Introduction à la robotique mobile **GLO-4001/GLO-7021**

Philippe Giguère A14

Laboratoire N^{o}6

Kinect

Travail en équipe de 2

Matériel fourni : règle de 1 mètre, Kinect, clé USB.

Partie 1 – Utilisation de la Kinect

1.1 – Mise à jour de la clé USB

Avant de débuter, vous devrez effectuer la mise à jour du code sur **l'ordinateur portable du robot** et changer le fichier de configuration pour activer l'utilisation de la Kinect. Pour ce faire, vous devez télécharger le fichier `update.zip`, le décompresser avec la commande `unzip` et exécuter le script `udpdate.sh`. Vous devrez donc ouvrir une console, vous déplacer vers le dossier `update` grâce à la commande :

```
> cd /chemin/vers/votre/dossier/update
```

et changer les droits d'accès pour permettre l'exécution du script grâce à la commande :

```
> chmod 777 update.sh
```

Vous pouvez maintenant exécuter le script grâce à la commande :

```
> ./update.sh
```

Vous avez maintenant la dernière version du code compilé ainsi qu'un fichier de configuration qui permet l'utilisation de la Kinect. Assurez-vous que la Kinect est alimentée et branchée sur l'ordinateur portable. Comme d'habitude, ouvrez une nouvelle console et démarrer le robot grâce à la commande

```
> ./start
```

1.2 – Observation des données

Avant de pouvoir observer les informations produites par la Kinect, vous devez télécharger le dossier compressé `codeKinect.zip` contenant le code sur le site du cours et le placer dans un dossier contenant la dernière version compilé (avec la commande mex) du dossier ros4mat sur l'ordinateur de base. Ensuite, vous devez changer la variable *IP* dans le script *initialize.m* par la valeur du IP de l'ordinateur portable du robot. Exécutez le script *initialize.m* et *subscribeKinect.m* pour débuter la communication et obtenir les données de la Kinect.

Pour les étapes suivantes, vous pouvez utiliser le script *labo6Kinect.m* disponible dans le dossier que vous avez téléchargé. En exécutant le code vous devriez obtenir une figure contenant deux images :

l'image couleur RGB et l'image de profondeur. Effectivement, la Kinect utilise une caméra couleur (RGB), et une paire émetteur/capteur infrarouge qui permet d'obtenir la profondeur pour chaque pixel. La commande *colorbar;* (en commentaire dans le code fourni) permet d'afficher la légende de couleur représentant la profondeur des pixels pour la seconde image.

À noter :

1. La boucle d'affichage des images sera exécutée jusqu'à ce que la fenêtre de la figure soit fermée. De plus, la dernière image RGB et image de profondeur sont conservées à la fermeture de la figure. Vous pouvez donc utiliser ces données par la suite.

2. La fonction *flipdim* est appliquée sur l'image BGR envoyée par la Kinect afin de renverser les canaux de couleur (i.e. troisième indice dans la matrice) et d'obtenir le format RGB utilisé pour l'affichage dans Matlab.

3. La fréquence d'affichage des images est assez basse, car les données sont transférées par le réseau. La Kinect peut obtenir les informations à environ 33 Hz (1 image par 30 ms), ce qui est bien plus élevé que ce que vous obtenez sur la figure. De plus, il y aura une certaine latence (délai) entre la prise de données et la réception/affichage de l'information (1 à 3 secondes).

Les données ainsi obtenues de la Kinect sont utiles dans plusieurs contextes en robotique, mais il peut être intéressant de les transformer sous la forme d'un nuage de point. Pour faire la transition, il suffit de calculer la position x et y de chaque pixel connaissant la longueur focale de la caméra RGB, la position du pixel dans l'image et la position en z (profondeur). La matière vue en classe à propos des caméras et du théorème de Thalès vous permet d'effectuer ces calculs, mais nous vous avons fourni la fonction *depthToCloud* pour éviter de refaire le travail. Consultez le script pour connaitre le format retourné par la fonction. Notez que l'unité utilisée par l'image de profondeur est le millimètre et que pour le nuage de points les positions sont en mètres. Aussi, cette fonction assume une distance focale particulière, qui peut être plus ou moins proche de la valeur réelle dans votre Kinect. Pour plus de précision, il vous faudrait utiliser la focale réelle de votre Kinect, soit par calibration, soit en utilisant des fonctions (non-supportée par ros4mat) qui utilisent des paramètres de calibration interne de votre appareil.

Utilisez cette fonction *depthToCloud* ainsi que la fonction *plot3(x,y,z,'.');* pour afficher le nuage de point sans les couleurs associés aux pixels. Assurez-vous que l'échelle de vos tous vos axes est la même, autrement votre nuage semblera déformé (vous pouvez regarder le code de *labo6Kinect.m* qui conserve l'échelle lors de l'affichage des images). Par exemple, pour une image de profondeur `depthMap`, les commandes suivantes afficheront le nuage de point 3D sous forme de petits points noirs :

```
DepthData = depthToCloud(depthMap);
plot3(DepthData(:,:,1),DepthData(:,:,2),DepthData(:,:,3),'k.','MarkerSize',1);
axis equal
```

Pour déplacer le point de vue de la caméra dans le rendu matlab, cliquez d'abord sur l'icône dans la fenêtre matlab indiqué ici-bas, puis cliquez dans l'espace d'affichage où sont les points, en déplaçant la souris. Vous devriez voir le nuage de points changer d'orientation.

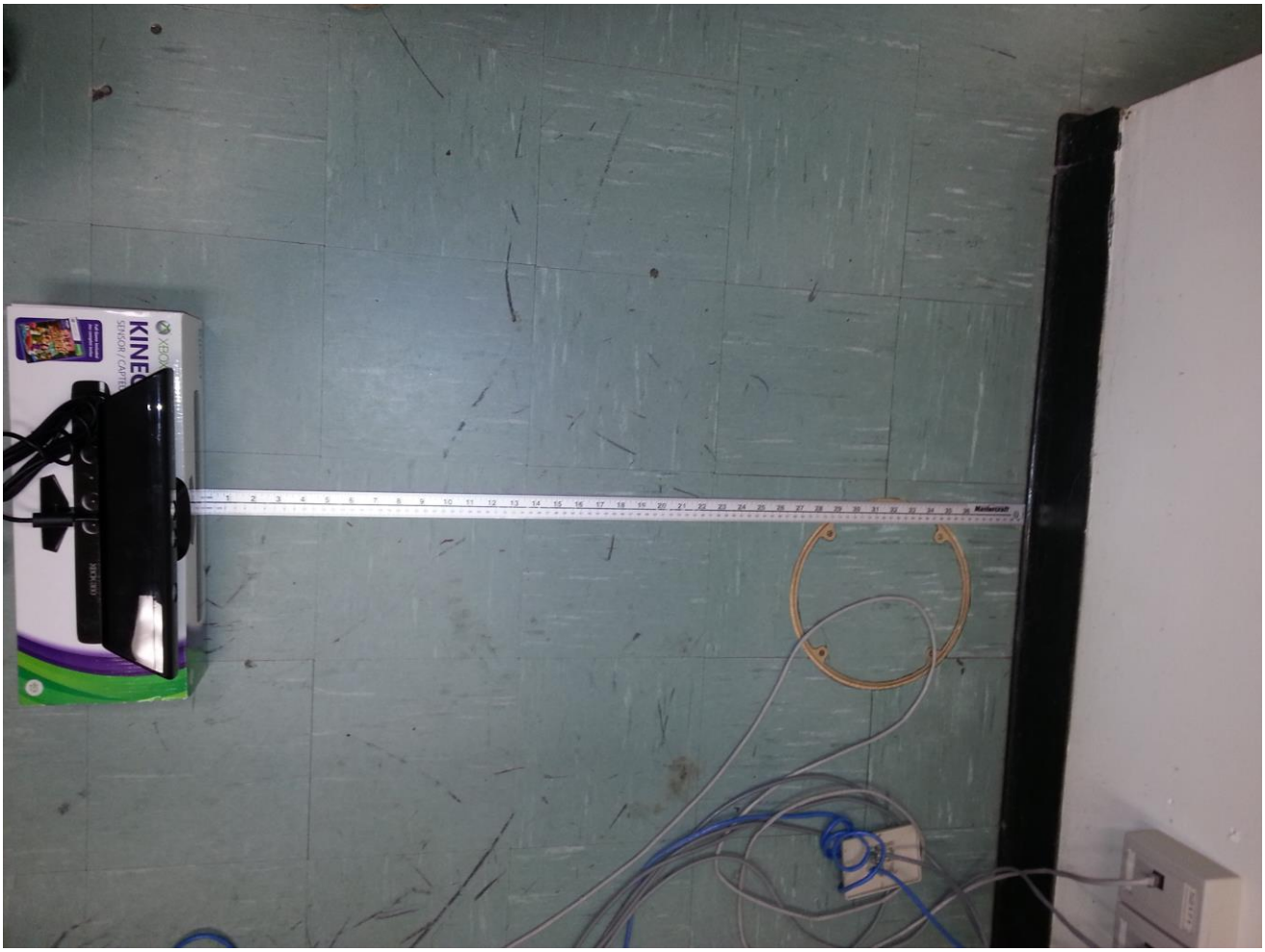
Attention : Matlab n'est pas l'outil optimal pour afficher une grande quantité de points en 3D. Il est fortement suggéré de sous-échantillonner le nuage de point obtenue. Par exemple, si votre matrice de nuage de point se nomme *pointCloud*, vous pourriez utiliser *pointCloud(1:2 :end,1:2 :end, :)* pour obtenir seulement ½ des rangées et ½ des colonnes.

Facultatif : Vous pouvez aussi afficher le nuage de point sous-échantillonné avec les couleurs RGB grâce à la fonction: *scatter3(vecteurX, vecteurY ,vecteurZ, tailleDesPoints, [vecteurRouge, vecteurVert, vecteurBleu], filled);* où les vecteurs sont des vecteurs colonnes. Notez que l'affichage est plus lent et qui faut idéalement utiliser moins de points. Notez encore une fois que ce n'est pas l'outil optimal d'affichage de nuage de points et que les couleurs sont difficiles à interpréter.

Partie 2 – Observation du bruit et des pas de quantification

2.1 – Selon l'axe z (profondeur)

Les données obtenues avec la Kinect, comme avec tous autres capteurs, sont bruitées. De plus, les profondeurs seront quantifiées (ne prendront pas des valeurs continues mais vont varier par des sauts fixes), avec un pas variant avec la profondeur. Afin de mieux voir ces effets, placez la Kinect sur une boite à environ un mètre du mur tel qu'illustré ci-dessous. Assurez-vous, autant que possible, que le montage soit parallèle au mur.



Les lignes de code suivantes (actuellement en commentaire) permettent d'afficher une croix noire au centre optique ainsi que deux lignes verticales (au 1/6 et au 5/6 de la largeur) dans l'image RGB:

```
c = axis();
line(c(1:2), [size(rgbImage,1)/2 size(rgbImage,1)/2],'Color','k');
line([size(rgbImage,2)/2 size(rgbImage,2)/2],c(3:4),'Color','k');
line([size(rgbImage,2)/6 size(rgbImage,2)/6],c(3:4),'Color','k');
line([5*size(rgbImage,2)/6 5*size(rgbImage,2)/6],c(3:4),'Color','k');
```

Les lignes suivantes (aussi en commentaire) permettent uniquement d'afficher la valeur en z (depth) aux trois croisements des lignes :

```
leftDepth = depthMap(round(size(rgbImage,2)/2), round(size(rgbImage,2)/6));
opticalCenterDepth = depthMap(size(rgbImage,1)/2, size(rgbImage,2)/2);
rightDepth = depthMap(round(size(rgbImage,2)/2), round(5*size(rgbImage,2)/6));
fprintf('Left:%f | Center:%f | Right:%f \n', leftDepth, opticalCenterDepth, rightDepth);
```

En vous assurant que la Kinect soit bien perpendiculaire au mur, exécutez le code et observez les valeurs de profondeur (z) affichées en boucle. Est-ce que ces valeurs sont stables? Est-ce que la stabilité est semblable au centre optique que dans les côtés? Quel est la grandeur (pas) des sauts? Au besoin, déplacez légèrement l'ensemble boîte-Kinect pour observer ces sauts fixes de distances en z.

Répétez la démarche avec la Kinect à 3 mètres du mur, puis à 5 mètres. Est-ce que la précision sur les mesures/grandeurs des pas est la même pour de plus grandes distances? D'après vos observations, est-ce que la taille des sauts est *a)* fixe *b)* croit linéairement avec z *c)* croit en z^2?

2.2 – Selon x

Le bruit dans une image produite par une caméra se présente sous plusieurs formes (e.g. distorsion radiale). Étant donné que le calcul de la position en x et en y pour chaque pixel dépend de la valeur bruitée en z et de la position du pixel dans l'image (elle aussi bruitée), il serait intéressant d'observer la précision des valeurs obtenues dans le nuage de point.

Placez le coté d'une boite au centre optique horizontal de la caméra et utilisez des règles (voir la figure ci-dessous) pour mesurer la position relative réelle d'un repère (e.g. le coin inférieur droit de la boite).

Toujours en utilisant le code de *labo6Kinect.m*, ajoutez la fonction *[x,y] = ginput(1)* pour obtenir l'indice du pixel choisi dans la figure. Vous pouvez maintenant créer le nuage de point grâce à la fonction *depthToCloud* et vérifier si la position mesurée selon x correspond à la valeur dans le nuage de point. Observez quelques mesures à cette position et répétez le processus pour une plus grande distance en x et en z. Est-ce que la précision est similaire dans les deux cas ?

Partie 3 – Extraction des surfaces planes : RANSAC (facultatif)

L'extraction de surfaces planes dans des nuages de points est une tâche clef dans le traitement et l'analyse de ces derniers. Dans un environnement intérieur, ces surfaces planes correspondront à des murs, au plancher, et à toutes autres surfaces planes présentes dans le champ visuel de la Kinect. Comme plusieurs équipes utiliseront la Kinect pour naviguer, une des premières tâches consiste à identifier le plancher dans un nuage de point. Cette méthode vous sera donc très utile pour vos projets!

Dans la première moitié du cours, nous avons vu l'algorithme RANSAC. L'extraction de surface plane est une application intéressante de RANSAC, car vous cherchez une hypothèse (un plan en 3D) dans un ensemble de donné bruités. La méthode consiste à piger, au hasard, 3 points dans le nuage. Ces trois points définissent, de façon unique, un plan en 3D. S'il y a peu d'inliers à ce plan, cette hypothèse n'est donc pas valide. Si, par contre, beaucoup de points sont des inliers, alors il y a de fortes chances qu'ils constituent un plan.

Algorithm 1 RANSAC

1: Select randomly the minimum number of points required to determine the model parameters N_{min}
2: Solve for the parameters of the model.
3: Determine how many points from the set of all points fit with a predefined tolerance ε_{Tol}.
4: If the fraction of the number of inliers over the total number points in the set exceeds a predefined threshold τ, re-estimate the model parameters using all the identified inliers and terminate.
5: Otherwise, repeat steps 1 through 4 (maximum of N times). $N \approx \frac{1}{w^{N_{min}}}$ w = prob. inlier

note: on peut aussi faire toutes les N itérations, et garder le modèle avec le plus grand nombre d'*inliers*

Adaptez l'algorithme RANSAC fourni par le fichier *PlanParRansac.m* pour extraire les principaux plans présents dans le nuage de points. Le plancher devrait correspondre au plan a) ayant un très grand nombre de points b) horizontal c) le plus bas. Il vous faudra en particulier trouver une valeur de tolérance ε qui soit assez grande pour tenir compte du bruit sur le capteur, mais suffisamment petite pour ne pas incorporer les surfaces raboteuse ni courbe.

Pour comprendre comment le code calcule la distance entre un plan et un point, vous pouvez vous référer au site web suivant : http://mathworld.wolfram.com/Point-PlaneDistance.html
Essentiellement, il faut commencer par calculer le vecteur normal unitaire $\hat{n}$ de ce plan par l'équation

$$\hat{n} = \frac{(\vec{x}_2 - \vec{x}_1) \times (\vec{x}_3 - \vec{x}_1)}{|(\vec{x}_2 - \vec{x}_1) \times (\vec{x}_3 - \vec{x}_1)|}$$

où les vecteurs $\vec{x}_i$ sont les 3 points choisi au hasard, $\times$ est le produit croisé et $|\ |$ la norme d'un vecteur. La distance d entre ce plan et un autre point $\vec{p}$ est $d = \hat{n} \cdot (\vec{p} - \vec{x}_i)$, où $\vec{x}_i$ est n'importe quel des trois points choisis au hasard, et $\cdot$ est le produit scalaire. Notez que le code *PlanParRansac.m* utilise au maximum le calcul matriciel, afin d'accélérer le traitement. Une boucle for-end en matlab sur une matrice est entre 40-100 fois plus lente que des opérations natives matlab sur cette matrice entière.

FIN

Note : afin de bien terminé l'exécution du code, vous devriez appeler le script *finalize.m* qui arrête l'appel des données de la Kinect et arrête la communication réseau.