué.

Chapitre 2

Poursuite de cible - Tracking

2.1 Objectifs

Vous allez implanter un algorithme de poursuite de cible (en anglais "tracking") pour que le robot e-puck puisse estimer sa position (se localiser) par rapport à une cible (le barycentre d'un "blob" noir sur fond blanc) vu par sa caméra. L'estimation doit être suffisamment rapide pour être exploitée dans une loi de commande en boucle fermée et doit fonctionner lorsque le robot et/ou la cible bougent (en gros, si il existe un mouvement relatif quelconque entre les deux). Vous aurez besoin des paramètres d'étalonnage obtenus dans la première séance du projet, et vous exploiterez ces résultats pour la suite du projet e-puck.

2.2 Matériel

— un robot e-puck avec sa caméra,
— une feuille avec la cible imprimée,
— une feuille de papier millimétré permettant de quantifier la précision de localisation,
— une machine linux pour enregistrer les images acquises par le robot et pour les traiter afin de suivre la cible.

2.3 Expérimentation

Ce travail se fera en quatre étapes :

1. Enregistrement d'une séquence d'images de la cible, avec le robot e-puck en mouvement.

2. Développement d'un algorithme pour détecter le barycentre de la cible sur chaque image. Cet algorithme sera développé sur les images enregistrées (donc sans avoir besoin du robot). Cet algorithme devra afficher abscisse et ordonnée du barycentre de la cible (en coordonnées pixel de l'image).
3. Projection du barycentre dans le repère robot, en utilisant les paramètres étalonnage, et à partir de cela : localisation du robot par rapport à la cible.
4. Validation de l'algorithme sur le robot en mouvement et sur la cible en mouvement.

2.3.1 Acquisition d'une séquence d'images avec le robot e-puck

Placez l'e-puck devant la cible (à 32 cm, pose indiquée sur le papier millimétré) pour que la cible soit visible par la camera. Suivez les instructions du document **mode_emploi_epuck_etudiants.pdf**. Par exemple, si vous utilisez le robot avec adresse ip 192.168.1.2, dans une fenêtre de terminator lancez :

```
$ ping 192.168.1.2
```

et dans une autre :

```
$ cd epuck-client/build
$ make
$./apps/epuck-app setWheelCmd 192.168.1.2 150 150
```

Ce programme permet de sauvegarder les images acquises par l'e-puck dans le répertoire `epuck-client/logs/<dateetheure>`. Pendant que le programme tourne, le robot avance vers la cible, avec vitesse 150 sur chaque roue.

2.3.2 Traitement des images pour détécter le barycentre de la cible

L'objectif de cette étape est de développer un algorithme pour détecter le barycentre de la cible sur chaque image. Cet algorithme sera testé sur les images enregistrées auparavant (donc sans avoir besoin du robot), et ensuite (étape 2.3.4) validé sur le robot en mouvement.

L'algorithme devra afficher à la console les valeurs en coordonnées pixel du barycentre de la cible, et dessiner sur l'image un cercle correspondant

au barycentre. Il devra aussi calculer l'aire de la cible, qui sera utilisée pour estimer la distance robot-cible. Vous allez développer cet algorithme en C++ dans la fonction `ProcessImageToGetBarycenter` qui est dans le fichier `controlFunctions.cpp` (dans `epuck-client/src/epuck`).

Il existe déjà une squelette simplifiée de la fonction. Pour la tester sans robot, lancez – toujours depuis le répertoire `epuck-client/build` :

```
$./apps/imgProcessing-app ../logs/<dateetheure>/
```

avec `<dateetheure>` indiquant la séquence sur laquelle vous souhaitez travailler (celle que vous venez d'enregistrer en 3.1). N'oubliez pas le `/` à la fin de la commande ! L'application affiche une par une les images contenues dans le répertoire `epuck-client/logs/<dateetheure>`. Vous verrez apparaitre les fenêtres suivantes.

- L'image en couleur enregistrée (fenêtre `loggedImg`).
- La même image en noir et blanc (fenêtre `greyImg`).
- La même image après traitement (fenêtre `processedImg`). Elle est initialement identique à l'image noir et blanc car dans la squelette actuelle, on n'effectue aucun traitement : les images sont juste converties en noir et blanc, et par défaut le pixel central est considéré comme barycentre de la cible.
- La position du robot dans le repère monde qui a : origine sur la cible, Y parallèle au mur, et X perpendiculaire au mur sortant (fenêtre `map`). La position du robot rouge est estimée en utilisant uniquement les encodeurs, tandis que la position du robot noir est estimée en utilisant uniquement la caméra. Initialement, la position du robot noir ne change pas, car dans la squelette initiale aucun traitement n'est effectué.

Pour passer d'une image à la suivante : sélectionnez avec la souris la fenêtre nommée `<processedImg>`, ensuite appuyez sur une touche quelconque du clavier pour passer aux images suivantes. La fonction affiche aussi le temps de calcul (en millisecondes) nécessaire à traiter l'image.

Il faudra modifier la fonction `ProcessImageToGetBarycenter` là où est le commentaire `//TODOM2`. Pour cela, adaptez le bout de code simple développé en C++ pour OpenCV3 ici : https://www.learnopencv.com/find-center-of-blob-centroid-using-opencv-cpp-python/. Pour adapter ce code, tenez compte du fait que dans `ProcessImageToGetBarycenter` :

- `greyImg` est votre image source, en noir et blanc, soit l'*entrée* de votre algorithme.
- `processedImg` est une copie de `greyImg` sur laquelle vous pouvez faire vos traitement et dessiner des amers pour débogage (notamment, le cercle correspondant au barycentre)

— `targetBarycenter` doit contenir le barycentre de la cible. Ce sera donc une des deux *sorties* de votre algorithme.
— `areaPix` doit contenir la surface (en pixel) de la cible – équivalent au moment m_{00}. Ce sera la deuxième *sortie* de votre algorithme.

Remarque : en changeant le dernier argument que vous passez à la fonction OpenCV `threshold`, vous pourrez inverser l'image...

A chaque fois que vous modifiez `ProcessImageToGetBarycenter`, n'oubliez pas de compiler, et ensuite de relancer l'application :

```
$make
$./apps/imgProcessing-app  ../logs/<dateetheure>/
```

Une fois que le barycentre est bien détecté sur chaque image de la séquence, et cela avec des temps de calcul "raisonnables", vous pouvez passer à l'étape suivante.

2.3.3 Localisation du robot par rapport à la cible

On note $\left({}^{C}X_B, {}^{C}Y_B, {}^{C}Z_B\right)$ les coordonnées du barycentre B dans le repère caméra (orange en Fig. 2.1). Dans le cas où la cible est parfaitement parallèle au plan image, la relation suivante permet de calculer la profondeur de la cible (en mètres) dans le repère caméra :

$$ {}^{C}Z_B = \sqrt{\frac{A_m \alpha_u \alpha_v}{A_p}} \tag{2.1} $$

Avec A_m la surface de la cible en mètres, A_p la surface de la cible en pixels, α_u et α_v les paramètres intrinsèques de la caméra que vous avez trouvé dans le Chapitre 1. Même si l'hypothèse de parallelisme n'est pas parfaitement verifiée, on peut utiliser (2.1) pour estimer ${}^{C}Z_B$. Ensuite, à partir de l'équation (1.2) on a :

$$ \begin{cases} u_B S = u_B \, \lambda {}^{C}Z_B = \lambda \left(\alpha_u \, {}^{C}X_B + u_0 \, {}^{C}Z_B\right) \\ v_B S = v_B \, \lambda {}^{C}Z_B = \lambda \left(\alpha_v \, {}^{C}Y_B + v_0 \, {}^{C}Z_B\right). \end{cases} \tag{2.2} $$

On en déduit :

$$ \begin{cases} {}^{C}X_B = \frac{(u_B - u_0) \, {}^{C}Z_B}{\alpha_u} \\ {}^{C}Y_B = \frac{(v_B - v_0) \, {}^{C}Z_B}{\alpha_v}, \end{cases} \tag{2.3} $$

avec u_0 et v_0 les paramètres intrinsèques de la caméra que vous avez trouvé au Chapitre 1.

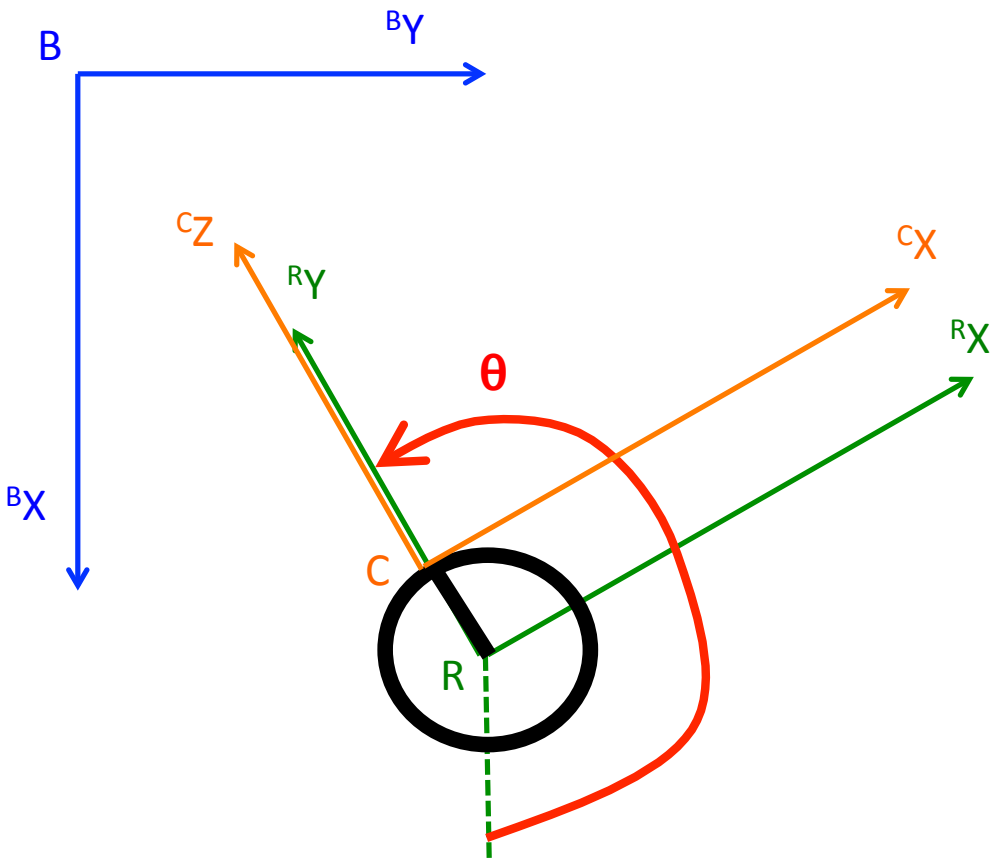

FIGURE 2.1 – Repères robot (vert), caméra (orange) et barycentre (bleu).

On peut ensuite trouver les coordonnées du barycentre dans le repère robot (voir Fig. 1.2 et aussi le repère vert en Fig. 2.1) :

$$\begin{cases} {}^{R}X_B = {}^{C}X_B + {}^{R}p_{x,C} \\ {}^{R}Y_B = {}^{C}Z_B + {}^{R}p_{y,C}. \end{cases} \tag{2.4}$$

En utilisant l'orientation θ du robot, estimée via les encodeurs, on peut enfin calculer les coordonnées du robot dans le repère du barycentre (voir Fig. 2.1) :

$$\begin{pmatrix} {}^{B}X_R \\ {}^{B}Y_R \end{pmatrix} = \begin{pmatrix} \sin\theta & \cos\theta \\ -\cos\theta & \sin\theta \end{pmatrix} \begin{pmatrix} -{}^{R}X_B \\ -{}^{R}Y_B \end{pmatrix}. \tag{2.5}$$

Il s'agit d'utiliser ces équations pour compléter `GetCurrPoseFromVision` afin qu'elle calcule la pose du robot. Vous allez modifier la fonction là où est le commentaire `//TODOM2`. Dans cette fonction :

— `baryctr` est le barycentre de la cible, calculé à l'étape précédente. C'est la première *entrée* de la fonction. Ses champs `baryctr.x` et `baryctr.y` correspondent respectivement à u_B et v_B.
— `theta` est l'orientation θ du robot (voir Fig. 2.1) estimée par les encodeurs. C'est la deuxième *entrée* de la fonction.
— `areaPix` est la surface de la cible, calculée à l'étape précédente, et notée A_p. C'est la troisième *entrée* de la fonction.
— `data_folder` indique le dossier où la fonction doit sauvegarder la pose.
— `curRobPose` est la pose $\left({}^B X_R, {}^B Y_R, \theta\right)$ du robot par rapport à la cible. Ce sera la seule *sortie* de la fonction.

Songez à mettre des affichages dans le terminal (`printf` ou `cout`) pour vérifier les différentes étapes de la fonction.

2.3.4 Validation de l'algorithme sur le robot

Placez la cible devant l'e-puck pour qu'elle soit visible par la camera du robot. Si vous changez la postion initiale du robot, pensez à la modifier aussi dans le `main` (variable `initPose`).

De nouveau, si vous utilisez le robot avec adresse ip 192.168.1.2, dans une fenêtre de terminator lancez :

```
$ ping 192.168.1.2
```

et dans une autre :

```
$ cd epuck-client/build
$ make
$./apps/epuck-app setWheelCmd 192.168.1.2 150 150
```

Effectuez plusieurs tests : avec des vitesses des roues différentes (par example, 50, 60 pour faire pivoter le robot) et en bougeant la cible.

Chapitre 3

Asservissement visuel

3.1 Objectifs

L'objectif de cette partie du projet est de réaliser un asservissement visuel du e-puck, afin qu'il centre dans son image la cible détectée dans le Chapitre 2. A cette fin, vous contrôlerez les vitesses linéaire v et angulaire ω du robot, en fonction de la position du barycentre de la cible dans l'image (u_B, v_B). Le but sera de régler (u_B, v_B) jusque à une consigne (u_B^*, v_B^*).

3.2 Matériel

— un robot e-puck avec sa caméra,
— une feuille avec la cible imprimée,
— une machine linux pour vérifier la loi de commande sur les images enregistrées, avant de la mettre en place sur le vrai robot.

3.3 Expérimentation

Vous appliquerez la loi de commande suivante (voir : transparents de cours **asservissement_visuel**) :

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = -\lambda \mathbf{L}^{\dagger} \left(\begin{bmatrix} {}^{I}X_B \\ {}^{I}Y_B \end{bmatrix} - \begin{bmatrix} {}^{I}X_B^* \\ {}^{I}Y_B^* \end{bmatrix} \right) \tag{3.1}$$

avec :
— v et ω les vitesses linéaire et angulaire du robot e-puck (évoquées dans la Sect.3.3.1)
— λ un gain scalaire positif,

— $\left({}^{I}X_B, {}^{I}Y_B\right)$ les coordonnées courantes (en mètres) du barycentre de la cible dans le repère image,
— $\left({}^{I}X_B^*, {}^{I}Y_B^*\right)$ les coordonnées désirées (en mètres) barycentre de la cible dans le repère image,
— $\mathbf{L}$ la matrice d'interaction reliant $\left({}^{I}X_B, {}^{I}Y_B\right)$ et la vitesse de la caméra dans l'espace libre (qui a dimension 6) :

$$\mathbf{L} = \begin{bmatrix} -\frac{1}{{}^{C}Z_B} & 0 & -\frac{{}^{I}X_B}{{}^{C}Z_B} & {}^{I}X_B\,{}^{I}Y_B & -1-{}^{I}X_B^2 & {}^{I}Y_B \\ 0 & -\frac{1}{{}^{C}Z_B} & \frac{{}^{I}Y_B}{{}^{C}Z_B} & 1+{}^{I}Y_B^2 & -{}^{I}X_B\,{}^{I}Y_B & -{}^{I}X_B \end{bmatrix}. \tag{3.2}$$

La matrice $\mathbf{L}$ vérifie la condition suivante :

$$\begin{bmatrix} {}^{I}\dot{X}_B \\ {}^{I}\dot{Y}_B \end{bmatrix} = \mathbf{L} \begin{bmatrix} v_{X,C} \\ v_{Y,C} \\ v_{Z,C} \\ \omega_{X,C} \\ \omega_{Y,C} \\ \omega_{Z,C} \end{bmatrix}, \tag{3.3}$$

avec $[v_{X,C} \ldots \omega_{Z,C}]$ vitesse de la caméra dans l'espace libre.

Donc, la loi de commande (3.1) garantit la convergence asymptotique de $\left({}^{I}X_B, {}^{I}Y_B\right)$ à $\left({}^{I}X_B^*, {}^{I}Y_B^*\right)$. Le taux de convergence est déterminé par λ.

Sur le robot e-puck, la caméra ne peut pas se déplacer librement dans l'espace. Elle est contrainte et n'a que 2 dégrées de liberté.

Question 1 Ré-écrivez la matrice $\mathbf{L}$ pour prendre en compte uniquement ces 2 dégrées de liberté.

Dans cette partie du projet, on considère confondus l'origine du repère caméra et celle du repère robot.

Question 2 Sous cette hypothèse, écrivez les vitesses v et ω du robot en fonction des vitesses de sa caméra.

Question 3 Re-écrivez la loi de commande (3.1) en prenant en compte vos réponses aux Question 1 et 2.

Il s'agira maintenant de mettre en place cette loi de commande sur le e-puck. Pour cela, vous devez modifier `ControlRobotWithVisualServoing` qui est dans le fichier `ControlFunctions.cpp`. Cette fonction prend en entrée les coordonnées image du barycentre de la cible (u_B, v_B) (`baryc` dans le code) et doit calculer en sortie les vitesses du robot : v et ω (notées `vel` et `omega` dans le code). Dans le programme, vous pouvez donner une valeur constante (et réaliste) à ${}^{C}Z_B$. Pour exprimer les coordonnées pixels en mètres (afin d'obtenir ${}^{I}X_B^*$ et ${}^{I}Y_B^*$ en fonction de u_B et v_B), vous utiliserez les valeurs de u_0, v_0, α_u et α_v obtenues par étalonnage, ainsi que la valeur de la focale en

mètres de la caméra du e-puck qui vaut (d'après le data sheet du robot) 1.79 mm.

Une fois préparée la fonction, vous pouvez la tester sur des images enregistrées (pour éviter d'endommager le robot). Pour cela, rajoutez l'appel à la fonction dans `epuck-client/apps/imgProcessing-app/main.cpp`, après le calcul de `baryc` :

```
float vel, omega;
struct timeval startTime;
ControlRobotWithVisualServoing(baryc, vel, omega);
```

Pour vérifier si les valeurs de v et ω vous paraissent raisonnables en terme de signe et d'ordre de grandeur, rajoutez des affichages dans le terminal (`printf` ou `cout`). Ensuite testez sur les images enregistrées :

```
$ cd epuck-client/build
$ make
$./apps/imgProcessing-app ../logs/<dateetheure>/
```

Et enfin, si v et ω vous paraissent raisonnables, sur le vrai robot :

```
$./apps/epuck-app visServo 192.168.1.2
```

Question 4 Utilisez v pour asservir non pas la position de B dans l'image, mais sa profondeur ${}^{C}Z_{B}$. Cette variable évolue avec la vitesse linéaire v du robot suivant :

$$ {}^{C}\dot{Z}_{B} = -v \tag{3.4} $$

Si on note $\omega = \omega\left({}^{I}X_{B}, {}^{I}Y_{B}, {}^{I}X_{B}^{*}, {}^{I}Y_{B}^{*}, {}^{C}Z_{B}\right)$ la 2e équation du contrôleur (3.1), vous pouvez contrôler le robot avec la loi de commande suivante :

$$ \begin{cases} v = \lambda_{z}\left({}^{C}Z_{B} - {}^{C}Z_{B}^{*}\right) \\ \omega = \omega\left({}^{I}X_{B}, {}^{I}Y_{B}, {}^{I}X_{B}^{*}, {}^{I}Y_{B}^{*}, {}^{C}Z_{B}\right), \end{cases} \tag{3.5} $$

avec λ_{z} un gain positif (pas forcément identique à λ) et ${}^{C}Z_{B}^{*}$ la profondeur désirée. Modifiez la fonction `ControlRobotWithVisualServoing` pour réaliser (3.5). Pour mettre à jour ${}^{C}Z_{B}$, il va falloir passer en paramètre de la fonction aussi `areaPix` (et donc modifier aussi `ControlFunctions.h`).