Développement d’un démonstrateur de bras manipulateur intelligent

MAHAUT Raphaël - LASTENNET Louis - CONSTANT Evrard - PRETET Thibault - DANESI Nathaël

I – Introduction

I.1 – Contexte

Dans la quasi-totalité des secteurs industriels, les matières premières sont livrées ou extraites « en vrac » et sont donc inutilisables telles quelles. Une tâche de « dévracage » est alors nécessaire pour ordonner et trier ces pièces et matériaux afin de les rendre exploitables.

Cependant, employer des moyens humains pour effectuer cette tâche n'est pas forcément la meilleure solution. En effet, l'emploi d'opérateurs pour effectuer le tri et le dévracage de matériaux ou de pièces en milieu industriel pose de nombreux problèmes :

- Ce travail manuel peut présenter un fort risque d'engendrer des troubles musculosquelettiques (TMS) chez l'opérateur.
- Dans certains cas, l'emploi d'humains pour ces tâches est impossible : manipulation de matériaux dangereux, récupération en zone contaminée, tri de pièces trop grandes, etc.
- Ce travail est répétitif mentalement, ce qui engendre une perte de motivation et donc de productivité chez l'opérateur ainsi que des erreurs de tri.

Ainsi, il peut être intéressant que le tri et le dévracage soient réalisés par un robot. Il est donc pertinent de se pencher sur les problématiques suivantes dans le cadre de ce projet :

I.2 – Problématiques

- Comment effectuer efficacement le dévracage de pièces en secteur industriel ?
- Comment réaliser un tri des pièces en même temps que ce dévracage ?
- Comment organiser les pièces selon une structure donnée au fur et à mesure du dévracage ?

I.3 – Objectifs et démarche

L'objectif du projet est de réaliser un démonstrateur de bras manipulateur intelligent (voir Figure 1) capable dans un premier temps de réaliser un dévracage de pièces parallélépipédiques en bois disposées sur son espace de travail.

Afin de réaliser ce dévracage, le bras robotique sera muni d'une caméra montée sur un support. Il sera nécessaire dans un premier temps d'évaluer et, si nécessaire, de corriger la distorsion de cette caméra. Ensuite, il s'agira d'effectuer un traitement des images que cette dernière récupère, afin d'identifier la position des pièces que le robot devra trier. Ce traitement devra, pour assurer une efficacité maximale, être réalisé en temps réel : il est donc pertinent de le soumettre à des outils d'intelligence artificielle préalablement entraînés qui garantissent la rapidité de traitement.

Par ailleurs, le programme de traitement d'image retournera la position des pièces en bois dans le repère de la caméra. Il sera donc nécessaire de réaliser une double action pour rendre ces coordonnées utilisables par le robot : d'une part réaliser un changement de repère depuis celui de la caméra vers celui du robot, et d'autre part convertir ces coordonnées cartésiennes en positions angulaires des moteurs du robot. Cette dernière conversion s'appuie sur le modèle géométrique inverse du bras robotique, qu'il sera nécessaire d'établir.

Il faudra ensuite réaliser un programme de contrôle qui permettra de définir les instructions à donner au robot pour pouvoir réaliser la tâche de dévracage. Enfin il s'agira de relier entre elles les différentes composantes du code grâce au middleware ROS p our assurer la mise en mouvement du robot.

Figure 1 : bras robotique à disposition du groupe
Modèle : Openmanipulator-X Robotis RM-X52-TNM

II – Vision par ordinateur des pièces à ordonner

II.1 – Caméra (cas 2D)

Le groupe dispose de deux caméras Urban factory, de modèle « Webcam autofocus USB ». Ces caméras présentent une forte distorsion qui semble non négligeable, et il est nécessaire de leur construire un support fixe n'entravant pas les mouvements du bras robotique.

Dès à présent et pour la suite de ce rapport, nous considérons que les objets à trier par le robot sont situés à la même altitude, de sorte qu'utiliser une seule caméra avec un angle de vue normal au plan de travail du bras robotique est suffisant.

II.1.a – Conception du support

Le support de la caméra doit permettre de la placer avec un angle de vue normal au plan de travail du robot, coupant ledit plan de travail en un point proche de son centre, et ce tout en entravant au minimum les déplacements du bras. Un support s'adaptant au mieux à la demi-sphère de balayage du bras robotique paraît alors pertinent. Les plans de ce support sont représentés sur la figure 2 ci-dessous.

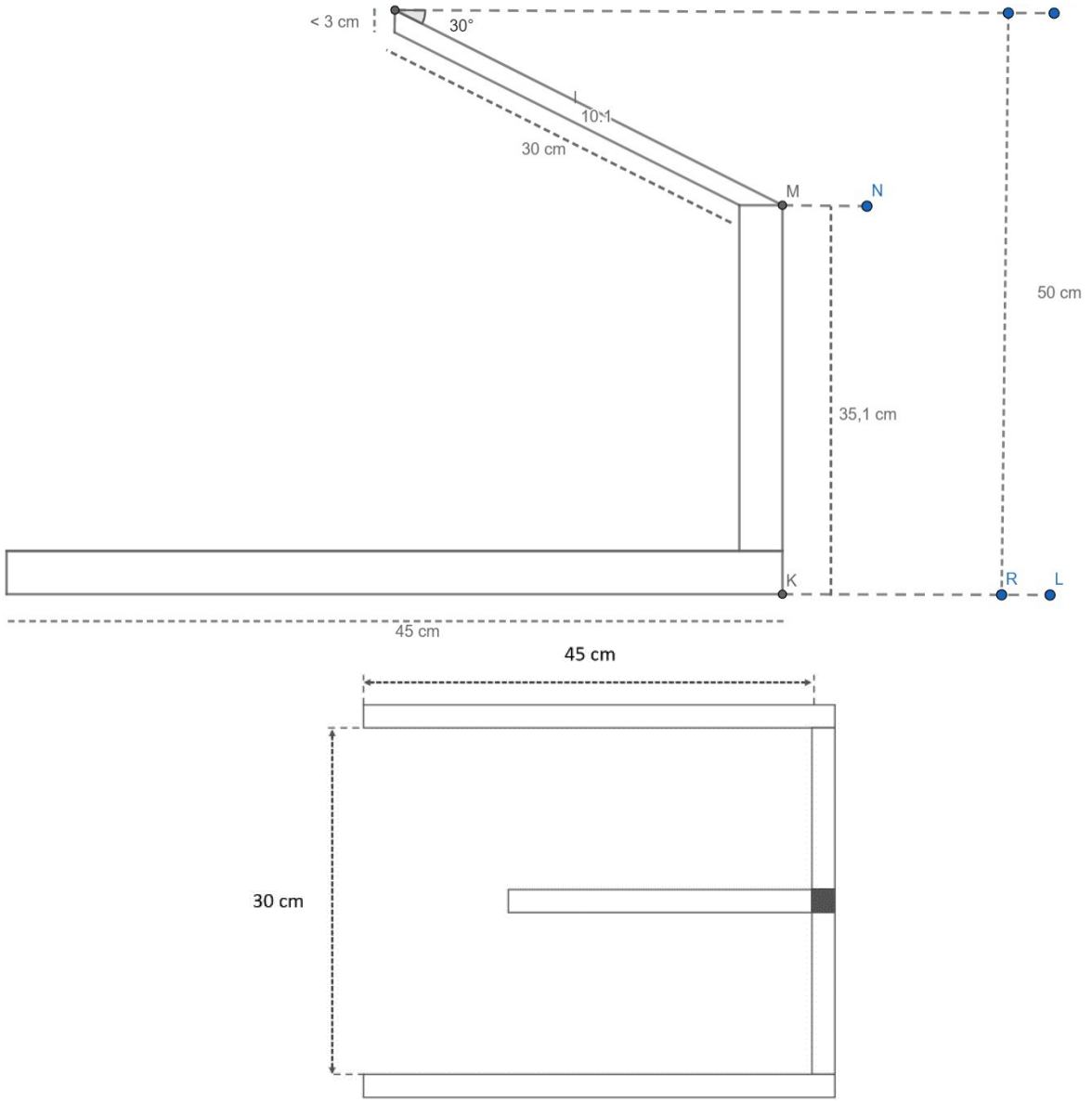

Figure 2 : schémas du support de la caméra ; en haut : vue de côté ; en bas : vue de dessus

II.1.b – Communication avec la caméra, bibliothèque cv2

Afin de récupérer les images recueillies par la caméra et de les traiter, l'utilisation de la bibliothèque graphique de renom OpenCV (Open Computer Vision) paraît tout indiquée. Cette dernière dispose d'une adaptation en une bibliothèque Python nommée cv2, qu'il est possible d'installer et d'importer via les commandes respectives suivantes :

```
pip install opencv-python
import cv2
```

La bibliothèque cv2 permet d'accéder à la vision de la caméra via un objet particulier : une « capture ». Une capture peut être créée et stockée dans une variable grâce à la fonction

`cv2.VideoCapture()`. Une fois la capture ouverte de la sorte, il est possible de prendre des photos par son intermédiaire en lui appliquant la méthode `read()`. Il est important de noter que sans l'appel à la fonction `cv2.waitKey()` immédiatement après la prise d'une photo, cette dernière sera entièrement grisée et inexploitable. Dès que la prise de photos est terminée, il est nécessaire de fermer la capture en lui appliquant la méthode `release()` (Voir code fourni en annexe).

En remarquant que, pendant que le bras robotique se déplace dans son espace de travail, il nuit à la vision de la caméra, il apparaît qu'une simple prise de photos lors des instants de repos du robot est suffisante, d'où la pertinence des captures présentées ci-dessus. Il sera cependant nécessaire de corriger la distorsion de la caméra pour pouvoir correctement exploiter les photos de la caméra.

II.1.c – Correction de la distorsion

La distorsion d'une caméra est une mesure de la déformation que subit une photographie lors de sa prise. Cette distorsion est caractérisée par la « matrice de caméra » et les « coefficients de distorsion » de la caméra, intrinsèques à cette dernière et définis comme suit :

$$camera\ matrix = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

Où f_x et f_y sont les distances focales des lentilles de la caméra et c_x et c_y sont les coordonnées de leurs centres optiques.

$$distorsion\ coefficients = (k_1 \quad k_2 \quad p_1 \quad p_2 \quad k_3)$$

Où les coefficients k_1, k_2 et k_3 sont responsables de la distorsion radiale, et les coefficients p_1 et p_2 sont liés à la distorsion tangentielle. Plus d'informations à ce sujet sont disponibles en [1].

Cette matrice de caméra et ces coefficients de distorsion sont constants pour une caméra donnée, il est ainsi suffisant de les mesurer une fois pour toutes et de conserver leurs valeurs. Il est nécessaire d'effectuer cette mesure afin de corriger la distorsion.

La mesure de la matrice de caméra et des coefficients de distorsion peut être effectuée grâce à la bibliothèque cv2 et à une série de photos d'un même échiquier prises avec la caméra [1] [2], comme détaillé dans le pseudo-code ci-dessous :

```
Lire liste_images_echiquier
Lire nb_coins   //nombre de coins internes de l'échiquier
liste_coins_internes ← []
Pour image_echiquier parcourant liste_images_echiquier faire :
|  Convertir image_echiquier en niveaux de gris
|  coins_internes ← cv2.findChessboardCorners(image_echiquier, nb_coins)
|  liste_coins_internes ← liste_coins_internes + [coins_internes]
Fin pour
matrice_camera, coefs_distorsion ← cv2.calibrateCamera(liste_coins_internes)
Ecrire matrice_camera
Ecrire coefs_distorsion
```

Le pseudo-code ci-dessus correspond au principe de la fonction `cameraMatrixRecover()` présente dans le code fourni en annexe. Dans cette même annexe, la fonction `liveCameraCalibration()` permet de prendre manuellement les photos de l'échiquier avec la caméra avant de les passer en arguments à la fonction `cameraMatrixRecover()` pour en déduire la matrice de caméra et les coefficients de distorsion. L'appel à la fonction `liveCameraCalibration()` avec la caméra Urban factory fournit les valeurs suivantes :

$$camera\ matrix = \begin{bmatrix} 426 & 0 & 332 \\ 0 & 426 & 198 \\ 0 & 0 & 1 \end{bmatrix}$$

$$distorsion\ coefficients = (-0{,}266 \quad 0{,}07 \quad -0{,}006 \quad 0{,}0008 \quad 0{,}0058)$$

Il est désormais possible d'utiliser ces valeurs pour faire appel à la fonction `cv2.undistort()` qui permet de corriger la distorsion d'une photographie prise avec la caméra.

II.1.d – Changement de repère

Maintenant que la distorsion de la caméra est corrigée, celle-ci va permettre de repérer la position des objets à trier sur le plan de travail du bras robotique. Pour fournir au robot les coordonnées de ces objets dans son propre repère, il est nécessaire d'effectuer un changement de repère depuis celui de la caméra.

D'après les dimensions du support de la caméra (voir II.1.a), il est possible de dessiner les repères respectifs de la caméra et du bras robotique, sachant que l'angle de vue de la caméra est normal au plan de travail du robot (Figure 3).

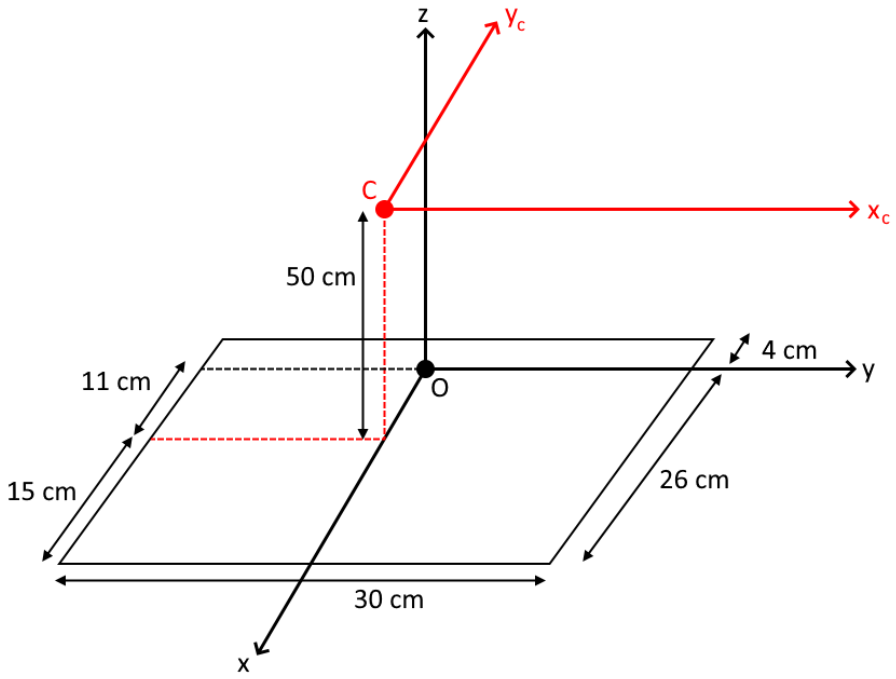

Figure 3 : schéma des repères respectifs du robot (O, x, y, z) et de la caméra (C, x_c, y_c)

En utilisant la figure 3, il est aisé d'établir la relation suivante pour tout point M de l'espace :

$$\begin{cases} x(M) = -y_c(M) + x(C) \\ y(M) = x_c(M) \end{cases}$$

Ce qui peut également s'écrire en termes de transformation homogène de la façon suivante :

$$\begin{bmatrix} x(M) \\ y(M) \\ 1 \end{bmatrix} = T_c^0 \begin{bmatrix} x_c(M) \\ y_c(M) \\ 1 \end{bmatrix} \; avec \; T_c^0 = \begin{bmatrix} 0 & -1 & x(C) \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_c(M) \\ y_c(M) \\ 1 \end{bmatrix}$$

Il ne reste alors plus qu'à déterminer $x_c(M)$ et $y_c(M)$ à partir d'une image de la caméra. C'est pour cette étape qu'il est crucial d'avoir corrigé la distorsion : la caméra sans distorsion peut être modélisée par une lentille mince de rayon très grand devant les dimensions du plan de travail du robot, ce qui permet de se placer dans l'approximation de Gauss des rayons lumineux.

Il en résulte qu'il est suffisant de connaître la distance réelle entre deux points A et B sur le plan de travail pour déduire, par une simple relation de proportionnalité, la distance réelle entre n'importe quels points sur une photo prise par la caméra. En notant x_p et y_p les équivalents en pixels mesurés sur la photo des coordonnées x_c et y_c, cette relation s'écrit pour tout point M de la façon suivante :

$$\begin{cases} x_c(M) = R \times x_p(M) \\ y_c(M) = R \times y_p(M) \end{cases} \; avec \; R = \frac{\sqrt{\left(x_c(B) - x_c(A)\right)^2 + \left(y_c(B) - y_c(A)\right)^2}}{\sqrt{\left(x_p(B) - x_p(A)\right)^2 + \left(y_p(B) - y_p(A)\right)^2}}$$

En termes de transformation homogène, la relation globale reliant x et y à x_p et y_p s'écrit donc :

$$\begin{bmatrix} x(M) \\ y(M) \\ 1 \end{bmatrix} = R \begin{bmatrix} 0 & -1 & x(C) \\ 1 & 0 & 0 \\ 0 & 0 & 1/R \end{bmatrix} \begin{bmatrix} x_p(M) \\ y_p(M) \\ 1 \end{bmatrix} \; avec \; R = \frac{\sqrt{\left(x_c(B) - x_c(A)\right)^2 + \left(y_c(B) - y_c(A)\right)^2}}{\sqrt{\left(x_p(B) - x_p(A)\right)^2 + \left(y_p(B) - y_p(A)\right)^2}}$$

On remarquera que la connaissance de la position des points O, A et B nécessite une calibration du système de vision par caméra. Cette calibration se fera de façon automatique et sera détaillée en II.4.

Il est ainsi possible de fournir au système de pilotage du bras robotique les coordonnées des objets à trier dans le repère de ce dernier. Il faut désormais effectuer un traitement des images recueillies par la caméra pour y détecter ces objets.

II.2 – Traitement d'images

Une image correspond à une matrice de pixels, c'est-à-dire une matrice de vecteurs de dimension 3 à valeurs dans $\{0, 1, \ldots, 255\}^3$. Un tel objet condense une grande quantité d'information et son analyse par un ordinateur s'avère immensément plus complexe que par un œil humain. C'est la raison pour laquelle une image doit être traitée pour être analysée correctement par un ordinateur.

Une technique puissante et classique de traitement d'images est la détection de contours. Elle permet de réduire une image à une matrice de réels valant tous 0 ou 1, où les coefficients valant 1 dessinent les contours visibles sur l'image originelle. Une fois cette opération effectuée, l'image des contours s'avère particulièrement pertinente pour diverses analyses, la recherche de formes par exemple.

Dès à présent et pour le reste de ce rapport, nous supposons que les objets à dévraquer par le bras robotique sont des planches de bois parallélépipédiques dont la couleur est suffisamment différente de celle du plan de travail du robot et importe peu. Nous chercherons à localiser ces planches sur une image au travers d'une détection de contours puis de formes.

II.2.a – Détecteur de contours de Canny

Afin de détecter les contours d'une image prise par la caméra, un algorithme moderne et puissant existe et paraît tout indiqué : l'algorithme de détection de contours de John F. Canny. Son fonctionnement simplifié se résume en les étapes ci-dessous :

1. L'image est convertie en niveaux de gris et un flou gaussien lui est appliqué.
2. Une approximation discrète du laplacien du niveau de gris est calculée en chaque pixel.
3. Élimination par « double palier » : les pixels dont le laplacien est suffisamment élevé sont considérés positionnés sur un contour, et ceux de laplacien trop faible sont éliminés.
4. Élimination par « hystérésis » : les pixels restants et trop éloignés des contours identifiés par double palier sont éliminés.

Cet algorithme est présent dans la bibliothèque cv2 sous la fonction `cv2.Canny()` qui prend en argument l'image à traiter et les deux paliers de l'étape d'élimination par double palier.

Il est important de remarquer que la détection des contours par un calcul de laplacien est très sensible au bruit, d'où l'intérêt du flou gaussien qui atténue ce bruit. Il est possible d'encore réduire ce bruit au préalable par une réduction du nombre de couleurs de l'image originelle suivie de l'application d'un flou gaussien, ce qui apparaît pertinent au vu du bruit présent sur les images recueillies par la caméra Urban factory.

Le principe global de la détection de contours appliquée aux images de la caméra (correspondant à la fonction `cannyEdgeDetector()` dans le code fourni en annexe) est explicité dans le pseudo-code ci-dessous :

```
Lire image
Lire k
Lire palier_bas
Lire palier_haut

// Réduction du nombre de couleurs pour atténuer le bruit de la caméra
Pour ligne parcourant image faire :
|  Pour pixel parcourant ligne faire :
|  |  Pour couleur parcourant pixel faire :
|  |  |  quotient ← quotient de la division euclidienne de couleur par k
|  |  |  couleur ← quotient * k
|  |  Fin pour
```

```
|  Fin pour
Fin pour
```

```
image ← cv2.GaussianBlur(image)   // Flou gaussien
image_contours ← cv2.Canny(image, palier_bas, palier_haut)
```

Une fois les contours détectés, il est alors possible de chercher ceux correspondant à la forme d'une planche de bois que le bras robotique doit déplacer.

II.2.b – Filtrage des contours

Afin de rechercher parmi les contours d'une image ceux qui correspondent à une forme rectangulaire, il est d'abord nécessaire d'isoler les différentes formes fermées dessinées par les contours. Une fonction de la bibliothèque cv2 permet de réaliser cette tâche lorsqu'elle est appliquée à l'image des contours : la fonction `cv2.findContours()`.

Il est important de mentionner une différence mathématique entre les contours (« edges » en anglais), déterminés par un maximum local de gradient ou de laplacien, et les frontières (« contours » ou « boundaries » en anglais), correspondant aux rebords fermés des objets et souvent calculées à partir des contours. La fonction `cv2.findContours()` recherche et isole les différentes frontières sur une image, ce qui est adapté pour de la détection de formes.

Une fois les différents contours fermés isolés, il est possible de les filtrer en fonction de plusieurs critères pertinents : leur aire et leur forme en particulier. Un contour fermé d'aire trop petite correspond à du bruit, tandis qu'un autre d'aire trop grande peut correspondre au plan de travail du robot par exemple : ces contours doivent être éliminés. De même, les contours qui dessinent une forme non-trapézoïde ne peuvent pas représenter une planche à trier et doivent être éliminés.

L'approximation de la forme d'un contour fermé est en réalité un problème complexe. Fort heureusement, la bibliothèque cv2 dispose de la fonction `cv2.approxPolyDP()` pour effectuer cette tâche : cette dernière permet de réduire un contour fermé à un tableau des coordonnées de ses sommets. Il devient alors possible, une fois cette opération réalisée, de ne conserver que les contours à 4 sommets pour observer les planches de bois sur le plan de travail du robot.

Bien entendu, ces conditions sur l'aire et le nombre de sommets ne suffisent en aucun cas à caractériser de façon unique les planches de bois à trier par le robot. Par exemple, un objet trapézoïde de taille semblable à celle des planches ne serait pas éliminé. Toutefois, en pratique, en supposant que le bras robotique ne doit trier que des planches, cette méthode de filtrage apparaît tout à fait convenable.

Dans le code fourni en annexe, la méthode de filtrage des contours décrite ci-dessus est mise en application par les fonctions `cleanContours()` et `findCentersGravity()`. Le principe de fonctionnement de ces deux fonctions est résumé dans les lignes de peuso-code suivantes :

```
Lire image_contours   //obtenue par détecteur de Canny
Lire aire_min
Lire aire_max
liste_planches ← []
liste_contours ← cv2.findContours(image_contours)
```

```
Pour contour parcourant liste_contours faire :
|  aire ← aire de contour
|  Si aire_max > aire et aire > aire_min faire :
|  |  liste_points ← cv2.approxPolyDP(contour)
|  |  Si liste_points contient précisément 4 éléments faire :
|  |  |  gravite ← centre de gravité des 4 points de liste_points
|  |  |  Ajouter gravite au début de liste_points
|  |  |  Ajouter liste_points à liste_planches
|  |  Fin si
|  Fin si
Fin pour
Ecrire liste_planches
```

Il est désormais possible d'obtenir les coordonnées des sommets des planches de bois et de leurs centres de gravité, ainsi que de les convertir en coordonnées utilisables par le bras robotique grâce au travail effectué en II.1.d.

II.2.c – Caméra bruitée et nécessité de multiples photos

Il reste un dernier problème à résoudre, dû au bruit important (mentionné en II.2.a) par lequel sont perturbées les acquisitions de la caméra : sur plusieurs photos successives de la même planche de bois, celle-ci peut parfois être détectée et parfois non, et deux détections successives de son centre de gravité peuvent différer légèrement.

Il en résulte qu'une unique photo peut ne pas suffire à détecter une planche. Il est donc nécessaire de prendre une salve de plusieurs photos, une dizaine par exemple, pour assurer la détection de toutes les planches de bois. Par conséquent, certaines planches vont être détectées sur plusieurs images : il est nécessaire de ne les prendre en compte qu'une seule fois.

Pour éviter les doublons, on garde simplement un tableau des planches détectées. Lorsqu'une planche est détectée parmi la dizaine d'images, on compare son centre de gravité et ses coins à ceux des planches déjà détectées : si ces 5 points correspondent (à une faible erreur près, comme toujours en informatique) entre la nouvelle planche et une déjà détectée, alors la nouvelle planche n'est pas si nouvelle et ne doit pas être prise en compte.

Le tableau final sans doublons peut ainsi être envoyé au bras robotique. Ce tableau présente une probabilité fortement accrue de contenir toutes les planches présentes sur le plan de travail.

II.2.d – Perspectives d'amélioration

Comme mentionné en II.2.b, l'algorithme détaillé précédemment ne permet de réaliser une détection convenable uniquement sous l'hypothèse que le robot ne doit dévraquer que des planches de bois identiques. Il est donc pour le moment incapable de différencier deux objets différents à trier, par exemple. Une approche possible pour pallier ce défaut serait d'approximer avec plus de précision la forme des différents objets sur le plan de travail du robot puis de les discriminer par rapport à cette forme plus précise et à leur couleur.

L'algorithme présente également un second défaut : le critère d'élimination des contours selon l'aire étant approximatif, il est impossible pour l'algorithme de distinguer une planche de bois d'un amalgame de 2 à 3 planches en contact formant un parallélépipède rectangle. Ceci entrave fortement la capacité du bras robotique à dévraquer des planches en contact (ou, pire encore, superposées). Une solution possible pourrait passer par une détection plus fine des contours et un travail avec les couleurs, pour chercher à séparer plusieurs planches trop proches.

II.3 – Capacité du bras robotique à saisir une planche

Il est important de remarquer que le bras robotique ne dispose, au niveau de son « poignet », que d'une unique liaison pivot au lieu de l'habituelle liaison rotule. Son incapacité à faire tourner son poignet l'empêche de saisir certaines planches de bois.

Il apparaît donc qu'il est nécessaire d'expliciter sous quelles conditions d'inclinaison le robot peut saisir une planche de bois. Une fois ces conditions déterminées, il sera possible de mesurer l'inclinaison d'une planche sur une image et de vérifier si le bras peut la saisir telle quelle ou si une action est nécessaire.

II.3.a – Condition de saisie d'une planche

L'emplacement optimal pour la pince du bras robotique voulant saisir une planche est au niveau du centre de gravité de cette dernière, les coordonnées duquel étant déterminées grâce aux algorithmes détaillés en II.2. La figure 4 ci-dessous illustre l'inclinaison limite qu'une planche saisissable par le robot peut avoir : l'objectif est de déterminer l'angle α qui y est illustré.

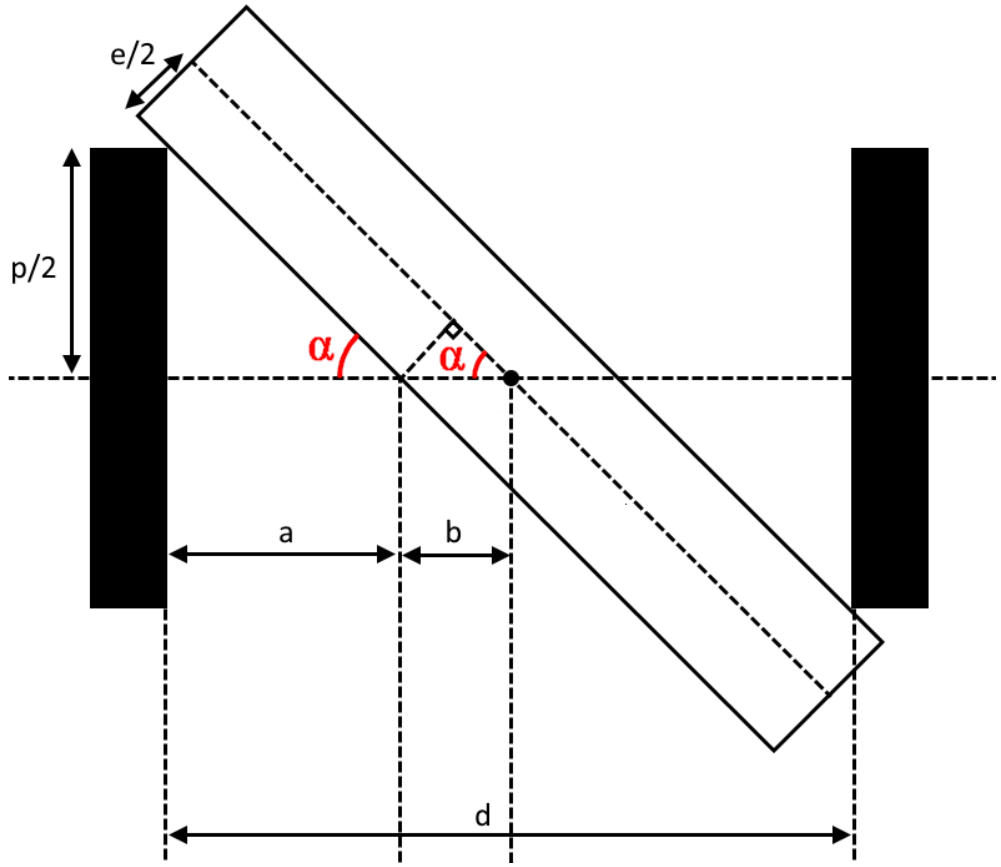

Figure 4 : schéma d'une planche de bois d'inclinaison limite saisissable par la pince du bras robotique

Une relation évidente apparaît sur la Figure 4 : $a + b = {}^{d}/_{2}$. Par ailleurs, les relations trigonométriques appliquées aux deux représentations de l'angle α donnent :

$$\begin{cases} \tan(\alpha) = \dfrac{p}{2a} \\ \sin(\alpha) = \dfrac{e}{2b} \end{cases}$$

En injectant dans la première relation, il vient :

$$\frac{p.\cos(\alpha)}{\sin(\alpha)} + \frac{e}{\sin(\alpha)} = d$$

Ensuite, en multipliant les deux membres par sin(α) puis en les élevant au carré, on obtient :

$$p^2 cos^2(\alpha) + 2pe.\cos(\alpha) + e^2 = d^2 sin^2(\alpha) = d^2\left(1 - cos^2(\alpha)\right)$$

Posant x = cos(α), le problème se ramène alors à la résolution de l'équation polynomiale suivante :

$$(d^2 + p^2)x^2 + (2pe)x + (e^2 - d^2) = 0$$

Sachant que e est plus petit que p et que d, et sachant que x est positif car α est inférieur à 90°, le calcul du discriminant puis de x = cos(α) s'opère de la façon suivante :

$$\Delta = 4p^2e^2 - 4(d^2 + p^2)(e^2 - d^2) = 4d^2(d^2 + p^2 - e^2) > 0$$

$$\cos(\alpha) = \frac{-pe + d\sqrt{d^2 + p^2 - e^2}}{(d^2 + p^2)}$$

$$\cos(\alpha) = 0{,}74$$

Où l'application numérique ci-dessus a été réalisée avec e = 2 cm, p = 2,47 cm et d = 5,7 cm. Puisque α ci-dessus représente un angle minimal (une planche de bois n'est pas saisissable pour un α inférieur) et puisque le cosinus est décroissant entre 0 et 90°, la valeur de 0,74 représente une valeur maximale de cos(α) pour une saisie possible de la planche de bois. Reste alors à mesurer l'inclinaison d'une planche sur une image capturée par la caméra.

II.3.b – Détermination de l'inclinaison d'une planche sur une image

La mesure de l'inclinaison d'une planche sur une image recueillie par la caméra peut se faire simplement à partir des coordonnées des sommets de la planche (acquises en II.2.b) et d'un calcul de produit scalaire. La figure 5 ci-dessous illustre la situation (Le point O représente l'origine du repère du bras robotique).

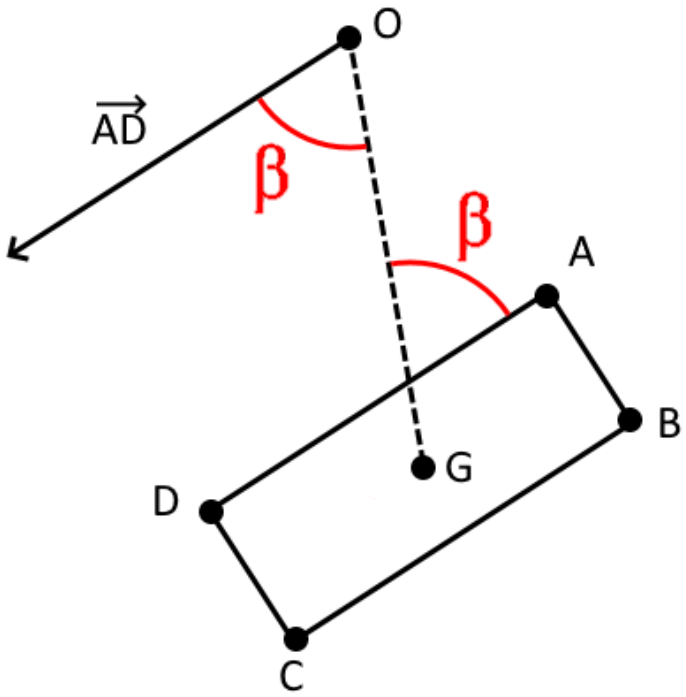

Figure 5 : schéma explicatif pour les calculs d'inclinaison d'une planche

Par un simple calcul de produit scalaire entre $\overrightarrow{AD}$ et $\overrightarrow{OG}$, comme illustré sur la figure 5, il est possible d'accéder à la valeur du cosinus de l'angle β. Puisque le système de coordonnées importe peu pour le calcul de ce produit scalaire, on obtient (en employant les notations utilisées en II.1.d) le résultat suivant :

$$\cos(\beta) = \frac{\overrightarrow{OG}.\overrightarrow{AD}}{OG \times AD}$$

$$\cos(\beta) = \frac{\big(x_p(G) - x_p(O)\big)\big(x_p(D) - x_p(A)\big) + \big(y_p(G) - y_p(O)\big)\big(y_p(D) - y_p(A)\big)}{\sqrt{\big(x_p(G) - x_p(O)\big)^2 + \big(y_p(G) - y_p(O)\big)^2}\sqrt{\big(x_p(D) - x_p(A)\big)^2 + \big(y_p(A) - y_p(A)\big)^2}}$$

Il faut toutefois rester vigilant et remarquer que l'angle β présent en figure 5 n'est pas identique à l'angle α présent en figure 4. En effet, on a $\alpha = {}^{\pi}/_{2} - \beta$. Ainsi, en utilisant la croissance de la fonction arcsinus (entre -1 et 1) et la décroissance de la fonction cosinus (entre 0 et π), la condition pour qu'une planche soit saisissable s'exprime en fonction de cos(β) de la façon suivante :

$$\cos(\alpha) < 0{,}74 \Leftrightarrow \cos\left(\frac{\pi}{2} - \beta\right) < 0{,}74 \Leftrightarrow \sin(\beta) < 0{,}74 \Leftrightarrow \beta < \arcsin(0{,}74)$$

$$\Leftrightarrow \cos(\beta) > \cos(\arcsin(0{,}74)) = 0{,}67$$

On retiendra donc la condition cos(β) > 0,67 pour qu'une planche puisse être saisie par le robot, avec cos(β) mesuré sur une image.

Afin de saisir une planche mal inclinée, le bras robotique devra la pousser de manière à la réincliner correctement. Plus de détails sur cette manœuvre seront donnés en IV.

II.4 – Calibration automatique de la vision par ordinateur

Comme mentionné en II.1.d, il est nécessaire de connaître la position sur une image capturée par la caméra de l'origine du repère du bras robotique ainsi que de deux coins de son plan de travail au minimum. L'opération de détection de ces points de façon automatique, dite calibration, est ici détaillée.

L'idée pour repérer des points sur une image est simple : il suffit par exemple d'appliquer un sticker d'une couleur très « pure » (uniquement rouge, vert ou bleu, et très intense) au niveau du point à détecter pour que la caméra puisse le reconnaître facilement à partir d'un filtre à couleurs. Cependant, l'origine du robot ne peut être détectée de cette manière à cause du bras robotique lui-même qui la cache.

L'idée est donc, plutôt que de détecter deux coins du plan de travail et l'origine du repère du robot, de détecter trois coins du plan de travail. Ainsi, en supposant l'angle de vue de la caméra suffisamment proche de la normalité au plan de travail, ce dernier peut être approximé par un parallélépipède et la connaissance de trois de ses points suffit à en déduire le quatrième. Par ailleurs, connaissant les proportions de ce plan de travail et la position de l'origine du robot, il est simple d'obtenir la position de cette dernière à partir des quatre coins (Voir Figure 6).

$$\overrightarrow{AO} = \frac{1}{2}\overrightarrow{AD} + \frac{4}{30}\overrightarrow{DC}$$

Figure 6 : Exemple de résultat de calibration automatique (haut) et équation définissant l'origine O du repère du robot (bas)

Afin d'obtenir le résultat présenté en Figure 6, des stickers rouges ont été appliqués aux coins A, B et C du plan de travail du robot. Après acquisition d'une image par la caméra, cette image est passée sous un filtre rouge permettant d'isoler les trois *stickers* qui sont ensuite détectés par leurs formes (comme illustré en II.2). Le point A est défini comme le point le sticker le plus proche du coin supérieur gauche de l'image, et le point C comme celui qui en est le plus éloigné. Ainsi le point B correspond au sticker restant, et les points D et O sont obtenus par des relations vectorielles comme détaillé plus haut. L'équation vérifiée par le point O est explicitée en Figure 6.

Ainsi, il nous est possible de calibrer automatiquement la vision par ordinateur pour effectuer sans peine la conversion des coordonnées des planches vers le repère du robot.

II.5 – Vision alternative par classificateur

En guise de conclusion du II, nous discuterons ici d'une méthode alternative pour l'identification des planches de bois.

Il existe une seconde méthode pour détecter des objets en temps réel avec OpenCV, celle des Haar Cascades. Haar Cascade est un classificateur souvent employé à la détection de visages mais son champ d'action s'étend bien au-delà. Comme lors d'une détection en temps réel de parties d'un visage, un Haar Cascade permettrait de renvoyer en temps réel les coordonnées d'une planche de bois sans se baser uniquement sur la forme des contours et sur leur aire, améliorant donc théoriquement la précision (en pratique, ce n'est pas le cas – voir plus bas).

Or, cet algorithme a besoin soit d'un modèle pré-entraîné trouvé dans la documentation scientifique (comme pour la détection de visage ou toutes les autres applications classiques du Haar Cascade), soit d'être entraîné sur des images personnalisées, et donc d'une base de données personnalisée. Une telle base de données doit contenir deux groupes d'images, les positives (où l'objet recherché est présent) et les négatives (où il est absent), ainsi que des positions de l'objet recherché sur les images positives, afin que l'algorithme ne se contente pas de renvoyer un simple booléen décrivant l'absence ou la présence de l'objet.

Il est important de noter que la méthode Haar cascade ne nécessite pas de base de données d'entraînement de grande taille. Ceci peut sembler tout indiqué pour notre application : aucune base de données de planches de bois n'est présente dans la documentation scientifique et prendre des photos de ces planches est un travail long et fastidieux.

Toutefois, nous n'avons pas concrètement exploité cette méthode dans le cadre de ce projet, et ce pour plusieurs raisons. La première est que les « bords » de l'objet détecté renvoyés pour l'algorithme sont très approximatifs (il renvoie plus une "region of interest" (ROI) qu'une position exacte), on court donc le risque d'indiquer de mauvaises coordonnées au robot, d'autant plus que quelques centimètres d'écart suffisent parfois à rater complètement la cible. La méthode du détecteur de contours de Canny est précise et ne présente pas ce défaut.

La deuxième raison est que du fait de la création "artificielle" d'images positives et de la petite taille de notre base de données, l'algorithme est susceptible de renvoyer beaucoup de faux positifs, ce qui peut être fatal lorsque l'algorithme fonctionne en temps réel. Une fois encore, l'algorithme de Canny évite ce problème car il ne fonctionne pas par apprentissage.

III – Etablissement des modèles géométriques direct et inverse

Pour piloter les mouvements du robot, on souhaite lui fournir une consigne de position à atteindre pour l'extrémité de la pince dans le repère cartésien de base du robot R_0. Il faut donc développer un algorithme permettant de convertir la position de l'organe terminal du robot en rotations pour chaque moteur présent sur chaque articulation. On suit alors les étapes détaillées dans les parties qui suivent.

III.1 – Détermination des matrices de passage avec la convention de Denavit-Hartenberg

L'idée de cette convention est de placer intelligemment des référentiels sur chaque lien d'un mécanisme sériel, de manière à faciliter significativement le calcul des matrices de passage qui s'ensuit. On place alors les référentiels de telle manière que l'axe z_i soit le long de l'axe de l'articulation et on contraint l'axe x_i d'intersecter l'axe z_{i-1} à angle droit. Grâce à cette double contrainte, nous avons besoin de seulement 4 paramètres géométriques au lieu de 6 pour décrire la pose du référentiel F_i par rapport au référentiel F_{i-1} : 2 distances et deux angles. On a représenté la position des repères liés à chaque articulation dans le cas du robot étudié sur la figure 7 ci-contre.

Le passage du repère F_{i-1} au repère F_i s'effectue en choisissant la convention où les repères vérifient les quelques règles d'usages suivantes :

- $\propto_i$ désigne l'angle de rotation entre les axes z_{i-1} & z_i autour de x_i
- a_i désigne la distance entre les axes z_{i-1} & z_i mesurée le long de x_i
- θ_i désigne l'angle de rotation entre les axes x_{i-1} & x_i autour de z_{i-1}
- d_i désigne la distance entre les axes x_{i-1} & x_i mesurée le long de z_{i-1}.

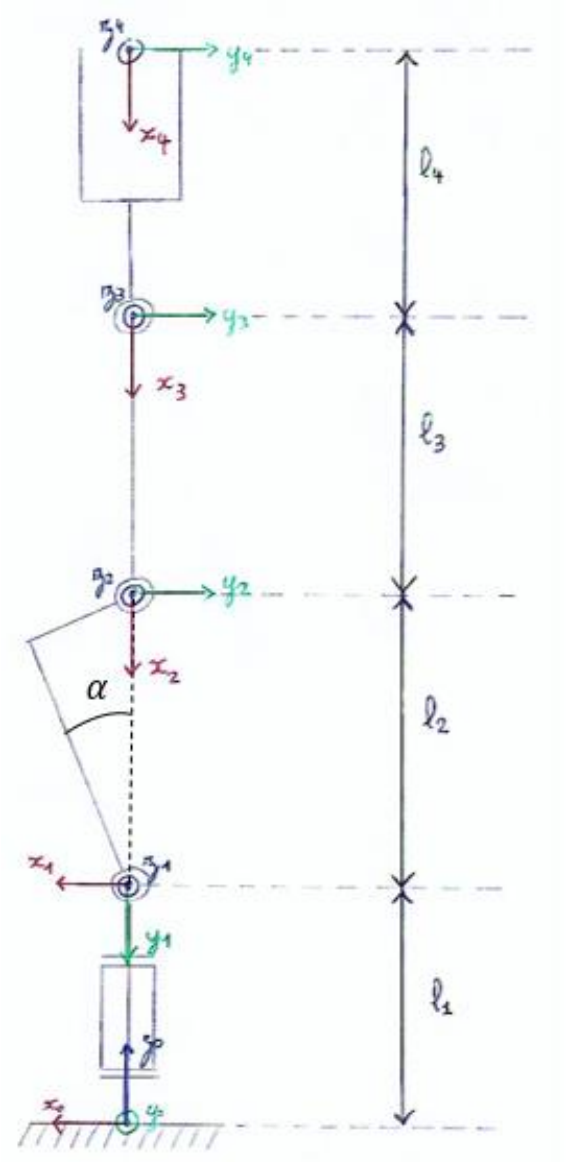

Figure 7 : choix des repères sur chaque articulation

En respectant cette convention, on a dans le cas du robot étudié les paramètres de Denavit-Hartenberg suivants (T_{ij} désignant la transition du repère i vers le repère j) :

	T_{01}	T_{12}	T_{23}	T_{34}
$\propto_i$	$-\pi/2$	0	0	0
$\boldsymbol{a_i}$	Δ_1	l_2	l_3	l_4
$\boldsymbol{\theta_i}$	θ_1	$\theta_2 - \pi/2 + \alpha$	$\theta_3 + \pi/2 - \alpha$	θ_4
$\boldsymbol{d_i}$	l_1	0	0	0

Dans le tableau ci-dessus, certains paramètres sont corrigés avec l'ajout de constantes : cela traduit un décalage entre la convention d'origine choisie sur le schéma et celle choisie par le logiciel de pilotage du robot :

- La longueur Δ_1 corrige le décalage d'origine de l'axe x_0 par rapport à la verticale de l'articulation suivante de $\Delta_1 = 0{,}0119\ m$.
- L'angle α, représenté sur la figure ci-dessus, corrige le décalage existant entre la position d'origine du bras pour le robot et celle choisie sur le schéma de la figure. Il en est de même pour les autres corrections angulaires de $\frac{\pi}{2}$ qui permettent de passer d'une convention d'origine à une autre.

On peut alors pour chaque transition $T_{i-1,i}$ écrire la matrice de passage correspondante :

$$T_{i-1,i} = \begin{pmatrix} \cos(\theta_i) & -\cos(\alpha_i)\sin(\theta_i) & \sin(\alpha_i)\sin(\theta_i) & a_i\cos(\theta_i) \\ \sin(\theta_i) & \cos(\alpha_i)\cos(\theta_i) & -\sin(\alpha_i)\cos(\theta_i) & a_i\sin(\theta_i) \\ 0 & \sin(\alpha_i) & \cos(\alpha_i) & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

III.2 – Etablissement du modèle géométrique direct

Pour obtenir le modèle géométrique direct, c'est-à-dire la position du repère 4 dans le repère 0 (Cf. Figure 7), on doit calculer la matrice TCP, avec la formule suivante :

$$TCP = T_{04} = T_{01} \times T_{12} \times T_{34} = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{pmatrix}$$

On peut alors obtenir à l'aide des 4 variables θ_i (rotation de chaque articulation) la position de l'organe terminal du robot dans le repère cartésien R_0 avec la matrice TCP. Les coordonnées (x, y, z) de l'extrémité du robot dans le repère de base sont en fait les termes (a_{14}, a_{24}, a_{34}) de cette dernière matrice.

III.3 – Etablissement du modèle géométrique inverse

III.3.a – Méthode du gradient améliorée

On souhaite maintenant réaliser l'opération inverse : déterminer un quadruplet d'angles d'articulations $(\theta_1, \theta_2, \theta_3, \theta_4)$ à partir d'une position donnée à atteindre par la pince (x, y, z). On va alors

appliquer la méthode du gradient pour déterminer le quadruplet adéquat par itérations. On définit tout d'abord arbitrairement un quadruplet initial d'angles d'articulations $(\theta_{1_0}, \theta_{2_0}, \theta_{3_0}, \theta_{4_0})$, nous reviendrons sur le choix de ce quadruplet par la suite. La solution trouvée à l'étape k de l'algorithme se détermine alors grâce à la formule de récurrence suivante :

$$q_k = \begin{pmatrix} \theta_{1_k} \\ \theta_{2_k} \\ \theta_{3_k} \\ \theta_{4_k} \end{pmatrix} = q_{k-1} + \alpha_k \times J^+(q_{k-1}) \times \left(\begin{pmatrix} x \\ y \\ z \end{pmatrix} - MGD(q_k) \right) + \beta_k \times NJ \times dH$$

- $J^+(q_{k-1})$ est la pseudo-inverse de la matrice Jacobienne pour le vecteur d'angles q_{k-1}, avec la matrice Jacobienne $J(q_k)$ qui s'obtient par la formule suivante :

$$J(q_k) = \begin{pmatrix} \frac{dx}{d\theta_1} & \frac{dx}{d\theta_2} & \frac{dx}{d\theta_3} & \frac{dx}{d\theta_4} \\ \frac{dy}{d\theta_1} & \frac{dy}{d\theta_2} & \frac{dy}{d\theta_3} & \frac{dy}{d\theta_4} \\ \frac{dz}{d\theta_1} & \frac{dz}{d\theta_2} & \frac{dz}{d\theta_3} & \frac{dz}{d\theta_4} \\ \frac{d\theta_x}{d\theta_1} & \frac{d\theta_x}{d\theta_2} & \frac{d\theta_x}{d\theta_3} & \frac{d\theta_x}{d\theta_4} \\ \frac{d\theta_y}{d\theta_1} & \frac{d\theta_y}{d\theta_2} & \frac{d\theta_y}{d\theta_3} & \frac{d\theta_y}{d\theta_4} \\ \frac{d\theta_z}{d\theta_1} & \frac{d\theta_z}{d\theta_2} & \frac{d\theta_z}{d\theta_3} & \frac{d\theta_z}{d\theta_4} \end{pmatrix} \Bigg| (\theta_{1_k}, \theta_{2_k}, \theta_{3_k}, \theta_{4_k})$$

  On a aisément pu calculer cette matrice en python en utilisant du calcul symbolique : l'ordinateur est capable de calculer les expressions de chacune des dérivées en format littéral.

  Comme on n'a que quatre conditions imposées par la suite (les positions cartésiennes du bout de la pince x, $y, z,$ & l'orientation de la pince θ_y), on n'a besoin que de quatre équations. On garde donc les quatre lignes de la matrice Jacobienne qui imposeront ces conditions : les trois premières et la cinquième. Notre matrice Jacobienne est donc finalement une matrice carrée de dimension 4.

- $MGD(q_k)$ est la fonction du modèle géométrique direct : elle détermine pour le vecteur d'angles q_k la position atteinte par l'organe terminal du robot dans le repère de base du robot R_0,
- $\begin{pmatrix} x \\ y \\ z \end{pmatrix}$ est la position finale à atteindre pour l'organe terminal du robot,
- α_k est un coefficient permettant de régler la vitesse de convergence de la méthode,

Comme le robot étudié possède plus de degrés de liberté (4) qu'il n'y a de contraintes imposées (3), on a rajouté un terme dans la formule ci-dessus, permettant d'imposer des intervalles de valeurs pour chacun des angles θ_i. Cela permet de toujours converger vers la bonne solution physiquement atteignable par le robot dans le cas de positions singulières (où plusieurs quadruplets $(\theta_1, \theta_2, \theta_3, \theta_4)$ sont solutions) :

- NJ est définie par la relation suivante : $NJ = I_4 - J^+(q_{k-1}) \times J(q_{k-1})$,
- β_k est un coefficient permettant de régler l'amplitude de la fonction de respect des contraintes.
- dH est défini par la relation suivante (abusive, mais qui permet d'expliciter une division terme à terme) :

$$dH = -2 \times (q_k - q_{mean})/({q_{diff}}^2)$$

- $q_{mean} = \frac{1}{2}(q_{max} + q_{min})$
- $q_{diff} = (q_{max} - q_{min})$

avec $q_{max} = \begin{pmatrix} \theta_{1_{max}} \\ \theta_{2_{max}} \\ \theta_{3_{max}} \\ \theta_{4_{max}} \end{pmatrix}$ et $q_{min} = \begin{pmatrix} \theta_{1_{min}} \\ \theta_{2_{min}} \\ \theta_{3_{min}} \\ \theta_{4_{min}} \end{pmatrix}$

On définit alors pour chacune des articulations l'intervalle $[\theta_{i_{min}}, \theta_{i_{max}}]$ dans lequel doit se situer l'angle θ_i.

- Il apparait assez intuitivement par exemple que $\theta_1 \in [{}^{-\pi}/_2, {}^{\pi}/_2]$ pour rester dans la zone de travail matérialisée par le plateau, dans le champ de vision de la caméra.
- Pour θ_2 & θ_3, chaque bras ne peut pas effectuer un tour complet sous peine de rentrer en collision avec un autre bras du robot, on a donc fixé $\theta_2, \theta_3 \in [{}^{-\pi}/_2, {}^{\pi}/_2]$.
- Enfin pour θ_4, toutes les situations de travail envisagées se déroulent avec le bras perpendiculaire par rapport au sol. On a donc spécifié que $\theta_4 \in [{}^{\pi}/_2 - \theta_2 - \theta_3 - \Delta\theta\ , {}^{\pi}/_2 - \theta_2 - \theta_3 + \Delta\theta]$ afin que la pince reste perpendiculaire par rapport au sol. On a ici choisi $\Delta\theta = 0{,}3\ rad$ pour que l'algorithme converge ; il n'est pas absolument nécessaire que la pince soit droite au degré près pour qu'elle puisse agripper des pièces.

On itère l'utilisation de la formule ci-dessus jusqu'à ce que l'écart de consistance de la méthode soit inférieur à un certain seuil (ici 1mm). Si l'algorithme converge, il renvoie le nombre d'itérations, les positions (x, y, z) de consigne et réellement atteinte par l'organe terminal ainsi que le quadruplet d'angles correspondant (Cf. figure 8). Sinon, l'algorithme s'arrête au-delà d'un certain nombre d'itérations et renvoie un message d'erreur.

```
Convergence en 48 étapes

Position cartésienne (cm)
x = 0.0 ----> 0.00085
y = 0.09 ----> 0.08942
z = 0.07 ----> 0.0701

Avec les paramètres angulaires suivants (rad)
q1 = 1.56124
q2 = -0.46392
q3 = 0.51515
q4 = 1.60381
```

Figure 8 : résultats retournés par l'algorithme

III.3.b – Validation & réglage des paramètres du modèle

Afin de vérifier la validité de notre modèle avant d'implémenter notre algorithme de géométrie inverse sur le système physique, on a cherché à représenter la position du robot pour certains points de la trajectoire grâce à Python. On a donc développé un algorithme permettant pour chaque position finale (x, y, z) (carré rouge sur la figure 9 ci-contre) de dessiner la position du robot. On peut ainsi vérifier que le robot atteint bien le bon point, et qu'il peut effectivement se retrouver avec de tels angles sur chaque articulation.

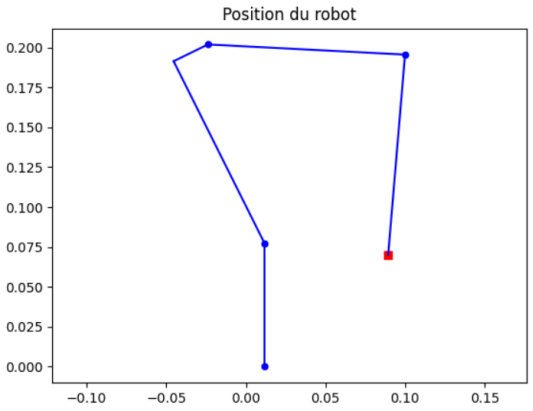

Figure 9 : Affichage de la position du robot

Afin de garantir la convergence de l'algorithme dans toutes les positions de la future trajectoire, on peut jouer sur différents leviers :

- Tout d'abord, l'enjeu pour ce type d'algorithme de descente du gradient est de trouver des conditions initiales qui permettent une convergence. Il a donc fallu trouver des conditions initiales pour la descente du gradient qui assuraient la convergence de l'algorithme pour toutes les positions demandées. Par tâtonnement, on a pu trouver des conditions permettant la convergence pour une trajectoire type de déplacement d'objet.
- Le deuxième jeu de paramètres à modifier pour assurer la convergence de l'algorithme est $(\alpha_k\,,\beta_k)$ dans l'équation :
    - Augmenter le premier paramètre assure une convergence plus rapide vers une solution *théoriquement* atteignable, c'est-à-dire trouver $(\theta_1,\theta_2,\theta_3,\theta_4)$ qui permettent effectivement que l'extrémité de la pince atteigne le point désiré,
    - Augmenter le second paramètre permet de mieux assurer que la solution finale soit *physiquement* atteignable. On donne alors plus d'importance dans l'équation au terme de correction de chaque θ_i pour qu'il se trouve bien dans son intervalle défini au préalable. Autrement dit, augmenter sa valeur assure davantage que le robot n'a pas les bras pliés dans tous les sens.

    On a pu trouver par tâtonnements un set $(\alpha_k\,,\beta_k)$ assurant une convergence rapide mais respectant également les intervalles pour chaque θ_i.

Une fois tous nos paramètres correctement réglés, on a pu valider notre modèle grâce à une simulation sur WeBots. Il a fallu pour vérifier que l'algorithme fonctionnait correctement s'adapter à de nouvelles origines, différentes de celles du robot physique. Il n'y a pas par exemple de décalage de $\Delta_1 = 0{,}0119\ m$ suivant l'axe X du repère R_0, et la position terminale affichée correspond effectivement au bout de la pince et non pas à un point légèrement plus au centre de la pince comme sur le robot physique. Une fois ces nouvelles conventions prises en compte, on a pu vérifier que notre algorithme fonctionnait correctement ! Pour l'ensemble des positions d'une trajectoire type imaginée de déplacement d'objets, l'algorithme fournissait un set $(\theta_1,\theta_2,\theta_3,\theta_4)$ permettant effectivement d'atteindre la position demandée avec la bonne orientation de pince !

III.3.c – Optimisation des mouvements

Une amélioration possible de l'algorithme est d'optimiser les mouvements effectués par le robot. Prenons comme exemple la figure 10 ci-dessous, il existe deux jeux de $(\theta_1,\theta_2,\theta_3,\theta_4)$ permettant d'atteindre le même point. On pourrait vouloir choisir la configuration minimisant la somme totale des angles nécessaires pour passer de la position précédente à cette dernière. Ou alors minimiser les mouvements de tel ou tel axe prioritairement pour des raisons de consommation d'énergie. Il faut donc mettre en place un algorithme minimisant le critère de notre choix, ici nous prendrons la minimisation de la somme des angles.

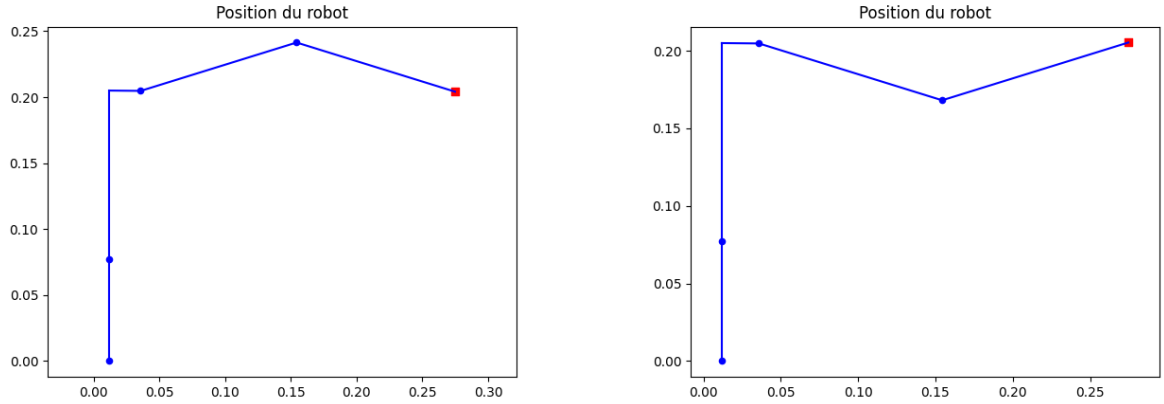

Figure 10 : une position cible du bout de la pince peut être obtenue avec 2 configurations différentes du robot

Mais alors, comment obtenir par le calcul les angles correspondant aux deux configurations du robot de la figure 10 ? Il faudrait en fait faire fonctionner notre algorithme de géométrie inverse plusieurs fois avec des conditions initiales différentes. Le calcul pourrait alors converger vers des solutions $(\theta_1, \theta_2, \theta_3, \theta_4)$ différentes. Prenons comme exemple le schéma de la figure 11 ci-dessous, traduisant de manière simplifiée notre problème. Suivant le point de départ (1 ou 2 sur le schéma), on peut converger avec la méthode des gradients vers la solution D ou la solution F. Si notre erreur est bien inférieure au seuil fixé d'erreur dans les 2 cas, les 2 solutions sont bien valables.

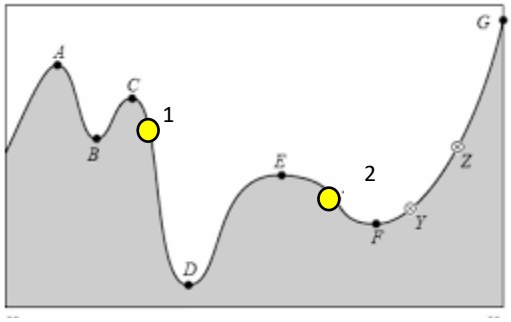

Figure 11 : Existence de minima locaux

Il faut donc pour minimiser les mouvements effectués par le robot :

- Faire fonctionner notre algorithme de descente du gradient pour plusieurs conditions initiales,
- Evaluer pour chacune des solutions le critère de minimisation,
- Renvoyer les $(\theta_1, \theta_2, \theta_3, \theta_4)$ permettant de minimiser le critère.

IV – Pilotage du bras robotique

IV.1 – Modélisation Webots

La réalisation d'un modèle Webots du bras robotique apparaît particulièrement judicieuse pour tester différentes applications sans endommager le bras et ce même lorsqu'il n'est pas disponible. Notre groupe a donc décidé de mettre en place un tel modèle, dont une représentation 3D est donnée Figure 12.

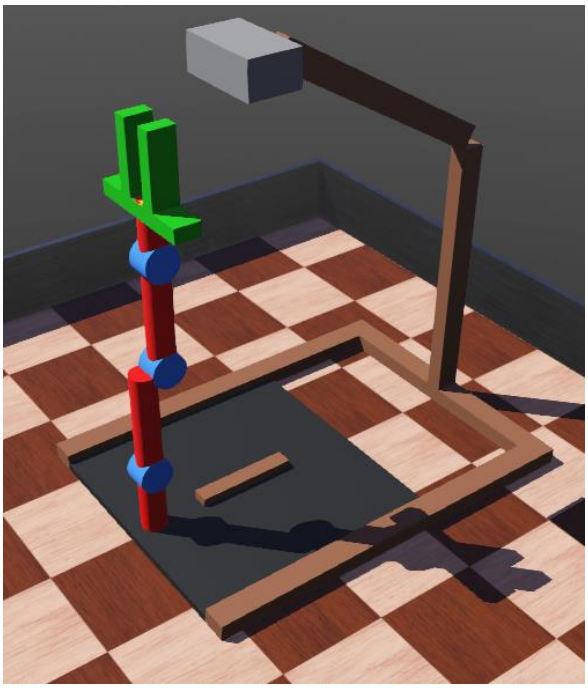

Figure 12 : capture d'écran de la représentation 3D du modèle Webots du bras robotique muni d'une caméra avec support. Le bras correspond aux cylindres rouges et bleus visibles sur la gauche

Ce modèle Webots a été conçu en prenant en compte les différents aspects physiques du système : la longueur des sections entre les liaisons correspond aux longueurs réelles, on peut les retrouver sur la documentation en ligne. Un programme de vérification permet d'empêcher le bras de la simulation de rentrer à l'intérieur du support.

Par ailleurs, le logiciel Webots permet une commande du robot en position angulaire des moteurs ou en position cartésienne à l'aide des touches du clavier. Ceci a permis de vérifier, entre autres, que le support de la caméra détaillé en II.1.a n'entravait pas les déplacements du robot, et ce même lorsque celui-ci a saisi une planche de bois. Cela a également permis de vérifier que le modèle géométrique inverse fonctionne pour toutes les positions atteignables par le robot.

IV.2 – Contrôle du bras robotique

IV.2.a – Services pour le contrôle du robot

Le constructeur du bras robotique a fourni un programme ROS2, permettant de contrôler aisément le robot par l'intermédiaire de services ROS2, que l'on peut retrouver sur le manuel en ligne du robot [4] ou sur wikiROS [5]. Parmi ces services, trois sont particulièrement intéressants :

- SetKinematics/goal_task_space_path : Il permet de contrôler la position de la pince dans un repère cartésienne avec pour origine la base du robot attaché au support. On peut également à l'aide de ce service contrôler l'inclinaison de la pince qui est représenté par un quaternion.

- SetJointPosition/goal_joint_space_path : Il permet de contrôler la position de la pince à l'aide de la position angulaire des différents moteurs. Cela nécessite donc l'utilisation d'un modèle géométrique inverse.

- SetKinematics/goal_tool_control : Il permet de contrôler l'ouverture de la pince. Il est important de savoir que la position du moteur "gripper" de 0,01 rad correspond à l'ouverture maximale, et -0,01 rad correspond à la fermeture maximale.

De plus les informations concernant l'état du robot peuvent être trouvée à tout moment dans différents topics :

- JointState/joint_states : où l'on peut trouver les positions angulaires des moteurs
- KinematicsPose/joint_states : où l'on peut trouver la position cartésienne des liaisons du robot
- OpenManipulatorState/states : où l'on peut trouver l'état de fonctionnement du bras (s'il est en mouvement, si les moteurs sont actifs ou mis en sécurité …)

Avec cela on peut commander le robot. Un exemple intéressant de l'utilisation de ces éléments est le programme teleop_keyboard.py fourni par le constructeur, qui permet de réaliser ce contrôle par l'intermédiaire des touches du clavier.

IV.2.b – Choix de la méthode de pilotage et nécessité du modèle géométrique inverse

À la lumière des apports du constructeur détaillés précédemment, il est naturel de penser tout d'abord à contrôler le bras en position cartésienne à l'aide des services fournis. Cependant la pratique a montré que cette méthode présente un décalage non-négligeable entre la position demandée et la position atteinte, et qui n'est pas le même partout sur la zone de travail. Par exemple, une position du robot sur la droite de son plan de travail est associée à une position angulaire du premier moteur de 1,58 rad tandis que la position diamétralement opposée est associée à -1,70 rad (on s'attendait à 2 valeurs opposées).

Pour pallier ce problème, on pourrait penser à corriger la position avec d'envoyer la requête au robot, mais pour cela il faudrait connaitre l'erreur en tout point du plateau et cela est difficile à mettre en œuvre. Notre groupe a donc opté pour contrôler le robot avec la position angulaire de chacun des moteurs en utilisant le modèle géométrique inverse établi en III (qui présente l'avantage de piloter la position du bout de la pince, contrairement au modèle du constructeur).

IV.2.c – Choix des trajectoires

Étant donné que l'on se place dans le cas où toute les planches sont au niveau du support, pour éviter de rentrer en contact avec des planches il suffit d'effectuer tous les mouvements dans un plan horizontal situé à cinq centimètres au-dessus du plan de travail. Ainsi pour effectuer une trajectoire, on

lève d'abord la pince verticalement, puis on déplace la pince là où l'on veut prendre ou déposer une planche, puis on redescend verticalement la pince jusqu'à l'objectif.

On décide également de travailler uniquement avec la pince orientée vers le bas, car les planches seront posées à plat sur le plan de travail et c'est donc la manière la plus naturelle de prendre les planches.

IV.2.d – Programme de contrôle

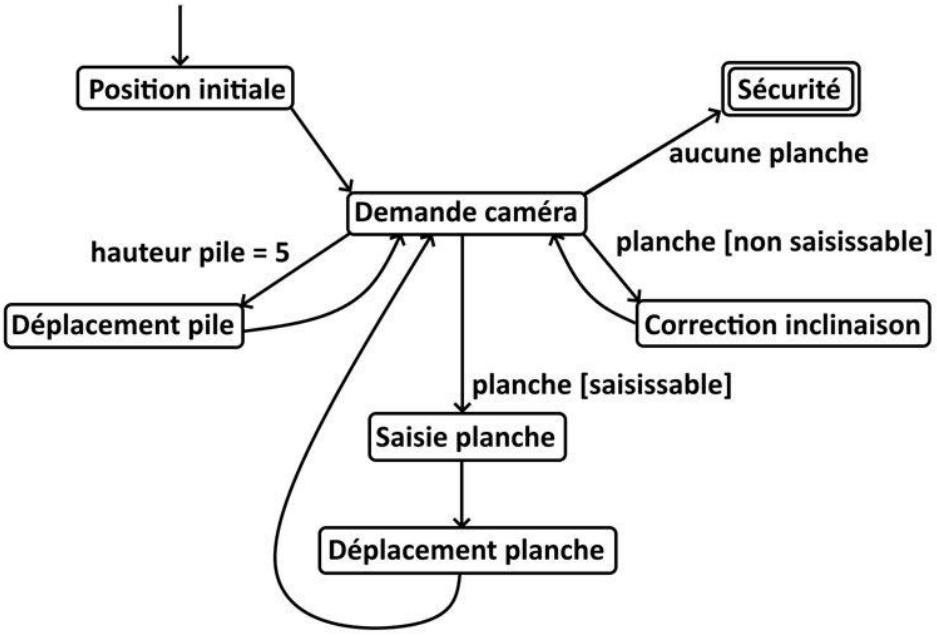

Figure 13 : automate non-temporisé schématisant le fonctionnement de l'algorithme de pilotage du robot

Le programme de contrôle permet de définir les mouvements que doit effectuer le robot en ayant les informations recueillies par la caméra, pour réussir à former des piles avec les planches qui encombrent le plan de travail.

Nous connaissons normalement le centre des planches et pour les attraper nous envoyons le robot au niveau de ce centre la pince ouverte avant de refermer la pince. Ainsi un petit écart entre la position estimée par la caméra et la position réelle de la planche n'empêche pas de la saisir.

L'idée du programme est simple, tant que la caméra voit des planches qui ne sont pas triées, le programme effectue l'une des trois actions suivantes :

- Si le programme voit au moins une planche saisissable sur le plan de travail, le programme envoie le robot la planche la plus proche de la position actuelle du robot saisir et la mettre en pile. Cette pile sera construite à côté de la base du robot.
- Si toute les planches sont insaisissables, le programme fait effectuer au robot une manœuvre pour changer l'orientation d'une planche arbitrairement choisie. Cette manœuvre consiste à balayer l'espace de travail avec la pince fermée selon un mouvement rectiligne et lent, dirigé par la droite passant par l'origine du robot et le coin de la planche le plus éloigné de cette origine. Une fois la position initiale de ce coin atteinte, le robot ouvre à nouveau sa pince, ce qui achève de corriger

l'inclinaison de la planche. Le robot se replace enfin sur le côté et redemande la position des planches sur le plan de travail.

- Si la pile que le robot construit atteint 5 planches de haut, alors il la déplace vers un autre endroit de l'autre côté du plan de travail et recommence une nouvelle pile. Cela permet de travailler toujours au même endroit pour le robot, et la manœuvre permet également d'égaliser l'alignement des planches qui peuvent ne pas avoir été prises exactement par leur centre.

Le fonctionnement de ce programme de pilotage est résumé en Figure 13. Il est important de noter que le programme garde en mémoire la position des tas de planches déjà positionnés par le robot, ainsi que celle du tas en construction. En effet, ces tas vont être détectés par l'algorithme de vision de la caméra et pourtant ils ne doivent plus être déplacés. Si une planche est détectée à 2 cm près d'un tas déjà positionné, celle-ci sera ignorée par le programme de pilotage.

Par ailleurs, le programme ne regarde les informations renvoyées par la caméra que lorsque le bras est immobile sur le côté du support et qu'il ne gêne pas la détection des planches. La gestion des actions à effectuer est alors réalisée grâce à une file dans laquelle le programme place les actions à effectuer par le robot. Ce dernier effectue alors les actions une par une jusqu'à ce que la file soit vide.

Enfin, si le programme de vision ne renvoie la position d'aucune planche qui n'a pas déjà été triée par le robot, alors le programme s'arrête après avoir mis le bras dans une position de sécurité où il touche le sol. Cette position de sécurité est de grande importance pour éviter d'endommager le matériel car le robot s'effondre lorsqu'il est déconnecté.

IV.3 – Structure ROS2

IV.3.a – Structure du programme de vision par ordinateur

Figure 14 : structure ROS2 du programme de vision par ordinateur

Le programme de vision possède une structure très simple, comme montré sur la figure 14. Il s'agit d'un nœud publisher qui envoie les informations sur la position des planches dans le topic « /camera » (conteneur d'information).

L'objectif est ici de parvenir à communiquer au topic « /camera » les informations recueillies par le programme de vision par ordinateur, c'est-à-dire, pour chaque planche détectée, un tuple décrivant la position cartésienne 2D de son centre de gravité, un booléen indiquant si elle est saisissable, et les deux tuples des positions cartésiennes 2D des deux points définissant la trajectoire de correction d'inclinaison si elle est insaisissable. Afin de transmettre ces informations, on utilise le message ROS2 PoseArray de la classe geometry_msgs, qui consiste en une liste de messages de type Pose contenant une position et une orientation en quaternion.

Chaque message de type Pose offre donc la possibilité de communiquer 7 réels différents : 3 par la position (x, y et z) et 4 par le quaternion (x, y, z et w), ce qui correspond exactement au nombre de réels qu'il faut transmettre pour chaque planche. Ainsi chaque message de type Pose transmet les informations relatives à une planche, et le message de type PoseArray contient les informations de toutes les planches détectées par la caméra.

On remarque qu'on emploie ici, par aisance, les messages de type Pose de façon détournée. Dans une perspective d'amélioration, il serait plus rigoureux de construire un type de message exactement adapté à la transmission des informations d'une planche, et ce à l'aide d'un package ament_CMake.

Il reste désormais à détailler la structure du programme de contrôle du bras robotique, qui viendra récupérer les informations contenues dans le topic « /caméra ».

IV.3.b – Structure du programme de contrôle du bras robotique

Le programme de contrôle (dont la structure ROS2 est schématisée en Figure 15) possède d'une part un nœud subscriber (« /appel_camera ») qui récupère les informations du topic « /camera », et d'autre part un nœud « /mouvement » qui définit les clients liés aux services de commande du programme open_manipulator_x_controller (les services définis en IV.2). C'est par ce nœud que sont envoyées les requêtes permettant de déplacer le bras, d'ouvrir la pince ou de fermer la pince.

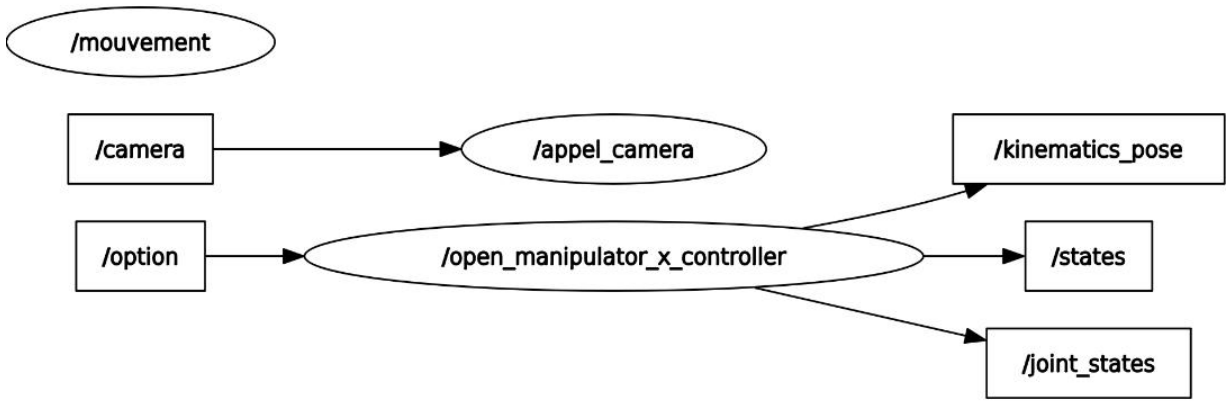

Figure 15 : structure ROS2 du programme de contrôle du bras robotique

L'ensemble des éléments de structure ROS2 présentés jusqu'ici a été combiné dans un package ROS2 ament_python afin d'être mis en pratique. Un tel package, une fois compilé, permet au bras robotique d'effectuer correctement une tâche de dévracage sous les hypothèses énoncées au fil de ce rapport.

Pour parvenir à réaliser tout ce que nous avons présenté dans ce rapport, nous avons développé un package ROS2 ament_python.

V – Résumé des perspectives d'approfondissement

Nous sommes parvenus à faire effectuer au robot un dévracage autonome de son support, dans le cas où ce dernier doit ordonner des planches identiques qui ne sont ni collées les unes ni empilées, et ce peu importe leur orientation.

Pour poursuivre ce projet, nous pourrions traiter des cas plus complexes où les planches se touchent et se chevauchent, ou bien encore où les objets à trier diffèrent par leur forme et/ou leur couleur. Nous pourrions également demander au bras robotique de construire des structures avec les planches une fois celles-ci rangées.

VI – Remerciements

L'ensemble du groupe S1.1.A du pôle projet Robotique souhaiterait remercier chaleureusement les différentes personnes qui ont permis le bon déroulé de ce projet.

Nous voudrions d'abord remercier nos encadrant Madame Makarov, Monsieur Godoy et Monsieur Da Silva pour leur aide précieuse ainsi que leur écoute bienveillante, qui nous ont permis de relever de nombreux défis posés par ce projet et d'acquérir des compétences techniques de grande valeur.

Nous remercions également nos camarades du pôle projet Robotique pour leur attention et leurs questions pertinentes lors de nos présentations, qui nous permettront certainement d'approfondir ce projet. Nous remercions enfin la Fabrique qui nous a permis de concevoir notre support de caméra et donc de réaliser nos tests ainsi que notre MVP.

Références

[1] OpenCV : Camera calibration, https://docs.opencv.org/3.4/dc/dbb/tutorial_py_calibration.html [en ligne]
[2] GitHub learnopencv, **MALLICK Satya**, https://github.com/spmallick/learnopencv [en ligne]
[3] Cours de modélisation dynamique en robotique, **GIBARU Olivier**, Arts et Métiers 2013
[4] Manuel en ligne du bras robotique https://emanual.robotis.com/docs/en/platform/openmanipulator_x/overview/
[5] Liste des topics du constructeur https://wiki.ros.org/open_manipulator_controller