Documentation projection LIDAR-RGBD

# Objectifs :

* Mettre en place un système de perception sur la plateforme roulante mise à disposition (Renault Twizy) capable de faire de la fusion de données RGBD-LIDAR

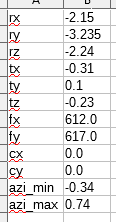
## Projection LIDAR-RGBD

* run\_projection() : args = None
* listener() : args = None
* callback() : args = image, lidar (*messages image et LIDAR)*
* convert\_c2s() : args = x,y,z *(coordonnées cartésiennes)*
* apply\_transform() : args = lidar\_point, transform\_matrix (*coordonnées cartésiennes du point + matrice pour la transformation LIDAR-image)*
* pixel\_color() : args = distance
* load() : args = path, delim, newline (*chemin du fichier csv et paramètres de fichier)*

L’objectif de ce module est de reprojeter sur l’image RGBD les points générés par le télémètre LIDAR. Ceci nécessite de considérer chaque point généré par le télémètre (exprimé en coordonnées cartésiennes dans le référentiel du LIDAR) et de :

1. Les transposer dans l’espace pour qu’ils s’expriment dans l’espace de la caméra RGBD via la matrice de transformation spatiale
2. Les projeter sur le plan image de la caméra via la matrice de projection

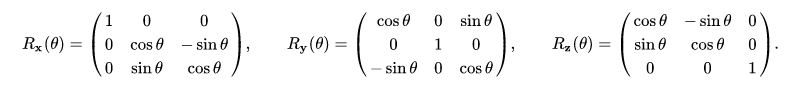
### Acquisition des matrices

Les coefficients des matrices utilisées sont stockés dans un fichier csv (délimité par des virgules et d’interligne \n). Il nous faut dans un premier temps les récupérer pour former les matrices considérées.

Le fichier csv utilise la structure vue ci-contre. Celle-ci ne doit pas changer car le programme de projection la présuppose. Les premières lignes contiennent les angles pour les rotations selon x, y et z. On a ensuite les coefficients de la transformation de translation selon chaque axe, puis les paramètres intrinsèques de la caméra. Enfin, les limites du champ de vue de la caméra sont définies en dernier.

Le logiciel utilise la fonction **load(***path, delim,newline***)**charger les matrices utilisées. On commence par charger le csv via la librairie python csv, puis on itère chaque ligne pour les charger dans un tableau. On ferme ensuite le fichier.

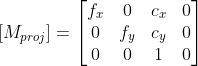
Les matrices de rotation sont ensuite formées avec la structure suivante (via wikipedia) :



La matrice de rotation complète est obtenue en multipliant les matrices de rotation élémentaire :

On peut ensuite charger la matrice de translation de structure :

La matrice de projection est ensuite chargée, de structure :



Les limites d’azimut sont également chargées et stockées.

Pour obtenir une unique transformation, on calcule déjà la transformation spatiale permettant de transposer les points du repère LIDAR dans l’espace RGBD, de format :

On calcule ensuite la matrice finale (permettant de passer directement de l’espace LIDAR à l’image) :

Cette matrice est ensuite rendue, en complément des limites d’azimut.

### Démarrage avec ROS

Pour être intégré dans l’interface graphique conçue pour le véhicule, la projection est lancée en appelant la fonction **run\_projection()**, qui fait simplement office de fonction main pour le module de reprojection. Cette fonction commence par créer un nœud ros (velodyne\_image\_projector) et un moyen de publier les informations de reprojection. Il lance ensuite la fonction **listener()**, qui fait office de boucle principale du programme.

### Boucle listener() et synchronisation

Comme dit précédemment, cette fonction représente la boucle principale du programme. Dans un premier temps, la fonction utilise la fonction **load()** décrite précédemment pour charger les matrices nécessaires au fonctionnement du programme. Ceci n’est fait qu’une fois.

On défini ensuite le temps maximal admissible entre des trames RGBD et LIDAR considérée comme simultanées (ici 50ms).

L’architecture logicielle utilise un bloc synchroniseur : c’est un bloc qui récupère les données générées par les capteurs de façon asynchrone et notifie le reste du système lorsqu’il détecte une trame LIDAR et une trame RGBD qui sont assez proches, temporellement parlant (en utilisant le délai de 50ms défini au préalable). En outre, le synchroniseur fournira à **listener()** une paire de trame synchronisée lorsqu’il en reçoit, et **listener()** attendra cet évènement jusqu’à ce qu’il arrive (la boucle).

Lorsque deux trames proches sont détectées, la fonction **callback()** est appelée.

### Callback()

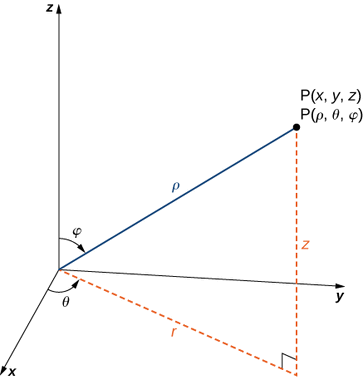
Cette fonction à pour objectif de traiter les données synchronisées obtenues en faisant la reprojection LIDAR. Elle reçoit en argument les trames LIDAR et RGBD synchronisés.

Dans un premier temps (opt), la fonction notifie l’utilisateur de la réception de trames valides via le terminal. La trame image générée par la caméra RGBD est ensuite immédiatement convertie vers un format compatible avec la librairie OpenCV2 (format BGR8) pour que l’on puisse effectuer un traitement d’image dessus. On en profite pour relever les dimensions de cette image.

Après vérification du format des données LIDAR, nous pouvons maintenant procéder à la reprojection. La boucle utilisée itère à travers chaque point LIDAR dans la trame et effectue les étapes suivantes :

1. On extrait les coordonnées cartésiennes du point LIDAR et on calcule l’azimut/la distance du point via une fonction permettant d’exprimer ce point en coordonnées sphériques (décrite plus bas)
2. Si l’azimut du point en question **n’est pas** compris entre les bornes d’azimut définies dans le fichier csv, on passe au prochain point. Dans le cas contraire, on applique la transformation LIDAR-image (fonction définie plus bas) pour obtenir les coordonnées u,v du point sur l’image.
3. Dans l’éventualité où ce point n’est pas sur l’image à l’écran (de coordonnées u,v négatives ou plus grandes que la largeur/hauteur de l’image), nous passons au prochain point. Dans le cas contraire, nous utilisons la librairie opencv2 et sa fonction **circle()** pour tracer un cercle au coordonnées correspondantes à ce point sur l’image. La couleur de ce point est déterminé par une fonction **pixel\_color()** définie plus bas.On passe ensuite au prochain point.

Ce protocole est appliqué pour l’ensemble des points de la trame LIDAR. Une fois effectuée, l’image compatible opencv2 est ensuite convertie vers un format compatible avec ROS (‘imgmsg’) sous format bgr8, et cette image est publiée.

Le contrôle est ensuite rendu à la boucle **listener()** qui attends le prochain couple de trames synchronisées.

NB: les angles phi et theta sont intervertis entre l'image et la notation utiliée par l'auteur

### Annexe : conversion cartésien/sphérique

On convertis les point exprimés en coordonnées cartésien ( au système sphérique ( via la fonction **convert\_c2s(x,y,z)** que j’ai définie. Cette fonction retourne seulement les équivalents sphériques de coordonnées cartésiennes.

La norme est simplement la distance entre le capteur et le point considéré, calculé :

L’azimut est simplement l’angle fait entre le vecteur et le vecteur unitaire, calculé :

**NB : on utilise la fonction atan2 sous python**

L’élévation , bien que non exploitée, est l’angle entre le vecteur unitaire et le point , calculé :

**NB : on utilise la fonction acos sous python et ce résultat n’est pas rendu par la fonction**

### Annexe : détermination de la couleur

Dans l’objectif de présenter des résultats visuellement clairs, on souhaitera pouvoir tracer des points sur l’image dont la couleur dépend de la distance. C’est pour cela qu’on définit la fonction **pixel\_color(***distance).*

Cette fonction ne fait qu’associer une couleur (format BGR8) à une distance.

On a déjà défini plus haut un canvas de couleurs BGR8 (*color\_map),* commençant d’abord par le couleur associé aux points proches puis pour les points de plus en plus éloignés de la voiture.

On émet l’hypothèse que l’utilisation de couleurs différentes est particulièrement utilisée à courte portée : on veut faire apparaitre différent un point à 10m d’un point à 2m ou 20m, mais distinguer des points à 70 et 80m l’un de l’autre à moins d’intérêt. C’est pour cela que l’on utilisera une unique couleur pour l’ensemble des points de distance supérieur à la moitié de l’étendue de mesure du télémètre (ici c’est 100m), en réservant le reste à pour les points proches du véhicule.

On divise l’étendue restante (les 50 premier mètres) en intervalles de distance de même taille, auquel on associe chacun une couleur. On peut par exemple dire que les points de 0 à 10m seront bleu foncé, puis bleus clairs pour 10 à 20m, etc.

On associe donc la distance d’un point LIDAR à une couleur, que l’on retournera ensuite. Dans le cas où une erreur se produit, le rouge est choisi comme couleur par défaut.

### Application de la transformation

On veut pouvoir exprimer les coordonnées des points du référentiel LIDAR dans l’image, ce qui nécessite d’appliquer la matrice de transformation définie plus tôt. C’est pour cela qu’on définit la fonction **apply\_transform(***lidar\_point, transform\_matrix***).**

Cette fonction prends comme argument une trio de coordonnées x,y,z cartésiens (dans le repère LIDAR) et retourne un couple de coordonnées (u,v) correspondant à un pixel sur l’image finale.

Dans un premier temps, les coordonnées fournies sont mises sous forme homogènes en définissant un vecteur colonne de dimension 4x1 :

Les coordonnées homogènes de ce point dans l’image finale sont ensuite obtenues en multipliant la matrice de transformation par ce vecteur colonne :

Pour obtenir les coordonnées d’un **pixel**(u,v), nous faisant l’opération suivante :

Pour ne traiter que des valeurs entières (opencv2 ne permet pas de dessiner sur sous-pixels), on utilise int() pour ne conserver que la partie entière de ces résultats. Les coordonnées sont ensuite rendues.

## Projection RGBD-LIDAR

FONCTIONS :

* run\_inv\_projection() : args = None
* report\_info\_callback() : args = data (message MN*)*
* dnn\_get\_info() : args = msg, matrix (messsage MN et matrice inverse)
* inverse\_transform() : args = tf\_matrix, coords (*coordonnées cartésiennes du point + matrice pour la transformation camera-LIDAR)*
* estimate\_orientation() : args = data (*message MN)*
* create\_marker() : args = id, object, xyz\_coords, ori\_coords (*identifiant rviz du marqueur, son type, ses coordonnées dans l’espace lidar et d’orientation)*
* place\_markers() : args = coords, types, orientation (listes de coordonnées dans l’espace lidar, types et orientations des marqueurs à dessiner)
* load() : args = path, delim, newline (*chemin du fichier csv et paramètres de fichier)*
* convert\_c2s() : args = x,y,z *(coordonnées cartésiennes)*

L’objectif de ce module est de reprojeter des informations générées par la caméra RGBD dans l’espace LIDAR. On veut en outre placer des markers (formes géométriques, ex cylindres, sphères…) dans l’espace LIDAR/RVIZ correspondant aux objets détectés par le système de vision MobileNet

# Préambule : MobileNet est un système de détection d’objets basé sur l’intelligence artificielle. Il a ici été pré entrainé pour détecter certaines classes d’objets (des voitures et des personnes actuellement). Le système est capable de placer des boites englobantes autour des objets détectés.

# L’utilisation des paramètres intrinsèques de la caméra et de ses capacités 3D permettent de retrouver la position dans l’espace des objets détectés par MN, vis-à-vis de la caméra.

MobileNet fournis entre autres des trames de détection contenant la position et la classes des objets détectés à un instant donné, ainsi qu’une multitude d’autres informations. C’est ces données que l’on cherchera à exploiter.

### Acquisition des matrices

Comme précédemment, nous avons besoin de créer une matrice correspondant à la transformation spatiale permettant de transposer les données d’un capteur dans l’espace de l’autre.

Les paramètres présents dans le fichier csv ont été utilisés pour quantifier la matrice permettant de transposer les points LIDAR dans l’espace de la caméra. On note entre autres la rotation permettant de transposer les points de l’espace LIDAR dans celui de la caméra et la translation permettant de transposer ces mêmes points. On cherche ici , nécessaires pour constituer la matrice permettant de transposer les points dans l’espace LIDAR depuis celui de la caméra RGBD.

On sait cependant que :

Avec la matrice finale permettant cette transposition :

Comme précédemment, une fonction **load()** est responsable de l’acquisition des paramètres stockés dans un fichier csv, de même format que pour la projection LIDAR-RGBD. Cette fonction construit les mêmes matrices que précédemment (rotation et translation LIDAR-RGBD) puis utilise les relations ci-dessus pour former la matrice . La matrice obtenue est la seule à être rendue.

#### Démarrage avec ROS

Pour assurer l’intégration dans l’interface graphique conçue pour la twizy, la fonction « main » de ce bloc prends la forme de la fonction **run\_inv\_projection().** Cette fonction commence par créer un tableau de markers que l’on utilisera pour publier les markers que l’on souhaite placer. Elle charge ensuite la matrice élaborée précédemment, puis crée un nœud ROS. Enfin, elle s’abonne au flux du détecteur d’objets et crée un moyen de publication de markers. Elle attend ensuite l’arrivée de trames MobileNet, appelant une fonction **report\_info\_callback()** quand une trame arrive.

### Report\_info\_callback()

Cette fonction est appelée lorsque MobileNet publie une nouvelle trame. Son fonctionnement est simple :

* Elle accèpte en argument une trame MN
* Elle extrait les coordonnées cartésiennes des objets détectés ainsi que leur type via une fonction **get\_info()**
* Elle supprime les markers précédents (s’ils existent)
* Via une fonction **place\_markers()**, elle remplit le tableau de markers que l‘on publiera plus tard
* Elle publie ce tableau de markers

### Fonction get\_info()

Cette fonction est appelée peu de temps après que MobileNet publie une trame, et a pour objectif d’extraire les informations utiles de ces trames, à savoir les coordonnées 3D des objets détectés ainsi que leur classe.

Dans un premier temps on extrait simplement des tableaux contenant les coordonnées x,y et z des objets dans la scène, ainsi que leur type. C’est 4 tableaux sont à priori de même taille (l’indice d’entrée de chaque tableau correspondant à un objets distinct).

Pour chaque objet détecté (leur nombre est égal à la longueur de chacun de ces tableaux) :

1. On forme un trio contenant les coordonnées cartésiennes de l’objet et on relève la classe de l’objet (personne, voiture…)
2. On calcule la distance de l’objet à la caméra via la même fonction de conversion cartésien-sphérique **convert\_c2s()** détaillée précédemment.
3. On choisit de ne pas afficher les objets lointains sous forme de markers, car ceux-ci sont souvent juste une conséquence d’une mauvaise mesure de distance de la part de MN/RGBD. Dans le cas où cette distance excède une distance seuil préétablie (typiquement 30m, mais réglable), on passe au prochain objet. Si la distance est valide, on effectue la transformation nécessaire via la fonction **inverse\_transforme()** détaillé plus bas, qui permet de récupérer les coordonnées dans l’espace LIDAR.

Il y a parfois des phénomènes de « sous-saturation », où l’objet détecté est trop proche de la caméra est affiche une distance de 0.0m. Si c’est le cas (l’objet est à l’origine), on le note comme un objet de classe « vide » (plus d’informations dans la partie dédié au placement de markers).

### Annexe : fonction inverse\_transform()

Cette fonction applique la transformation permettant de transposer les points de l’espace RGBD dans l’espace LIDAR. Dans un premier temps on forme un vecteur colonne homogène avec les coordonnées x,y,z de l’objet détecté, exprimé dans l’espace caméra :

On obtient ensuite les coordonnées homogènes dans l’espace LIDAR :

Pour obtenir des coordonnées cartésiennes classiques, on effectue l’opération suivante :

Ce trio de coordonnées (x,y,z) est ensuite retourné.

### Placement de markers

L’extraction de données du message MN étant fait, on veut créer des markers correspondant à ces données. Dans un premier temps le tableau de markers est vidé via des appels successifs de la fonction ‘pop’, s’assurant que le tableau soit vide et que des anciens markers ne soit pas encore à l’écran.

On appelle ensuite la fonction **place\_markers(***coords, types****),*** qui a pour fonction de remplir ce tableau de markers.

Dans un premier temps, on crée un tableau de markers « de travail » que l’on retournera plus tard. Une première approche consistait à dessiner placer les markers correspondant aux coordonnées données en argument. Malheureusement on retrouvait souvent des vieux markers qui ne disparaissaient plus. Pour pailler à ce problème, on forcera la publication de markers vide pour effacer les anciens markers qui pourraient rester. Concrètement, on publiera un nombre fixe de markers (mais en excès par rapport au nombre d’objets détectés dans une scène). Le tableau de marker à publier est donc tout d’abord rempli avec les objets détectés dans la scène (de coordonnées données en argument), puis l’espace restant (une fois qu’il n’y a plus d’objets) est rempli par des markers invisibles, placés à l’origine. Ces derniers ont une classe réservée, nommée ‘null’.

La création d’un marker à mettre dans le tableau de marker est géré par une fonction nommée **create\_marker(***id, object\_type, xyz\_coords,ori\_coords***)**, qui crée et retourne un unique marker de paramètre donné. On note que sous RVIZ, chaque marker possède un identifiant distinct (*id)* le démarquant des autres markers, et ils doivent chacun être différents.

Dans un premier temps, cette fonction crée un marker « de travail » en spécifiant son identifiant, son repère de référence (velodyne), ainsi que l’horodatage lors de sa création.

Nous avons choisi de différentier le type des objets en utilisant des markers de forme et de couleur différente (des sphères rouges pour les voitures, des cylindres verts pour les personnes). La classe de l’objet en question (les classes définis et le type ‘null’) est donc évalué pour fixer la taille, le type et la couleur du marker. La position et l’orientation sont ensuite fixées en fonctions des arguments de la fonction (orientation fixe par défault).

Le marker tout neuf est ensuite retourné et la fonction **place\_markers()** le place dans le tableau de markers. Ce tableau est ensuite retourné à la fonction callback.

### Détection de l’orientation

On souhaite pouvoir déduire l’orientation des objets détectés dans la scène a partir des prédictions émises par MobileNet, et en particulier leurs boites englobantes.

Pour l’instant, MN n’est capable de détecter que des voitures et des piétons. L’orientation des piétons n’ayant pas trop d’intérêt pour nous, on cherche à estimer l’orientation des voitures.

On fait la supposition que les véhicules dont l’avant ou l’arrière sont face à nous posséderont une boite englobante de forme plutôt carrée. On indique que le rapport entre la largeur et la hauteur est au voisinage de 1. En regardant des voitures dont le côté est face à la caméra, ce rapport passe plutôt aux alentours de 2 ou 3. On utilisera ce rapport pour estimer l’orientation du véhicule détecté.

Rviz et ses marqueurs défini la rotation des marqueurs tels que ‘1’ représente une rotation de 90°. Un objet d’orientation (x,y,z) = (0,0,0) est donc placé de manière parallèle à un axe de référence. Tel que nous avions construit l’application, avec des objets de taille (2,1,1), ceci représente des véhicules faisant dos (ou face) à la caméra. Un véhicule d’orientation (0,0,1) est cependant tourné tel que son côté soit face à la caméra.

On définit les quantités suivantes :

* Un rapport minimal *min\_wh* correspondant au rapport largeur/hauteur d’un véhicule orienté dos ou face à la caméra
* Un rapport maximal *max\_wh* pour un véhicule orienté de côté.
* *Delta,* la différence entre ces deux rapports

Après avoir quantifié le rapport entre la largeur et la hauteur de la boite englobante (qu’on contraint à l’intervalle [min\_wh, max\_wh]), on calcule ensuite la différence entre ce rapport et *min\_wh*.

Logiquement, si le résultat obtenu est nul, ceci indique que le véhicule considéré fait face ou dos à la caméra (puisque son rapport largeur/hauteur est égal à *min\_wh*). Cependant, si ce rapport largeur/hauteur est égal à *max\_wh*, la différence calculée devrait être égale à *delta.*

En raison de la contrainte précédemment fixée sur la valeur de ce rapport largeur/hauteur, la différence calculée ici sera forcément comprise entre 0 et *delta.* Pour ramener cette quantité entre 0 et 1, on la divise donc par *delta.* On note cette quantité *norm.*

Ce dernier résultat est ensuite utilisé pour fixer l’orientation d’un marqueur : l’orientation du marqueur associé à ce véhicule prends ensuite comme expression (0,0,norm).

### Publication

Une fois en possession d’un tableau de marker frais, la fonction callback publie ce marker sous la forme d’un topic *markerArray*, que l’on pourra ensuite visionner sous RVIZ. Le programme attend ensuite la prochaine trame MN pour créer de nouveau markers.

## Détection du sol via transformée de Hough

FONCTIONS

* run() : args = None
* listener() : args = None
* callback() : args =lidar, image (*messages image et LIDAR)*
* convert\_c2s() : args = x,y,z *(coordonnées cartésiennes)*
* fit\_for\_hough() : args = plane\_coords, buffer\_size (*coordonnées dans le plan yz du point lidar et taille de l’image pour la transforme de hough)*
* apply\_transform() : args = lidar\_point, transform\_matrix (*coordonnées cartésiennes du point + matrice pour la transformation LIDAR-image)*
* pixel\_color() : args = distance
* load() : args = path, delim, newline (*chemin du fichier csv et paramètres de fichier)*

### Objectifs et acquis théoriques :

La détection d’indicateurs dans l’environnement du véhicule, tel que des marquages au sol, peut nécessiter de repérer et délimiter la chaussée dans le flux vidéo du capteur. C’est ce que l’on veut pouvoir entreprendre ici : délimiter (via une reprojection LIDAR par exemple) la route dans le flux vidéo des caméras embarquées.

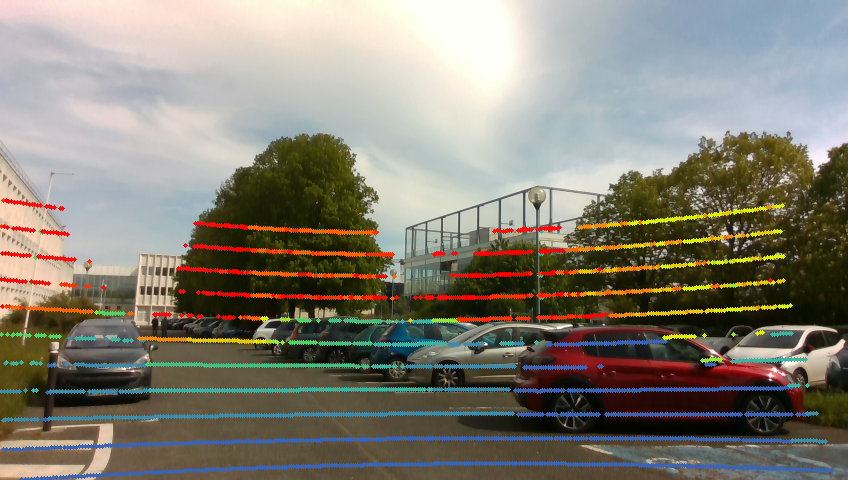
Ce travail s’avère d’autant plus compliqué en raison de la forme de la route (qui n’est pas parfaitement plane) et des mouvements du véhicule (tangage, roulis…). On ne peut tout pas supposer que les capteurs sont solidaires vis-à-vis de la route : il faut pouvoir la repérer dans les données des capteurs.

Mais comment faire ?

On voudrait pouvoir détecter le plan routier, plan qui est traduit dans les données lidars générés par le système. Mais comment repérer ce plan dans les données, sachant que les données LIDAR ne sont que des nuages de points ?

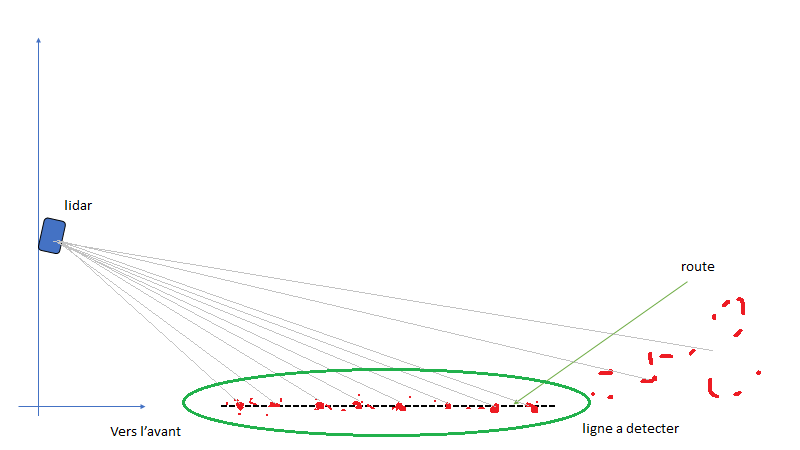
On utilisera la **transformée de Hough.**

Considérons les données LIDAR ci-présentes :



On rappellera que ces informations, bien que projetées sur ne image, contiennent tout de même une information concernant leur profondeur : certains points sont plus loin que d’autres.

Ainsi, si on considère un plan orthogonal à l’image (coupant le véhicule en deux dans le sens de la largeur de manière symétrique) et que l’on projète tout les points LIDAR sur ce plan, nous devrions obtenir une image de la forme suivante :



Les données LIDAR ne correspondant pas à des plans sont éparpillées sur l’image. Cependant, on note que les points présents sur le plan routier (et n’importe quel plan dans l’espace en réalité) sont relativement alignés dans cette représentation. La détection de plans dans l’espace nécessite donc de pouvoir détecter cette ligne pour ensuite évaluer la correspondance entre les données LIDAR et le plan correspondant. De plus, il faudra identifier le seul et unique plan routier, et ne pas détecter d’autres plans (ex : des murs).

La transformée de Hough est un bon outil pour détecter des lignes droites à partir de nuages de points. C’est ce que l’on utilisera ici, de plus que l’algorithme en question est implanté dans openCV2. On s’en servira pour détecter la route dans la scène.

### Démarrage

*Params = None*

Le lancement du programme est assuré par une fonction **run()** comme chaque programme décrit précédemment. Cette fonction commence par charger les paramètres extrinsèques et intrinsèques de la caméra pour la reprojection éventuelle, et crée un nœud ROS ainsi que les objets de publications qu’on aura besoin plus tard.

On lance ensuite une fonction **listener()** qui servira de boucle principale.

### Acquisition des matrices

La fonction load() utilisée ici est identique à celle utilisée pour la projection LIDAR. Elle charge une matrice à partir des données csv permettant de transposer les données LIDAR su l’image finale, ainsi que les limites azimutales de la caméra (« FOV »).

### Boucle listener()

Cette fonction servira de boucle principale. Dans un premier temps, le programme s’abonne aux données LIDAR et RGBD. Le synchroniseur est ensuite configuré pour opérer sur ces données lorsque des données synchronisées sont trouvées (< 50ms). Lorsque c’est le cas, un callback est appelé.

### Callback()

Cette fonction opère sur les données LIDAR et RGBD synchronisées précédemment.

Dans un premier temps, les informations RGBD sont converties vers un format compatible opencv2 pour un traitement ultérieur.

La transformée de Hough telle qu’elle est implémentée dans opencv2 opère sur des *images :* elle reçoit en argument une image et elle rend les paramètres des droites détectées, ordonnées par ordre décroissant de votes. On veut donc projeter les données LIDAR sur un plan (comme vu en introduction), puis ensuite fournir l’image correspondant à cette projection à opencv2 (on peut par exemple dire que 1m = 1 pixel depuis l’origine).

Un bémol : opencv2 à du mal à détecter la ligne souhaitée dans cette première image, et détecte souvent des lignes ne correspondant pas à la route. Pour faciliter la détection de cette ligne, on souhaitera étirer les points pour faire apparaitre des lignes plus définies.

On définit donc **l’espace plan**, un plan 2D contenant les données de cette première projection, mais étirées horizontalement et verticalement pour faciliter la détection.

Concrètement, on commence par définir une image de résolution relativement élevée (750 par 3000 pixels) complétement noire. On itère ensuite à travers les données LIDAR, point par point, en éliminant les points qui ne sont pas dans le champ de vue de la caméra. De plus, on ne considère qu’un sous-ensemble de points dans le champ de vue : ceux à portée relativement courte. Les points à longue portée ont peu de chances de correspondre au plan routier et donc augmentent les chances qu’un plan autre que le plan routier soit détecté (et ils ralentissent le logiciel). On élimine également les points très proches pour s’assurer que le véhicule lui-meme n’est pas détécté. Les bornes entre lesquelles les points sont conservées est défini dans la partie paramétrage du logiciel, mais sont pour l’instant fixés à 0.5 et 10m.

On jette les coordonnées selon x des points restants, puis on applique l’opération suivante pour trouver les coordonnées (u,v) de ce point dans l’espace plan :

On ne conserve ici que les parties entières de ces coordonnées. **Width** n’est autre que la largeur de l’espace plan en pixel (3000 ici).

Les coefficients utilisés ici (à la fois multiplicatifs et diviseurs) n’ont pas de signification particulière. Ils ont été choisis selon deux critères :

* Que les points LIDAR soient bien centrées dans cet espace plan et bien visibles
* Que la transformée de Hough soit facilement capable de détecter les lignes désirées.

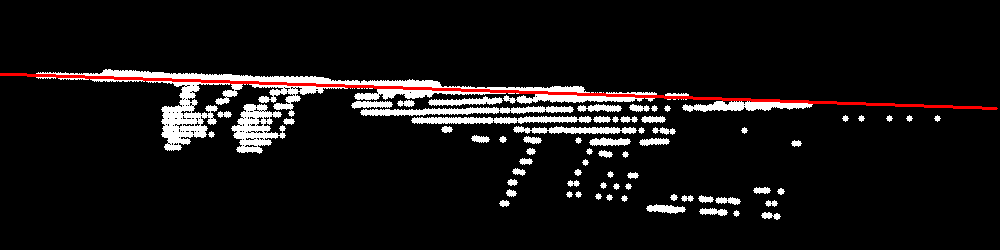
**NB : dans notre programme, cette opération est réalisée par une fonction, fit\_for\_hough()**

Un cercle blanc est ensuite placé aux coordonnées calculées. On obtient au final une image noire parsemée de points blancs. Cette image est ensuite fournie à la fonction de transformée de Hough **houghlines()** de opencv2, qui détecte ensuite les lignes présentes dans l’image.

On veut ensuite déterminer deux points de cette droite pour une utilisation ultérieurs, d’où l’opération suivante :

**Given**

On peut ensuite tracer cette ligne sur l’espace plan via la fonction de dessin de lignes de opencv2, voire même publier cette image finale pour le diagnostic :



NB : la droite rouge correspond bien avec la droite correspondant au plan routier.

Maintenant que la droite recherchée est caractérisée (et donc le plan routier), on veut ensuite déterminer si les données LIDAR enregistrées correspondent à peu près (ou pas du tout) à ce plan afin de décider s’il faut les projeter ou non.

Plutôt que de générer le plan routier à partir de la droite obtenue précédemment (nécessitant de la géométrie plutôt poussée et beaucoup de calculs), on choisira plutôt de projeter les points LIDAR dans l’espace plan pour évaluer leur distance à la droite déjà établie. Si cette distance (en pixels) est inférieure à un seuil préétabli, on considèrera que le point LIDAR est sur la route, et il sera ensuite projeté.

La distance entre le point aux coordonnes (u,v) et la droite est la suivante :

Si la distance est suffisamment faible, le point LIDAR est projeté sur l’image. Cette opération est effectuée pour tout les points LIDAR dans le champ de vue de la caméra, avec une couleur dépendant de la distance du point au LIDAR (voir les fonctions **pixel\_color()** et **apply\_transform()** correspondante dans la partie reprojection).

L’image obtenue après reprojection est ensuite convertie en format compatible ROS, puis elle est publiée.



## Détection du sol via transformée de Hough + détection de trottoirs

FONCTIONS

* run\_road\_detect() : args = None
* listener() : args = None
* callback() : args =lidar, image (*messages image et LIDAR)*
* convert\_c2s() : args = x,y,z *(coordonnées cartésiennes)*
* fit\_for\_hough() : args = plane\_coords, buffer\_size (*coordonnées dans le plan yz du point lidar et taille de l’image pour la transforme de hough)*
* apply\_transform() : args = lidar\_point, transform\_matrix (*coordonnées cartésiennes du point + matrice pour la transformation LIDAR-image)*
* load() : args = path, delim, newline (*chemin du fichier csv et paramètres de fichier)*
* extrapolate\_from\_rings : args = (point\_y, rings, ring\_nbr)
* get\_ring\_coords, args = (lidar\_ring\_coords) (coodonnées xmax, xmin, ymax et ymin de chaque anneau lidar)
* color\_stats : args= img, pixel, max\_idx (image, coordonnée du pixel concerné et taille max de la banque de stats)
* between\_sides : agrs = point, left\_side, right\_side, draw\_distance (point de l’image, limites a gauche et à droite de l’axe x délimitant la route, distance sur laquelle tracer la route)
* detect\_peaks : args = None
* stats : args = x (coordonnée x d’un point LIDAR)
* fill\_road : args = img, color\_avg\_bgr, color\_std\_bgr, rings, mode, left\_side, right\_side (image de travaille, infos statistiques sur les couleurs du plan routier, coordonnées des anneaux, mode de fonctionnement, distance selon l’axe x séparant le véhicule du trottoir)
* draw\_lines, args = img, transform, left\_side, right\_side, draw\_distance (image de travail, transformation LIDAR-image, distance selon x séparant le véhicule du trottoir, distance sur laquelle tracer la route)
* framerate : args = None
* iter\_velodyne : args = lidar (un message LIDAR)
* seg\_intersection : args= seg\_a, seg\_b (deux ségments définits par une paire de coordonnées)

L’application précédente nous à permis de détecter le plan routier via transformée de Hough. On se propose de faire deux choses supplémentaires ici :

* L’application ne distingue pas bien la route lorsque celle-ci est à un niveau similaire du trottoir. On voudra pouvoir détecter la route à l’aide d’une deuxième méthode pour affiner notre détection du plan routier. On veut ici entre autres pouvoir surligner le trottoir.
* Le LIDAR PUCK16 implanté sur notre véhicule ne possède que peu d’anneaux. Nous souhaiterions pouvoir surligner la route : on utilisera les informations statistiques sur la couleur de la route pour surligner l’ensemble des pixels ayant une couleur similaire sur le plan routier.

Comment faire ?

Le principe derrière la détection de Hough ayant déjà été exploré dans la section précédente (et n’ayant pas changé dans son implémentation), on ne le détaillera pas.

La détection de trottoirs repose sur un principe semblable. On s’intéressera cette fois ci aux coordonnées x des points LIDAR, et plus particulièrement à leur distribution (en d’autres termes la fréquence d’apparition de certaines coordonnées x sur un nuage de point). Si notre véhicule était situé sur un plan parfaitement plat, on s’attendrai à obtenir une distribution de coordonnées relativement uniforme, les anneaux LIDAR traçant des cercles relativement uniformes autour du véhicule.

Cependant, si le véhicule se situe sur un terrain muni de reliefs (tel que des arbres, ou des bords de route…), on pourra remarquer un *pic* dans cette distribution, car une multitude d’anneaux LIDAR intersectent le trottoir supposé parallèle à la direction de trajet :

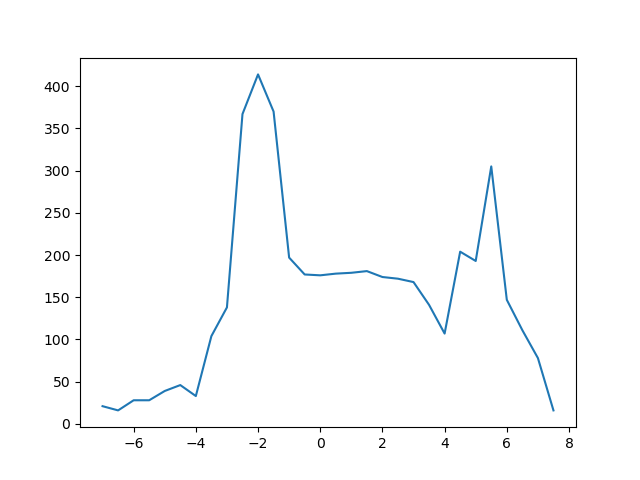


Figure : coordonnée selon x en abscisse, fréquence d'apparition (nombre de points dans la classe de distance) en ordonnée

On voit ci-dessus deux pics dans la distribution. Ils sont la manifestation de la présence de trottoir et autres obstacles le long de la route. Il suffit donc de les détecter automatiquement (chose que nous détaillerons plus tard) pour estimer la distance latérale du trottoir au véhicule.

Etant capable de discerner l’ensemble des points LIDAR appartenant au plan routier, on peut commencer à documenter certaines statistiques concernant la couleur de ce plan routier pour ensuite généraliser la détection routière à l’ensemble de l’image. On cherchera ici à déterminer la couleur moyenne (par canal R, G et B) du sol, ainsi que son écart type.

On pourra ensuite parcourir les pixels de l’image (ou au minimum un sous ensemble susceptible de contenir le plan routier) pour ensuite marquer ceux de couleur arbitrairement proche de la couleur moyenne.

### Boucle principale

En l’attente d’un couple de données LIDAR-vidéo synchronisées, le programme attends. Lorsque des données convenables sont reçues, on appelle la fonction *callback().*

Dans un premier temps, on définit des tableaux en mémoires qui stockeront :

* La fréquence des coordonnées des points LIDAR
* Les classes de distances considérées pour la distribution de coordonnées x (le nombre de classes et leur étendue étant défini par l’utilisateur)
* Les coordonnées (xmax, xmin, ymax, ymin) des anneaux LIDAR

Comme pour le logiciel précédent, on convertit l’image vidéo en format opencv2 pour travailler avec. On génère aussi une image noire pour la transformée de Hough, comme précédemment également

La première manipulation consiste à trouver le plan routier et (éventuellement) le trottoir.

La détection du plan routier est inchangée par rapport au logiciel précèdent : nous ne la détaillons pas. Cependant, en itérant à travers le nuage de point, on récupère la distribution de coordonnées x en notant la fréquence à laquelle les points tombent dans chacune des classes de distance qui ont étés prédéfinies. Si par exemple on décide de considérer les points de coordonnées x par exemple, et que l’on défini 40 classes de distance distinctes (ici d’étendue 0.5m), et qu’un point LIDAR est déterminé comme appartenant à la *i-ème* classe (0 < i < 40), alors un compteur traquant la fréquence de points dans la *i-ème* classe sera incrémenté.

On pourra ensuite détecter les pics éventuels à l’aide de la librairie *scipy:signal* et de sa fonction *findpeaks().* Ici, on a développé la fonction *detect\_peaks()* pour appeler cette fonction scipy puis pour traiter les données résultantes (de sorte à n’avoir que les coordonnées x des trottoirs).

Entre autre, cette fonction *detect\_peaks()* appèle la fonction find\_peaks de *scipy* pour ensuite récupérer une liste de pics dans la distribution dont les propriétés correspondent à ce que l’on cherche (réglable) . Il y a ensuite trois cas possibles :

* Si le nombre de pics est inférieur à deux, on ne peut rien faire : la distribution de coordonnées ne permet pas de distinguer un trottoir.
* Si le nombre de pics est exactement égal à deux, on conserve ces pics sans traitement particulier.
* Si le nombre de pics est supérieur à deux, on ne conserve uniquement que les pics le plus proches de la coordonnées x = 0 (le centre du véhicule). Ceci découle d’une hypothèse que le plan routier est plat, et donc qu’il ne devrait pas y avoir de pics proches de x = 0m avant ceux correspondant au trottoir.

Dans le cas ou deux pics sont extraits, on pourra noter la coordonnée de chaque pic, qui est théoriquement la distance latérale du véhicule par rapport à un bord de la route. On retourne ces valeurs au programme en complément d’un drapeau indiquant leur validité. Dans le cas contraire, un drapeau attestant de leur non-validité est rendu.

Si l’utilisateur le demande, on pourra tracer les lignes de fuite correspondant aux bords de la route à partir de ces détections. Pour ce faire, on développe la fonction *draw\_lines()*, qui défini quatre points dans l’espace environnant la voiture :

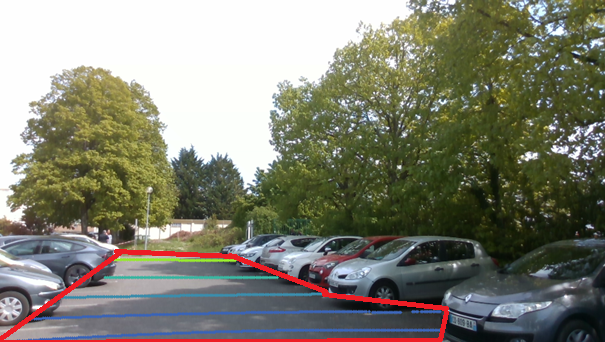
* Deux points ‘bas’ dont les coordonnées x correspondent aux pics définis précédemment, de coordonnée y = 0m et de coordonné z = -1.8m (approx la hauteur du LIDAR par rapport au sol). Ces deux points coïncident avec le bord de la route de part et d’autre du véhicule.
* Deux points ‘haut’ de même coordonné x et z que précédemment, mais de coordonnée y d’environ 20m. On applique ensuite une compensation à z pour prendre en compte l’inclinaison du LIDAR. Ces points coïncident avec le bord de la route à 20m du véhicule.

On applique ensuite la transformé lidar-image à ces points pour obtenir des coordonnées images correspondantes.

Grâce à ces 2 paires de points on pourra tracer deux segments modélisant la bordure de la route sur l’image.

Dans un deuxième temps, on itère une seconde fois à travers le nuage de points LIDAR. Cette fois ci, on note également l’indice de l’anneau qui contient le point LIDAR considéré. Ces données serviront à récolter des statistiques sur les anneaux LIDAR qui nous sera utile plus tard. Comme précédemment, nous calculons la distance de chaque point LIDAR au plan par l’intermédiaire de l’espace plan. On ne conservera uniquement que les points sur le plan routier. Si l’utilisateur le désire, ces points peuvent être tracés sur l’image finale.

Nous nous intéressons aux coordonnées (xmax, xmin, ymax, ymin) des anneaux LIDAR pour assurer une meilleure démarcation du plan routier. Une fois les statistiques sur la couleur du plan routier en main, on ne souhaite tout de même pas évaluer chaque pixel de l’image pour vérifier sa couleur, car ce serai trop long et provoquerai plein de faux positifs. Au contraire, on ne considérera que l’espace délimité par les données LIDAR du plan routier (le polygone rouge) :



Nous avons donc défini une fonction *get\_ring\_coords* dont le but est d’évaluer les coordonnées de chaque anneau à partir des statistiques récoltées précédemment.

Si des statistiques couleurs ont déjà été récoltées, nous pouvons passer au marquage de la route à partir des données en notre possession.

Ceci est assuré par la fonction *fill\_road().*

### Affichage via fill\_road

Cette fonction se charge de surligner le plan routier sur l’image RGBD. Dans un premier temps, on considère la coordonnée y minimale du plan routier sur l’image : l’origine de l’image étant ‘en haut à gauche’ de l’image, rien ne sert de scruter le ciel (les coordonnées y faibles) pour y détecter la route. On se sert pour cela de corodonnée ymin de l’anneau le plus haut sur l’image pour cela (qui est en réalité le dernier dans la liste contenant les coordonnées des anneaux.

Nous définissons ensuite à l’aide des statistiques couleurs l’intervalle r,g et b dans lequel doit exister tout point appartenant au plan routier. On définit cet intervalle comme :

Ici, est la couleur moyenne d’un des canaux couleurs (r,g ou b), est l’écart type associé à ce canal, et k est une constante fixée par l’utilisateur (on utilise 0.85 pour les meilleurs résultats).

On procède ensuite ligne par ligne de l’image à partir de la coordonnée y minimale déterminée précédemment, en procédant vers là-bas de l’image (y croissants). Pour chaque ligne, l’opération suivante est effectuée :

1. Si la détection de coté avait été assurée, on relève les coordonnées x limite (sur l’image) des bords de la route associés à cette ligne y. Ceci est obtenu par interpolation (voir ci-dessous).
2. On associe la ligne en question à un anneau LIDAR en relevant le prochain anneau situé au-dessus (sur l’image, c’est-à-dire de ymin plus faible) de cette ligne. On relève ensuite les coordonnées x limites associés à cette ligne comme précédemment, par interpolation.
   1. Dans le cas où la détection de bord routier est assurée, on ne relève finalement que le segment défini comme l’intersection des segments associées à ces limites , via la fonction **seg\_intersection** (autrement dit la zone où les deux systèmes de détection ont détectés la route).
3. On itère ensuite sur ce segment selon les coordonnées x, en sautant éventuellement des pixels pour des raisons de performance. Si le pixel considéré est de couleur proche du plan routier, on trace un point à la coordonnée correspondantes sur l’image, de couleur arbitraire (fixée par l’utilisateur).
4. On passe ensuite à la prochaine ligne.

L’image ainsi obtenue est ensuite publiée.

### Annexe : Interpolation

Le procédé de traçage nécessite parfois « d’interpoler » les données que l’on possède pour éviter des artefacts d’affichage (effet d’escalier par exemple). Considérons les deux anneaux suivants par exemple :



Ils sont chacun définis par deux couples de coordonnées (x,y). On considère qu’un point situé entre ces deux anneaux « appartient » à domaine de l’anneau vert. Dans le cas d’un point situé au centre entre ces deux anneaux, il n’y a aucun problème : le point est évidemment contenu dans le polygone délimité par les anneaux LIDAR.

Mais que faire dans le cas suivant ?



Le point est dans le domaine de l’anneau vert, mais ne sera pas tracé car sa coordonnée x excède la coordonnée xmax de l’anneau vert (démarqué par le segment rouge sur l’image). A l’affichage, on obtiendra donc une sorte d’éffet « en escalier », où le surlignement de la route s’arrête brutalement.

On cherchera donc à « interpoler » les données : on veut, en bref, qu’un point présent entre ces deux anneaux soit limité en fonction de sa coordonnée y :



Pour cela, on définit la notation suivante :

* la coordonnée x maximale de l’anneau du dessus
* la coordonnée x maximale de l’anneau de dessous
* la coordonnée y maximale de l’anneau du dessus
* la coordonnée y maximale de l’anneau du dessous

On cherche simplement à déterminer, sur la droite rouge ci-dessus, la coordonnée x associée au point d’interception de la droite avec y.

Cette coordonnée est obtenue :

On pourra ensuite comparer la coordonnée x de chaque point sur cette droite y à afin de déterminer s’il faut le considérer ou non.

Cette idée est d’abord utilisée dans le contexte des anneaux LIDAR, mais peut être étendue dès lors que l’on cherche à tracer une droite entre deux points de l’image. On peut donc utiliser cette même technique pour déterminer si un point de l’image se situe à sur la route, connaissant quatre points délimitant le bord de la route (en supposant celle-ci relativement droite).

Les résultats sont les suivants :



Figure : détection de sol via transformée de Hough. En violet les anneaux LIDAR reprojetés, en rouge les bords de la route.