

# Localisation d'une caméra embarquée par Apriltag

Culture Sciences  
de l'Ingénieur

*La* Revue  
3E.I

Gilles Arthur FADE<sup>1</sup> - Anthony JUTON<sup>2</sup>

Édité le  
08/09/2024

école normale supérieure paris-saclay

<sup>1</sup> Elève de 3ème année au département Nikola Tesla de l'École Normale Supérieure Paris-Saclay,

<sup>2</sup> Professeur agrégé de physique appliquée au département Nikola Tesla de l'École Normale Supérieure Paris-Saclay,

Cette ressource fait partie du N°111 de La Revue 3EI de janvier 2024.

Cet article présente comment, à partir d'une seule caméra embarquée, il est possible d'atteindre une précision de localisation exceptionnelle. Tout d'abord, nous considérerons la solution des Apriltags c'est-à-dire des repères physiques qui permettent, par leur position fixe connue, de trouver les coordonnées du centre du robot. Puis l'obtention de la matrice de rotation de la caméra, pour remonter à son orientation, constituera un problème mathématique : la Perspective à N Points(PnP).

## 1 - Introduction

Avec le développement croissant de l'autonomie des systèmes automatisés motorisés (véhicules automatisés), les préoccupations concernant leur sécurité sont de plus en plus présentes. La solution privilégiée : la duplication des capteurs. Si l'un des capteurs présente une défaillance, les autres capteurs devraient être capables de détecter le danger. De plus, les systèmes doivent être capable de quantifier et classer les dangers selon leur urgence. Ainsi, la fiabilisation de l'acquisition et l'estimation des distances entre les objets sont des critères essentiels.

Dans la plupart des application, une caméra pour obtenir un flux vidéo est privilégiée. On peut alors effectuer un traitement sur les images pour prendre compte de l'environnement. Cependant, une seule caméra ne peut pas nous renseigner sur sa position dans l'espace elle-même. De la même manière que l'œil humain a besoin de 2 yeux pour estimer les distances.

Une des solutions les plus stables pour déterminer la position d'une caméra dans l'espace est alors l'utilisation de balises. On connaît parfaitement la position de ces balises donc la détermination de la position s'aide de ces informations pour retrouver sa propre position.

On s'intéressa alors aux balises Apriltags : des repères fiduciels qui fonctionne en tandem avec un algorithme de détection spécialisé. Ils se présentent sous la forme de QR codes et on verra comment les mettre en œuvre dans cet article.

## 2 - Présentation des librairies

### 2.1 - Calibration : obtention des paramètres intrinsèques

A partir d'un cliché de la caméra, on connaît la position en pixels des balises, dans le repère  $(u,v)$ , on obtient alors facilement des informations sur la position des balises dans le repère lié à la caméra,  $(X_c, Y_c, Z_c)$ . On souhaite alors, à partir des positions connues des balises dans le repère réel  $(X_w, Y_w, Z_w)$ , obtenir la matrice de changement de repère entre le repère de la caméra et le repère réel. C'est-à-dire la position et l'orientation de la caméra dans le repère réel.

L'opération de calibration cherche à, pour une caméra donnée, pouvoir déduire des informations sur la position des objets réels à partir de l'image prise. Les paramètres utiles pour déterminer position réelle d'un objet à partir d'une image sont : orientation de la caméra, niveau de zoom, déformations introduites par la caméra, distance focale.

Les paramètres de la caméras sont donc séparables en 2 catégories : les paramètres intrinsèques, ceux qui dépendent de la nature de la caméra, et les paramètres extrinsèques, ceux qui dépendent de la position et de l'orientation de la caméra dans l'espace.

On veut ces paramètres pour pouvoir effectuer un passage du repère image, en pixel, au repère caméra, en mètres. Ainsi, à partir d'une image numérique, on peut en déduire la position réelle des objets.

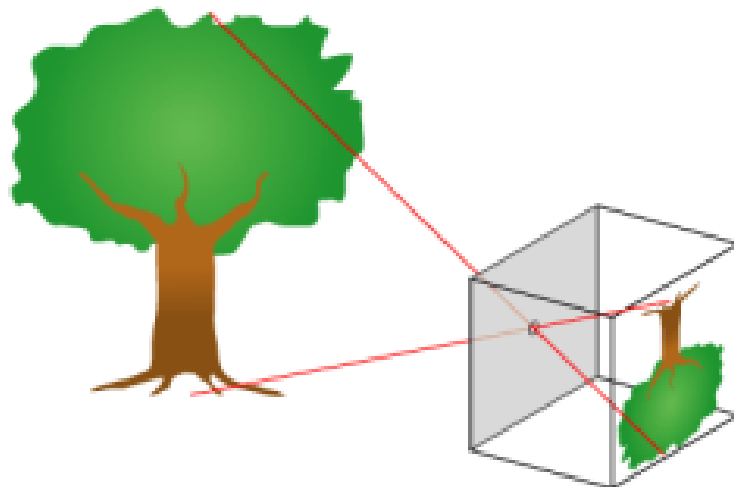


Figure 1 : Schéma sténopé[1]

Pour une image numérique, chaque pixel correspond à un rayon de lumière atteignant l'objectif provenant d'une certaine direction. On va donc considérer la caméra comme un sténopé, une sorte de chambre noire. Les seuls rayons de lumière captés sont ceux qui passent par le trou de la caméra. On a alors une relation entre la direction du rayon et sa position sur la surface détectrice.

On considérera que la surface détectrice correspond à la taille en pixel de l'image numérique. Comme l'image n'est pas inversée, on placera une surface détectrice virtuelle avant le trou. Ainsi, on peut associer à un pixel  $(u,v)$  un rayon incident de lumière. Si un objet est présent dans le pixel  $(u,v)$ , alors sa position dans le repère réel intersecte cette droite.

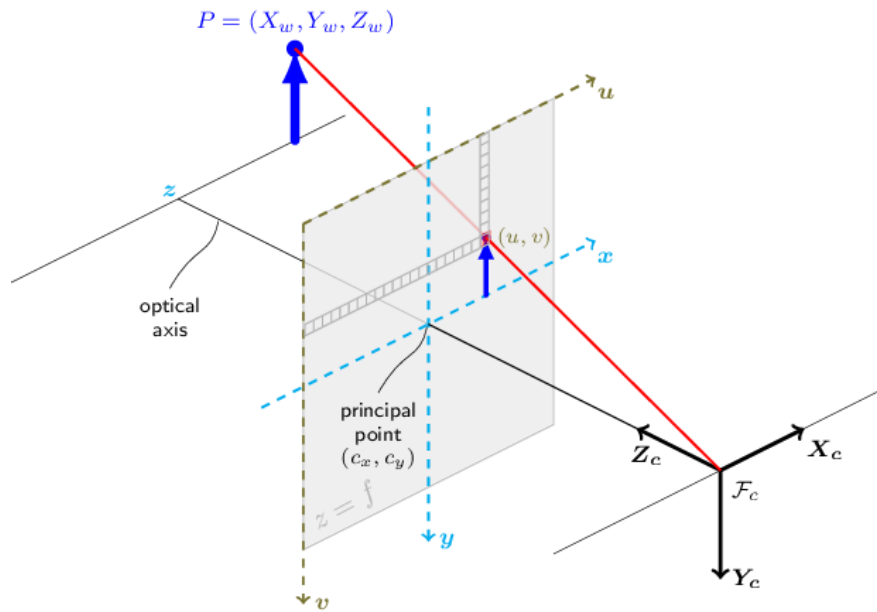


Figure 2 : Schéma repères[2]

Pour pouvoir effectuer des calculs à partir de l'image en pixels obtenue, on souhaite garantir que le comportement correspond bien à un sténopé : un pixel correspond à une direction, avec tous les rayons convergeant vers un seul point. Pour garantir cela, on va éliminer toute distorsion introduites par la caméra, c'est-à-dire estimer les paramètres intrinsèques pour pouvoir introduire une distorsion inverse.

Ce modèle nous permet de modéliser deux types de distorsions : les distorsions radiales, liées à la nature de l'objectif, et les distorsions tangentiellles, liées à la calibration de la surface détectrice virtuelle.

La distorsion radiale peut se représenter sous la forme :

$$x\_dist = x (1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \dots)$$

$$y\_dist = y (1 + k_1 r^2 + k_2 r^4 + k_3 r^6 + \dots)$$

Tandis que la distorsion tangentielle correspond à une erreur d'alignement entre le plan vertical et le plan de la surface détectrice réelle :

$$x\_dist = x + (2p_1 x y + p_2 (r^2 + 2x^2))$$

$$y\_dist = y + (p_1 (r^2 + 2y^2) + 2p_2 x y)$$

On a alors 5 paramètres à déterminer pour réaliser une calibration qui nous permettent d'arriver à une image satisfaisante.

De plus, il faut prendre en compte la conversion de pixels en unités réelles : deux caméras à la même position peuvent tout de même obtenir des clichés différents à cause de leur niveau de zoom et leur résolution. On a donc la distance focale et les dimensions de l'image qui sont des paramètres intrinsèques à prendre en compte. On les arrangera sous la forme d'une matrice de passage du repère caméra vers le repère surface détectrice.

Comme l'origine des deux repères est l'objectif de la caméra, il existe une relation linéaire entre la position du pixel considéré et celle de l'objet réel :

$$U = f_x \cdot X_{\text{réel}} / Z + c_x \quad [\text{centre image} \rightarrow \text{repère pixels}]$$

$$V = f_y \cdot Y_{\text{réel}} / Z + c_y \quad [\text{centre image} \rightarrow \text{repère pixels}]$$

On ajoute alors une équation pour fixer  $Z$ , étant donné que le cliché ne nous donne pas d'information dessus :

$Z=1$ , devient un facteur d'échelle.

$$k \begin{bmatrix} U \\ V \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 1 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}$$

OpenCV offre une méthode de calibration à partir d'un damier : on connaît la dimension réelles des cases et on sait qu'il s'agit de carrés. On peut alors, à partir de plusieurs images de calibration, retrouver les paramètres intrinsèques de la caméra.

La fonction `cv.calibrateCamera()` nous permet d'obtenir ces coefficients.

Il s'agit d'une méthode itérative : plus il y a d'images, plus on a une calibration précise. On peut alors effectuer un post-traitement sur les images pour éliminer les distorsions : on garantit que le plan  $(u,v)$  est bien un plan perpendiculaire à l'axe optique et les rayons convergent bien en l'objectif, c'est-à-dire que notre caméra numérique se comporte comme un sténopé.

Malheureusement, les coefficients des polynômes de distorsions obtenus à la fin de la calibration n'ont pas d'unité : il est difficile de juger si ils sont convenable, ce qui nécessite de tester la justesse de la calibration obtenue.

Généralement,  $f_x$  et  $f_y$  sont égaux et  $c_x$  et  $c_y$  correspondent à la moitié des dimensions de l'image.

On peut alors finalement effectuer une détection avec confiance. Il suffit désormais de trouver la relation qui lie le repère caméra,  $X_c, Y_c, Z_c$  à  $X_{\text{réel}}, Y_{\text{réel}}, Z_{\text{réel}}$ .

## 2.2 - Localisation : obtention des paramètres extrinsèques

Pour réaliser la localisation du robot, plus précisément de la caméra, dans l'espace, on utilisera la solution des Apriltags. Cela consiste en un groupe de QR codes, appelés familles d'Apriltags, qui incluent un algorithme de détection efficace. La bibliothèque logicielle permet à partir d'une image contenant un Apriltag, de récupérer la position du centre du tag et de ses bords et son inclinaison relative. De plus, grâce aux paramètres de la caméra, notamment les distorsions introduites par la caméra, on peut retrouver la distance entre le tag et la caméra.

Avec assez d'Apriltag dans l'image dont on connaît les positions et leurs distances, on peut obtenir la position réelle de la caméra donc du robot.

Lorsqu'on souhaite détecter un Apriltag dans une image, on précise tout d'abord à l'algorithme de détection quelle famille de tag nous intéresse pour le faciliter et l'optimiser.



Figure 3 : Exemple spécimen Apriltag

### Création d'une famille d'Apriltags

En effet, une famille de Apriltag est un groupe de tag respectant certaines caractéristiques. Le but est de garantir que lorsqu'on cherche des tags d'une certaine famille connue dans une image, on soit capable de les différencier très rapidement, qu'importe leur orientation ou si ils sont occultés. Ces exigences sont associées à la distance de Hamming : le nombre de bit nécessaire à changer pour passer d'un élément d'un dictionnaire binaire à un autre.

Imaginez sélectionner quels mots binaire de 8 bits associer à chaque lettre de l'alphabet pour pouvoir les différencier( Créer un dictionnaire binaire). Une solution serait de les associer un à un en partant du départ :

A 0000 0001 ; B 0000 0010 ; C 0000 0011 ... Z 0001 1010.

Mais alors, lors d'une communication, il faut vérifier chacun des bits avant de savoir quelle est la lettre reçue. Si un des bits est faux, alors on se trompera de lettre : la distance de Hamming de ce

dictionnaire est ici de 1. Si le 3eme bit de A est capté comme étant 1 au lieu de 0, on recevra un E.

Dès l'article original de 2011 sur la v1 [4], la contrainte de distance de Hamming était prise à cœur.

Maximiser la distance de Hamming signifie alors assurer que les mots binaires soient les plus différents possible en moyenne, c'est-à-dire permettre de différencier les lettres avec le moins de bits possibles nécessaire.

On peut penser à exploiter les 3 premiers bits pour différencier les lettres ayant une distance de Hamming de 1 :

A 0000 0001 ; B 0000 0010 ; C 0010 0011 ; D 0000 0100 ; E 0100 0101 ....

A,C,E ont désormais une distance de Hamming de 2, il faudrait une erreur sur 2 bits simultanément pour se tromper.

Pour la détection des Apriltags, avoir une erreur sur un des bits n'est pas réaliste : il s'agit d'objets physiques et une caméra ne devrait pas se tromper entre un pixel blanc et un pixel noir. Cependant, on peut avoir le cas d'un tag partiellement occulté : un objet empêche d'être sûr de la valeur d'un des carrés.

"In the case of visual fiducials, the coding scheme must be robust to rotation. In other words, it is critical that when a tag is rotated by 90, 180, or 270 degrees, that it still have a Hamming distance of d from every other code. The standard lexicode generation algorithm does not guarantee this property. However, the standard generation algorithm can be trivially extended to support this: when testing a new candidate codeword, we can simply ensure that all four rotations have the required minimum Hamming distance. The fact that the lexicode algorithm can be easily extended to incorporate additional constraints is an advantage of our approach." [4]

De plus, lorsqu'on détecte un tag, on ne connaît initialement pas son orientation, si à l'envers ou tourné à droite de 90°, donc, au lieu de créer des tags présentant des symétries, ils ont décidé de créer des familles de tag ayant une distance de Hamming minimale qu'importe leur rotation. Chaque tag physique appartenant à une famille correspond en réalité à 4 mots binaires, un pour chaque orientation possible, si on détecte l'un de ces 4 mots, alors on a trouvé notre tag.

La création d'une famille de tag nécessite alors de :

- Générer des mots binaires,
- Exclure les mots ayant des conséquences néfastes pour la détection : tag entièrement noir ou blanc, tag présentant des symétries...
- Assurer que le mot binaire et ses rotations ont une distance de Hamming supérieure à une certaine valeur par rapport à tous les mots déjà présents dans le dictionnaire.

Cette certaine valeur se retrouve dans le nom de la famille d'Apriltag : 36h10 signifie une famille de tag ayant 36 bits, 6x6, et respectant tous une distance de Hamming d'au moins 10 entre chacun des ses membres. On garantit que si 9 carrés sont faux ou mal captés, on pourra toujours en déduire le bon tag.

Le processus de création d'une famille peut être très intense. C'est pour cela que les familles sont déjà prégénérées dans l'algorithme de détection : il suffit de spécifier la famille visée et pour chaque mot binaire détecté, il regardera simplement dans une table de correspondance.

### Algorithme de détection

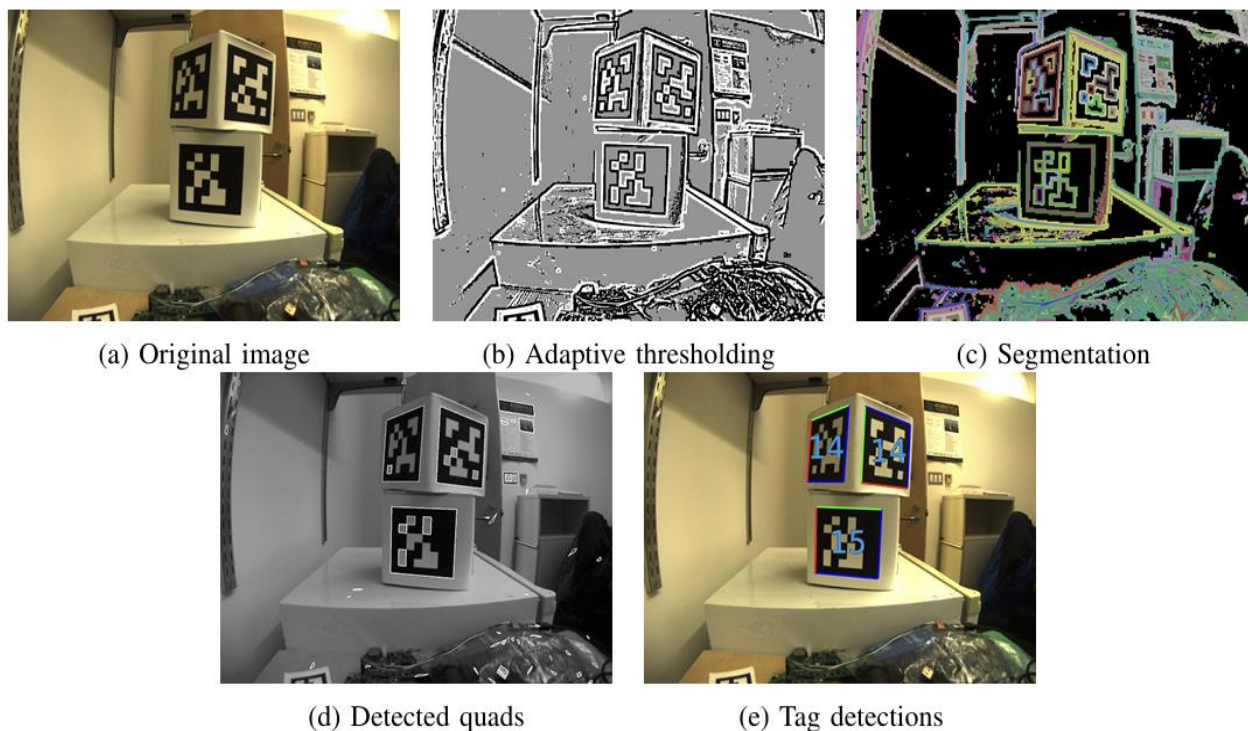


Figure 4 : Détection tags

La librairie Apriltag gère la détection et l'obtention des positions des Apriltags.

Comme les tags sont en noir et blanc, une approche naive de la détection serait de passer l'image en noir et blanc et puis d'effectuer une détection de bords pour obtenir les patterns/motifs des tags. Cependant, cette solution est dépendante de l'éclairage : si un tag est mieux éclairé qu'un autre, l'un des deux sera ignoré.

b) La solution utilisée est un seuillage adaptif : on regarde une petite zone de l'image pour déterminer la luminosité de cette zone, et on effectue le seuillage par rapport à cette luminosité moyenne. Ainsi, la détection des tags est aussi efficace à l'ombre que lorsqu'ils sont éclairés.

c) On effectue alors une détection de bords. On met en évidence les pixels à la frontière entre une zone sombre et une zone claire. Puis grâce aux zones délimitées par les bords, on peut effectuer une segmentation : on utilise les frontières pour rassembler les pixels appartenant à la même zone. On a alors des régions distinctes de pixels.

d) Avec les régions, on détermine leurs bords et on récupère les formes ayant 4 cotés. On peut alors très rapidement filtrer celles qui ne correspondent pas à des carrés et celles qui ne contiennent pas de QRcode à l'intérieur.

e) Une fois que l'on a tous les quadrilatères correspondant à des Apriltags, on peut enfin les faire correspondre avec les Apriltags de la famille recherchée en utilisant l'algorithme de détection. On associe leur index et on ajoute les lignes définissant les bords du tag. On récupère ainsi non seulement les coordonnées du centre des tags mais aussi de ses coins.

## Utilisation de la librairie

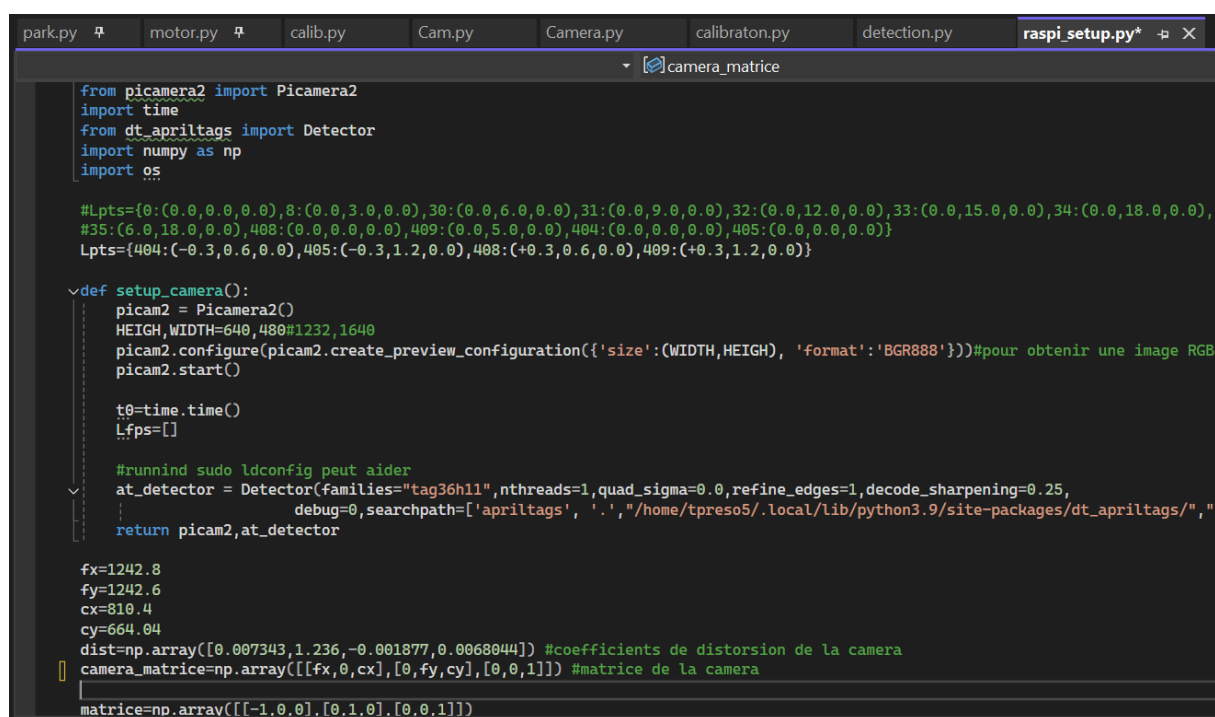
On utilise la librairie Python Apriltags, qui apporte des bindings vers la librairie C de dt\_Apriltags.

Une instance de la classe Detector est un algorithme utilisé pour traiter une image. On précise les paramètres que l'on souhaite.

Families, indique quelle famille de Apriltag on souhaite détecter.

Nthreads, combien de thread, c'est-à-dire combien d'opérations à faire en parallèle

[... le reste est des booléens]



```
park.py motor.py calib.py Cam.py Camera.py calibraton.py detection.py raspi_setup.py* X
camera_matrice

from picamera2 import Picamera2
import time
from dt_apriltags import Detector
import numpy as np
import os

#Lpts={0:(0.0,0.0,0.0),8:(0.0,3.0,0.0),30:(0.0,6.0,0.0),31:(0.0,9.0,0.0),32:(0.0,12.0,0.0),33:(0.0,15.0,0.0),34:(0.0,18.0,0.0),
#35:(6.0,18.0,0.0),408:(0.0,0.0,0.0),409:(0.0,5.0,0.0),404:(0.0,0.0,0.0),405:(0.0,0.0,0.0)}
Lpts={404:(-0.3,0.6,0.0),405:(-0.3,1.2,0.0),408:(+0.3,0.6,0.0),409:(+0.3,1.2,0.0)}

def setup_camera():
    picam2 = Picamera2()
    HEIGH,WIDTH=640,480#1232,1640
    picam2.configure(picam2.create_preview_configuration({'size':(WIDTH,HEIGH), 'format':'BGR888'}))#pour obtenir une image RGB
    picam2.start()

    t0=time.time()
    Lfps=[]

    #rinnind sudo ldconfig peut aider
    at_detector = Detector(families="tag36h11",nthreads=1,quad_sigma=0.0,refine_edges=1,decode_sharpening=0.25,
        debug=0,searchpath=['apriltags', '.', "/home/tpreso5/.local/lib/python3.9/site-packages/dt_apriltags/"],
    return picam2,at_detector

fx=1242.8
fy=1242.6
cx=810.4
cy=664.04
dist=np.array([0.007343,1.236,-0.001877,0.0068044]) #coefficients de distorsion de la camera
camera_matrice=np.array([[fx,0,cx],[0,fy,cy],[0,0,1]]) #matrice de la camera
matrice=np.array([[0,0,0],[0,0,0],[0,0,0]])
```

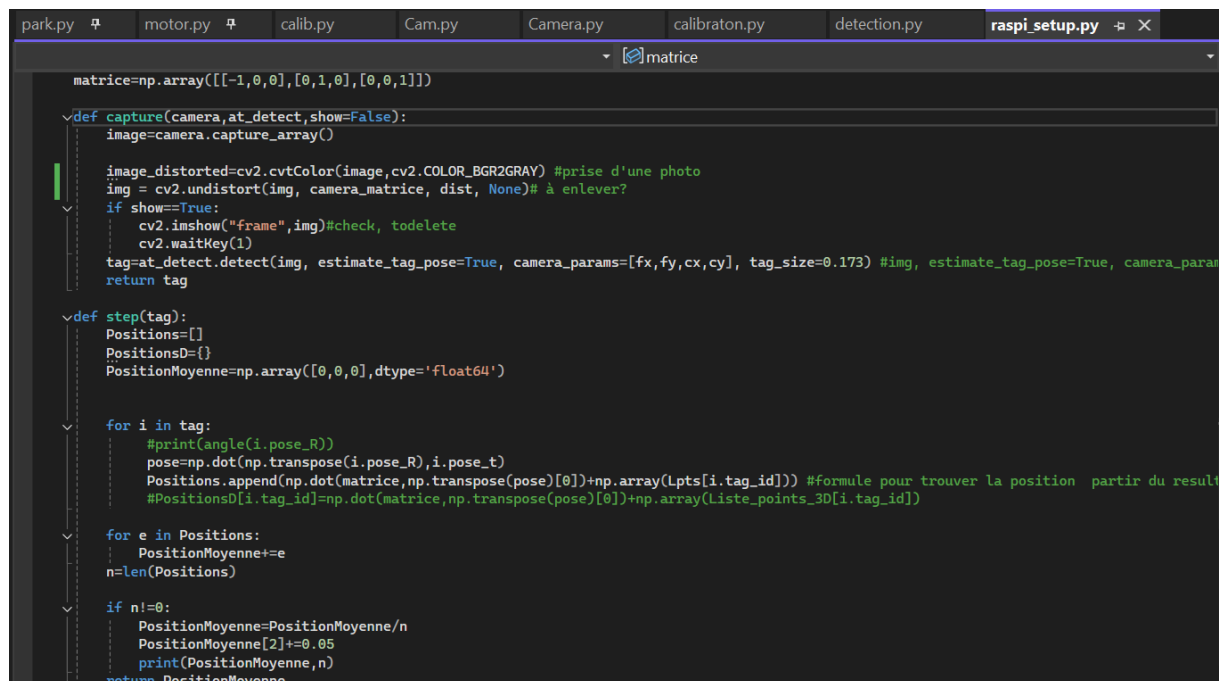
Figure 5 : Classe Detector



On peut alors utiliser sa fonction `.detect()` qui renvoie les positions détectées de chaque Apriltag dans l'image. C'est dans cette fonction que l'on précise les paramètres intrinsèques de la caméra obtenus à l'étape de calibration : `fx,fy,cx,cy` et les paramètres de calibration. On lui fournira aussi les dimensions des tags et le dictionnaire `Lpts` qui associe l'ID d'un tag à sa position réelle en mètres.

On obtient alors une structure contenant, pour chaque Apriltag, les composantes du vecteur de translation entre la caméra et l'Apriltag et la matrice de rotation de chaque tag détecté. On obtient la pose de chaque tag avec la caméra comme origine du repère.

Il suffit alors de soustraire les coordonnées connues de l'Apriltag stocké dans `Lpts` pour obtenir une estimation de la position de la caméra.



```

park.py  motor.py  calib.py  Cam.py  Camera.py  calibraton.py  detection.py  raspi_setup.py
matrice
matrice=np.array([[ -1,0,0],[0,1,0],[0,0,1]])

def capture(camera,at_detect,show=False):
    image=camera.capture_array()

    image_distorted=cv2.cvtColor(image,cv2.COLOR_BGR2GRAY) #prise d'une photo
    img = cv2.undistort(img, camera_matrice, dist, None)# à enlever?
    if show==True:
        cv2.imshow("frame",img)#check, todelete
        cv2.waitKey(1)
    tag=at_detect.detect(img, estimate_tag_pose=True, camera_params=[fx,fy,cx,cy], tag_size=0.173) #img, estimate_tag_pose=True, camera_params=[fx,fy,cx,cy], tag_size=0.173
    return tag

def step(tag):
    Positions=[]
    PositionsD={}
    PositionMoyenne=np.array([0,0,0],dtype='float64')

    for i in tag:
        #print(angle(i.pose_R))
        pose=np.dot(np.transpose(i.pose_R),i.pose_t)
        Positions.append(np.dot(matrice,np.transpose(pose)[0])+np.array(Lpts[i.tag_id])) #formule pour trouver la position partir du resultat
        #PositionsD[i.tag_id]=np.dot(matrice,np.transpose(pose)[0])+np.array(Liste_points_3D[i.tag_id])

    for e in Positions:
        PositionMoyenne+=e
    n=len(Positions)

    if n!=0:
        PositionMoyenne=PositionMoyenne/n
        PositionMoyenne[2]+=0.05
        print(PositionMoyenne,n)
    return PositionMoyenne

```

Figure 6 : fonction `capture()`

« `pose_t` » est le vecteur de translation vers le centre de la caméra dans le repère de la caméra

« `pose_R` » la matrice de rotation du tag par rapport au repère lié à la caméra

« `pose` » correspond alors au vecteur translation dans le repère réel avec l'Apriltag à l'origine.

On moyenne sur les 4 tags, en prenant en compte leurs coordonnées connues, pour obtenir la position estimée de la caméra. On obtient la pose moyenne de la caméra par rapport aux tags.

De même, on obtient la matrice de rotation de chaque tags par rapport au repère de la caméra. En les moyennant et on connaissant leur orientation, à plat sur le sol ou debout, on peut retrouver une estimation de la matrice de rotation de l'Apriltag.

## 2.3 - Décrire une orientation en 3 dimensions

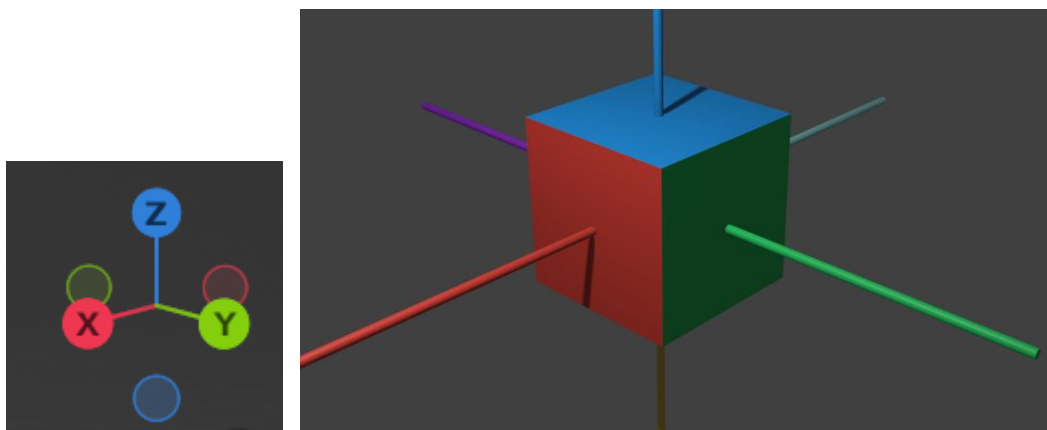
Lorsqu'une caméra se déplace dans l'espace, il est inévitable que son inclinaison soit aussi modifiée. Dans des applications de robotique, une erreur sur l'angle peut facilement se traduire par une erreur d'estimation de distance à plus grande échelle. Il est donc nécessaire de déterminer l'orientation d'une caméra lors de son fonctionnement.

Au-delà de sa détermination, il faut aussi faire le choix de sa représentation. Certaines applications sont plus adaptées à certaines représentations que d'autres.

### Par les angles

A première vue, décrire une orientation à partir de 3 angles, lié aux 3 axes  $Ox$ ,  $Oy$  et  $Oz$  semble être une solution simple. Cependant, on se rend compte de l'importance de la définition de ces angles : l'ordre dans lequel on effectue les rotations a une importance puisque les axes de l'objet varient après chaque rotation.

On peut aussi décider d'appliquer les rotations par rapport au repère immobile ou au repère de l'objet. On a 12 ordonnancements possibles des rotations et avec le choix repère fixe ou repère lié à l'objet, on a alors 24 possibilités pour décrire une rotation avec 3 angles.

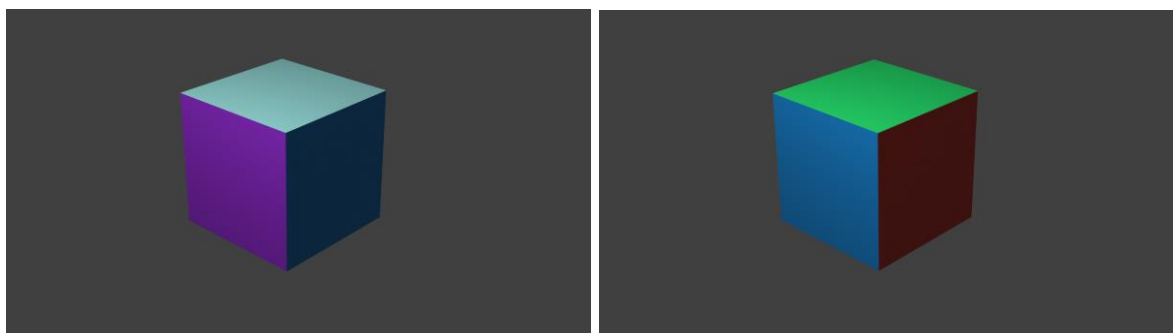


Repere fixe XYZ

Repere objet XYZ (lié aux couleurs des faces)

Figure 7.1 : Repères rotation

Avec seulement 2 angles, la problématique de l'ordre et du repère choisi est déjà présent :



(a)Rotation +90 Y puis +90 Z

(b)Rotation +90 Z puis +90 Y

Figure 7.2 : Rotation a



(c) Rotation +90 y puis +90 z

(d) Rotation +90 z puis +90 y

Figure 7.2 : Rotation b

Différentes conventions sont donc applicables, dans un repère ou l'autre :

-convention de Tait Bryan : 3 angles associés à 3 axes différents décrivent l'orientation.

(x-y-z, y-z-x, z-x-y, x-z-y, z-y-x, y-x-z)

-convention d'Euler : 3 angles associés à 2 axes différents décrivent l'orientation.

(z-x-z, x-y-x, y-z-y, z-y-z, x-z-x, y-x-y)

-Azimut Roulis et Tangage sont un cas particulier d'angles de Tait-Bryan qui correspondent aux angles nautiques : dans le repère lié à l'objet, on effectue les rotations selon les axes Oz puis Oy puis Ox. Ce sont les angles privilégiés en robotique car c'est la représentation la plus naturelle.

### Par la matrice de rotation

La matrice de rotation est une matrice 3x3 qui représente le passage du repère de départ au repère après rotation. Ses colonnes correspondent aux nouvelles positions des axes après rotation.

Il est facile, pour un ordinateur, de passer des angles Euler ou Tait Bryan à la matrice de rotation par multiplication de matrices. C'est une solution sans ambiguïté, mais qui introduit 9 variables.

Ces variables doivent en plus garantir que les 3 colonnes décrivent des vecteurs orthogonaux et de norme 1 : on a des contraintes sur les valeurs de la matrice. Une erreur numérique sur une des variables peut facilement se répercuter sur toute la matrice. L'opération d'inversion de matrice pour déterminer la rotation opposée peut facilement introduire des erreurs numériques.

## Par les quaternions

Les quaternions sont une solution qui permettent d'ignorer certains problèmes numériques qui apparaissent si les axes de rotations sont trop proche. Ils décrivent une rotation par des nombres complexes de dimension supérieure. De la même manière qu'un nombre complexe de module 1 décrit une rotation en 2 dimensions, un quaternion de module 1 décrit une rotation en 3 dimensions.

$$Z = a + b i + c j + d k$$

Ils partent de la notion que toute composition de rotations peut être décrite comme une unique rotation d'un certain angle selon un autre axe. Sur les exemples de la page précédente :

- (a) Selon l'axe diagonal  $(-1 ; 1 ; 1)$   $(-X ; +Y ; +Z)$
- (b) Selon l'axe diagonal  $(1 ; 1 ; 1)$   $(+X ; +Y ; +Z)$

On stocke alors uniquement le vecteur correspondant à l'axe de rotation composé et l'angle utilisé. On pourrait simplement stocker un vecteur avec des valeurs réelles et un angle mais la représentation sous forme de quaternions permet de trivialisier l'opération de composition. Là où une matrice de rotation nécessite de multiplier des matrices 3x3, les quaternions le permettent avec une unique multiplication qui suit les règles suivantes :

$$i*i = -1, j*j = -1, k*k = -1$$

$$ij = -ji = k$$

$$jk = -kj = i$$

$$ki = -ik = j$$

La composante réelle nous donne le cosinus de l'angle de rotation effectué, les composantes imaginaires décrivent l'axe de rotation, normalisées par le sinus de l'angle de rotation. Les quaternions respectent alors/ont une norme de 1 pour être une rotation pure.

Ils garantissent que chaque rotation a une unique représentation possible, écrite sous la forme de quaternions. L'interpolation entre deux orientations est donc triviale et est garantie d'être le chemin le plus court sur une sphère, l'orthodromie.

Ils sont tout de même moins lisibles pour un être humain mais plus efficace pour les ordinateurs : 4 nombres suffisent pour décrire la rotation. Et comparé à la matrice de rotation pour trouver la rotation inverse, il suffit de multiplier par -1. C'est pour cela que les quaternions sont la solution privilégiée pour les logiciels modélisations 3D/représentation de la 3D(dans l'industrie).

## Comparaison

Pour comparer les différentes méthodes, on résoudra un problème simple de rotation.

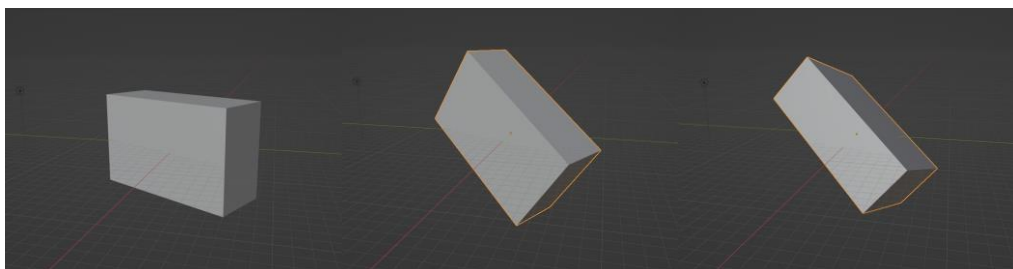
On choisit un point P de coordonnées cartésiennes (8,3,5) et on souhaite lui faire effectuer un rotation par rapport au centre O du repère et calculer les coordonnées de P', le nouveau point.

La rotation effectuée sera une rotation quelconque, décrite par, en angle de Tait-Bryan :

-une rotation de  $63^\circ$  selon +z(repère local à objet)

-une rotation de  $37^\circ$  selon +y

-une rotation de  $+70^\circ$  selon x



Rotation effectuée sur un pavé de 8x3x5 de dimensions

*Figure 8 : Rotation de comparaison*

On s'attend, « avec les mains », à un point avec X et Y similaires et Z légèrement négatif.

Comparons les méthodes matrice de rotation et quaternions.

Pour obtenir la matrice de rotation correspondant à ce mouvement, on va multiplier les matrices de rotations correspondant à chacun des 3 rotations :

$$M_Z = \begin{bmatrix} \cos(63) & -\sin(63) & 0 \\ \sin(63) & \cos(63) & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad M_Y = \begin{bmatrix} \cos(37) & 0 & \sin(37) \\ 0 & 1 & 0 \\ -\sin(37) & 0 & \cos(37) \end{bmatrix} \quad M_X = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(70) & -\sin(70) \\ 0 & \sin(70) & \cos(70) \end{bmatrix}$$

La matrice de rotation totale est donc :

$$M = M_Z * M_Y * M_X \quad (\text{ordre important})$$

$$M = \begin{bmatrix} 0.3625 & -0.0480 & 0.9307 \\ 0.7115 & 0.6591 & -0.2432 \\ -0.6018 & 0.7504 & 0.2731 \end{bmatrix}$$

Pour un total de 2 multiplications de matrices soit  $27+27=54$  multiplications.

Pour les quaternions,

$$Q_z = \cos(63/2) + 0i + 0j + \sin(63/2)k$$

$$Q_y = \cos(37/2) + 0i + \sin(37/2)j + 0k$$

$$Q_x = \cos(70/2) + \sin(70/2)i + 0j + 0k$$

Le quaternion représentant la rotation totale est :

$$Q = Q_z * Q_y * Q_x$$

En respectant les règles de multiplication des nombres imaginaires  $ijk$  :

$$Q = [(\cos(63/2) + \sin(63/2)k) * (\cos(37/2) + \sin(37/2)j)] * Q_x$$

$$Q = [0.8085 - 0.1657i + 0.2705j + 0.4955k] * (\cos(70/2) + \sin(70/2)i)$$

$$Q = \cos(70/2)[0.8085 - 0.1657i + 0.2705j + 0.4955k] + [0.8085i + 0.1657 - 0.2705k + 0.4955j] \sin(70/2)$$

(car  $ii = -1$ ,  $ji = -k$ ,  $ki = j$ )

$$Q = 0.7573 + 0.3280i + 0.5057j + 0.2507k$$

Pour un total de 2 multiplications de quaternions soit  $16 + 16 = 32$  multiplications.

Maintenant que l'on a les éléments représentant la rotation totale, on peut alors les appliquer au point P pour trouver P'.

Pour les matrices de rotations, un point est un vecteur colonne composé de ses coordonnées.

$$P' = M * \begin{bmatrix} 8 \\ 7,4101722 \\ 5 \end{bmatrix} = \begin{bmatrix} 6,4541212 \\ -1,1973573 \end{bmatrix}$$

9 multiplications.

Et pour les quaternions, P est un quaternion avec une composante réelle nulle. En retournant à l'analogie à un axe associé à un angle de rotation, P est assimilé à une rotation de  $0^\circ$  autour d'un axe passant par le point P.

$$P' = Q * (0 + 8i + 3j + 5k) * Q' \quad (Q' \text{ conjugué de } Q, Q' = +a - bi - cj - dk)$$

$$P' = [0.7573 + 0.3280i + 0.5057j + 0.2507k] * (0 + 8i + 3j + 5k) * [0.7573 - 0.3280i - 0.5057j - 0.2507k]$$

$$P' = [0.7573 + 0.3280i + 0.5057j + 0.2507k] * [5.3946 + 7.8348i + 2.6375j + 0.7249k]$$

$$P' = 0 + 7.4080i + 6.4518j - 1.1957k$$

12 + 16 multiplications si on exploite pas le fait que Q' est le conjugué de Q et que le résultat a une partie réelle nulle. Seulement 15 multiplications si prend en compte ces informations.

On obtient bien dans les 2 cas une valeur autour de :

$$P' = (7,41, 6,45, -1,19)$$

Les quaternions sont plus efficace lorsqu'il s'agit de manipuler des rotations entre elles et de les stocker en mémoire, mais beaucoup moins lorsqu'il s'agit de revenir à une représentation intuitive.

On préférera donc utiliser une matrice de rotation dans la suite de notre étude : on ne s'inquiète pas de l'efficacité algorithmique de notre solution. On accepte avoir un modèle légèrement moins efficace si il nous permet de remarquer à vue d'œil si le résultat est absurde.

La conversion de matrice de rotation vers les angles nautiques est simplement, pour une matrice de rotation de la forme :

$$\text{Rot} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}$$

Les relations suivantes :

$$\text{azimut} = \tan^{-1}(d/a)$$

$$\text{tangage} = \tan^{-1}(-g/\sqrt{h^2 + i^2})$$

$$\text{roulis} = \tan^{-1}(h/i)$$

## 3 - Mise en œuvre

### 3.1 - Mise en place algorithmique

Pour mettre en œuvre ces différents algorithmes, on peut tous les implémenter sous Python. En effet, la librairie Python openCV implémente le calibrage et propose plusieurs algorithmes de pour obtenir les paramètres de distorsion dont « findChessboardCorners ». Il suffit alors d'installer la librairie openCV et la librairie dt\_Apriltag.

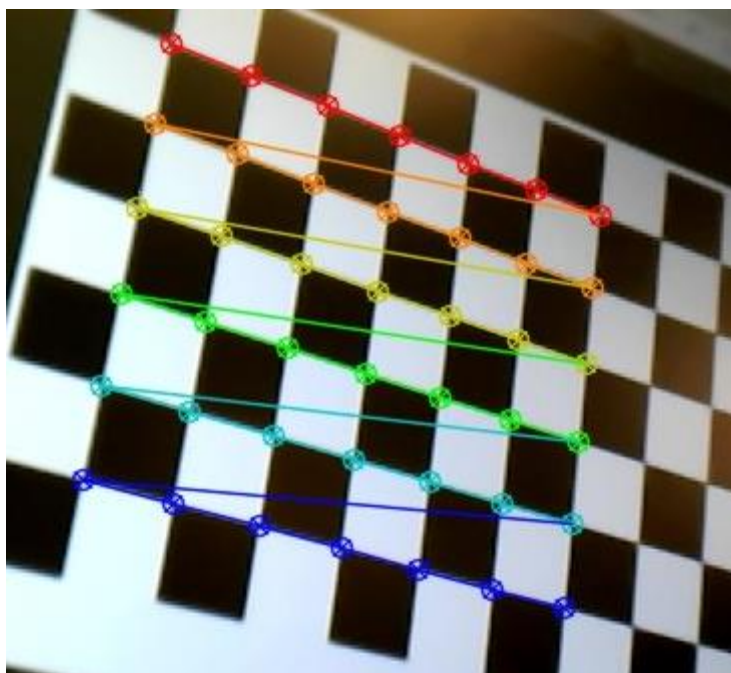


Figure 9 : Détection de damier pour la calibration

Comme il s'agit uniquement de codes tournant sous Python, il est donc possible de réaliser toutes les opérations sur une Raspberry Pi. On peut donc réaliser un système embarqué indépendant tant qu'on inclut une alimentation. On utilisera donc une caméra connectée à une Raspberry pi pour effectuer le traitement.

Le guide d'installation d'un OS sur Raspberry Pi à des fins de système embarqué est disponible sous [CoVAPSy : Premiers programmes Python sur la voiture réelle - CultureSciences de l'Ingénieur - éducol STI \(education.fr\)](#) [8]

Grâce aux capacités de la Raspberry pi, on peut simplement établir un lien entre un pc et les données obtenues.

On utilisera donc un routeur pour permettre la connexion en SSH entre un PC CLIENT et une Raspberry pi SERVEUR. La librairie Python « socket » est particulièrement adaptée pour cette tâche.

Comme on doit à la fois réceptionner les commandes et détecter en continu si les 4 Apriltags sont en vue, on mettra en œuvre du multi threading pour réaliser ces 2 tâches simultanément. Un thread



Network qui, dès que le message de détection est reçu, modifie une variable globale. Et un thread Detection, qui, lorsque la variable est modifiée, envoie alors les résultats au PC.

On peut ainsi organiser le projet en plusieurs parties :

-code Raspberry->Camera

-code Raspberry->PC

-code PC->Raspberry

### 3.2 - Validation vision

On implémentera donc cette solution logicielle sur un véhicule d'informatique embarquée. On va chercher à faire se garer un robot précisément dans sa place attitrée de parking.

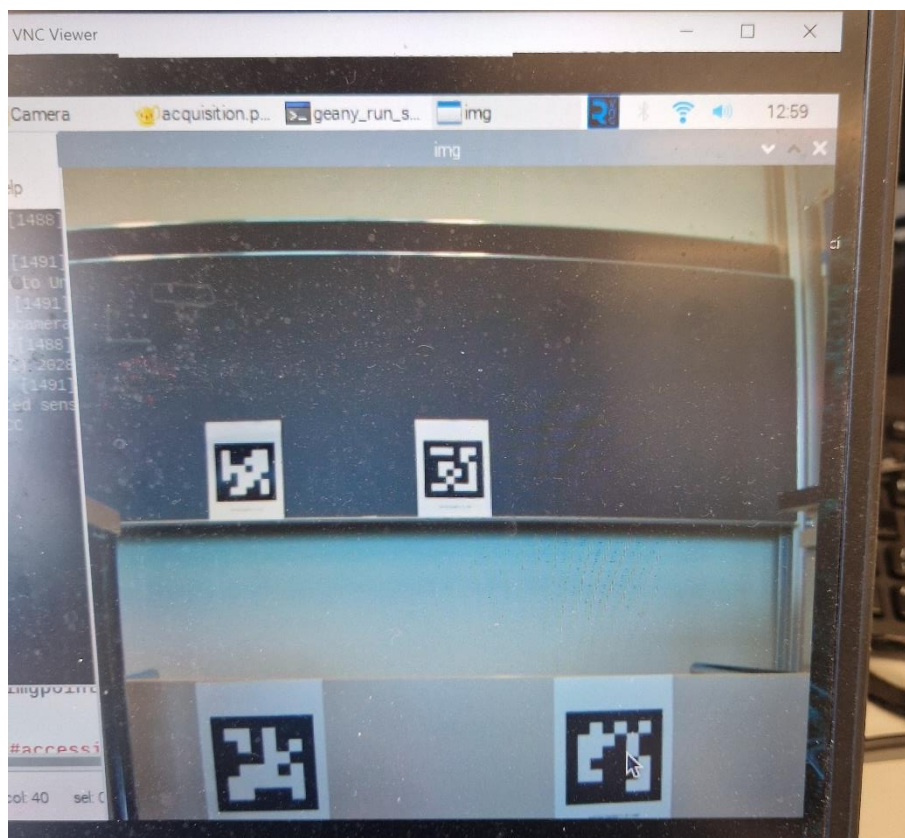


Figure 10 : Vision de la caméra

On relie une caméra à notre Raspberry pi pour effectuer la détection des Apriltags. Comme on peut spécifier les coordonnées des Apriltags, on les placera verticalement pour qu'ils soient visibles de loin. Il aurait été même possible de les placer au plafond.

Comme la matrice de rotation renvoyée est celle des Apriltags par rapport à la caméra, et qu'on les place tous avec la même orientation, on obtient une moyenne de la matrice de rotation du mur par rapport à la caméra :

L'angle qui nous intéresse pour le guidage est alors l'azimut [Y,Z] que l'on obtient par :

$$\text{Azimut} = \tan^{-1}(e/b).$$

Et on appliquera alors un angle opposé à cet azimut pour s'aligner avec le mur.

### 3.3 - Validation expérimentale

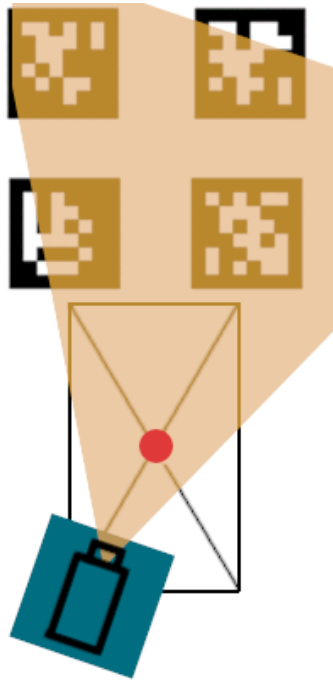


Figure 11 : Schéma montage expérimentale robot et place cible

On utilisera donc un robot composé de deux moteurs de directions et de deux moteurs de propulsion à l'avant, et d'une roue libre à l'arrière.

L'expérience sera alors constituée des 4 balises qui délimitent la place et d'un repère au centre de la place. On tentera alors de garer la voiture en alignant la caméra avec le centre de la place.

On place alors les balises en aval de la place et on considérera la place comme en amont du centre des balises pour que la caméra voit les 4 balises durant la manœuvre de parking.

Les moteurs de directions utiliseront l'azimut trouvé précédemment et les moteurs de propulsion avanceront d'un angle correspondant à la distance détectée.

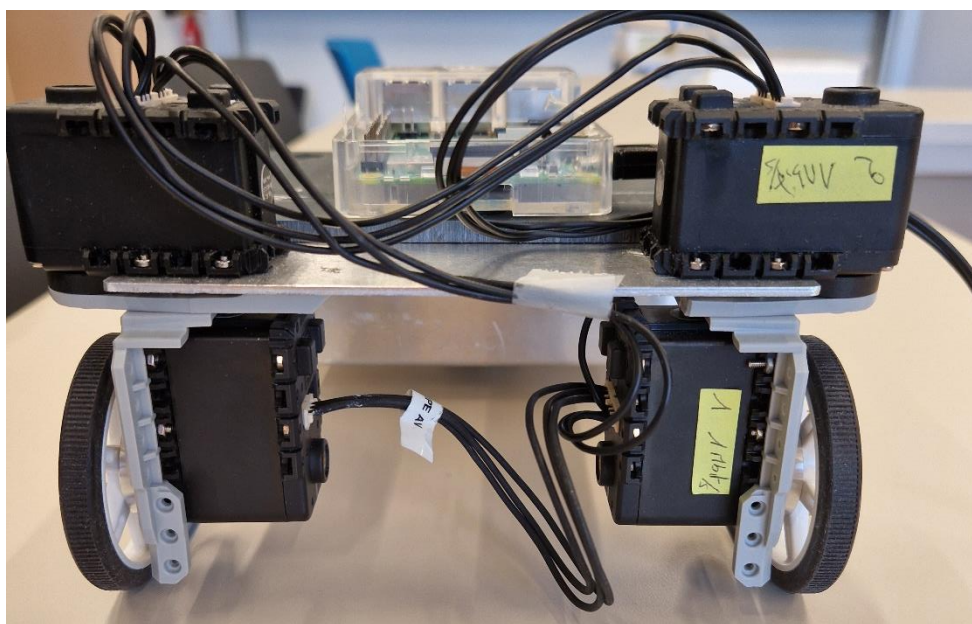


Figure 12 : Robot maquette

### 3.4 - Résultats

On mesurera alors à la fois la précision de détection, si les valeurs de la caméra sont justes et la précision de suivi, si le robot arrive correctement au centre de la place.

Mesures de position		
Distance réelle(cm)	Détection(cm)	Erreur suivi(cm)
100 ; 0	102,1 ; 1,7	97 ; 1,0
100 ; 30	101,7 ; 34	96 ; 2
100 ; -30	102,6 ; -28	95 ; -2,3
50 ; 0	48,3 ; 1,5	46 ; 0,8
50 ; 30	47,7 ; 33	46 ; 1,7
50 ; -30	50,2 ; -26	47 ; -2,0

*Tableau 1 : Résultats des expériences*

Lorsqu'on introduit un décalage le robot, et qu'il est alors nécessaire de calculer son azimuth pour se positionner correctement, l'erreur de suivi augmente grandement. On pourrait l'attribuer aux moteurs de direction qui ne suivent pas parfaitement l'angle souhaité.

De plus, le robot ne détecte rien en dehors des balises : il ne sait pas si il y un obstacle sur la route ou si il vaut mieux reculer et recommencer la manœuvre. Si on détecte les 4 Apriltags, on avance tout droit vers la place, sinon, on reste en place. Une solution serait de reculer jusqu'à ce qu'on ait les 4 Apriltags en vue, et cela marcherait si les 4 tags sont en face du robot, Mais dans des situations plus complexes(robot de biais), reculer n'aiderait pas.

## 4 - Conclusion

Cette ressource s'est donc intéressée à la problématique de localisation dans l'espace à partir d'une unique caméra. Pour les applications d'informatique embarquée souhaitant un repérage de qualité avec des balises visuelles, on a préféré la solution des Apriltags par leur facilité de mise en œuvre : il suffit d'imprimer les tags avec les dimensions de notre choix et de fournir à la classe Detector d'Apriltag les paramètres intrinsèques de la caméra pour obtenir le vecteur de rotation et la matrice de rotation entre le tag et la caméra.

La précision s'améliore avec le nombre de tags visibles mais on peut tout de même effectuer une localisation avec un seul tag : son asymétrie à la rotation nous permet de connaître notre orientation sans ambiguïté et sa taille connue nous renseigne sur la distance entre caméra et tag.

On peut alors aisément déployer cette solution en tant qu'outil pédagogique : que ce soit pour réaliser une localisation sommaire, ou en tant que capteur redondant pour la sécurisation.

## Références :

- [1]: [Sténopé — Wikipédia \(wikipedia.org\)](https://fr.wikipedia.org/wiki/Sténopé)
- [2] : [OpenCV: Camera Calibration and 3D Reconstruction](#)
- [3] : [wang2016iros.pdf \(umich.edu\)](#)
- [4] : [olson2011tags.pdf \(umich.edu\)](#)
- [5] : [krogius2019iros.pdf \(umich.edu\)](#)
- [6] : [Conversion between quaternions and Euler angles - Wikipedia](#)
- [7] : Annexe : installation de Raspberry pi OS : [CoVAPSy : Premiers programmes Python sur la voiture réelle - CultureSciences de l'Ingénieur - éducol STI \(education.fr\)](#)
- [8] : [Remote - SSH - Visual Studio Marketplace](#)

Ressource publiée sur Culture Sciences de l'Ingénieur : <https://eduscol.education.fr/sti/si-ens-paris-saclay>