Rapport du SLAM projet du IFT6145
Utiliser caméra monoculaire pour corriger la direction du robot et mesurer la profondeur en temps réel

Zhibin Lu

D.I.R.O de l'Université de Montréal

Montréal, Canada

Courriel : zhibin.lu@umontreal.ca

Résumé — **Ce projet met en œuvre plusieurs objectifs du SLAM en utilisant une caméra monoculaire, notamment la poursuite des points caractéristiques (tracking) de la scène, le calcul de la profondeur des points caractéristiques, le calcul de l'angle de mouvement du robot et enfin la correction de la direction du robot en temps réel.**

Mots-clés — **slam, caméra monoculaire, tracking, opencv, géometrie épipolair, mesure triangulaire, flux optique.**

I. INTRODUCTION

La technologie SLAM (Simultaneous localization and mapping) s'est développée rapidement ces dernières années et est devenue une branche d'application importante de la Computer Vision 3D. Il joue un rôle central dans la conduite autonome et dans divers types de fabrication robotique, tels que les robots de balayage.

Dans ce final projet du cours Vision Tridimensionnelle, nous utilisons un petit robot véhiculaire pour implémenter quelques buts du SLAM. Ce sont les environnements matériels et logiciels, ainsi que les objectifs expérimentaux.

1. **Environnements matériels et logiciels :**
   Matériels principaux pour le robot :
   - Raspberry pi 3,
   - Adafruit DC & Stepper Motor HAT for Raspberry Pi - Mini Kit
   - SainSmart Objectifs Grand Angle caméra fish-eye pour Raspberry Pi Arduino
   - 2 DC moteurs

   Logiciels principaux:
   - Raspbian Linux distribution 4.14.50-v7+
   - Adafruit-Motor-HAT-Python-Library
   - OpenCV 3.4.3
   - Python 3.6.6 et 3.5.3
   - Numpy 1.15

2. **Les objectifs expérimentaux :**
   - Faire poursuite(tracking) selon une vidéo venue de robot et en temps réel;
   - Corriger le trajet du robot en ligne droite en temps réel;
   - Corriger la direction du robot en temps réel.
   - Mesurer la profondeur pour les points de poursuite;

Notez que cette expérimentation utilise une résolution de 960×540 .

II. OBJECTIF DE LA POURSUITE

Calibration et Rectification

Tout d'abord, on doit calibrer la caméra, alors qu'en raison d'un fisheye dans notre environnement, on doit rectifier la distorsion. On utilise la méthode du patron non structurés pour obtenir en même temps la $\boldsymbol{K}$ interne matrice de la caméra modèle et le *LUT* (lookup table) entre caméra image coordonnée et écran image coordonnée.

Pour la méthode du calibrage planaire, il faut au moins **2** ***H*** homographies pour solver une ***W*** matrice de **4** degrés de liberté, et pour chaque homographie, il faut au moins **4** paires de points pour la ***H*** matrice de **8** degrés de liberté. Donc on prend **3** paires du *LUT* image, ce qui équivaut à **3** photos, ensuite on prend une de ces **3** paires comme position initiale, et les deux paires lui correspondent peux nous donner **2** ***H*** homographies, puis on choit **4** paires de points différentes (note que on doit choisir dans la zone effective de *LUT* image et évite de faire les points en ligne). On obtient la ***K*** interne matrice par la fonction *calibrateCamera* de *OpenCV3*.

```
[[334.08    0.      526.20]
 [  0.    283.12    337.45]
 [  0.      0.        1.   ]]
```

Table : La **K** interne matrice

Après obtenu le *LUT* image, on essayer de rectifier une image (frame) du vidéo du robot.

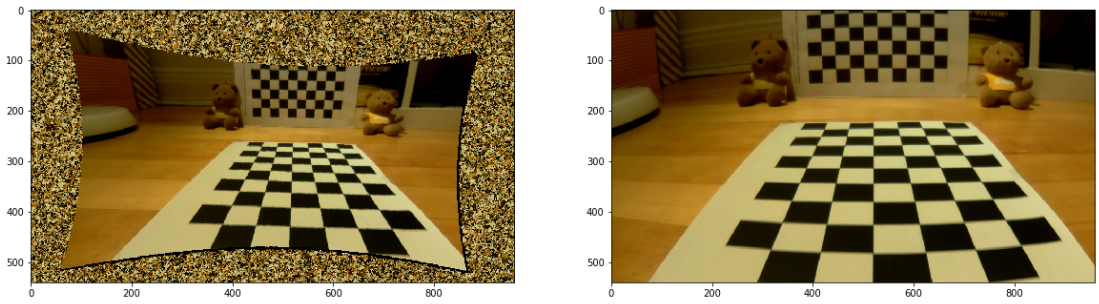


Figure : Rectification d'une image

Notez que on doit utiliser le *LUT* de projecteur(écran) à la caméra pour obtenir la valeur RGB de pixel de l'image de la caméra correspondant à une coordonnée de l'écran, puis remplir cette valeur à la coordonnée d'une image de l'écran.

Poursuite (Tracking)

La poursuite est à la base de notre objectifs suivants et l'effet de la poursuite a un impact important sur les objectifs suivants. Le flux principal de la poursuite est très simple: utiliser la fonction *goodFeaturesToTrack* de OpenCV pour obtenir les points caractéristiques de la première image de la vidéo, puis utiliser la fonction *calcOpticalFlowPyrLK* de OpenCV pour chaque image suivante afin de détecter la position actuelle des points caractéristiques obtenus. Le figure suivant est une poursuite sur une vidéo du robot qui roule vers **2** damiers.

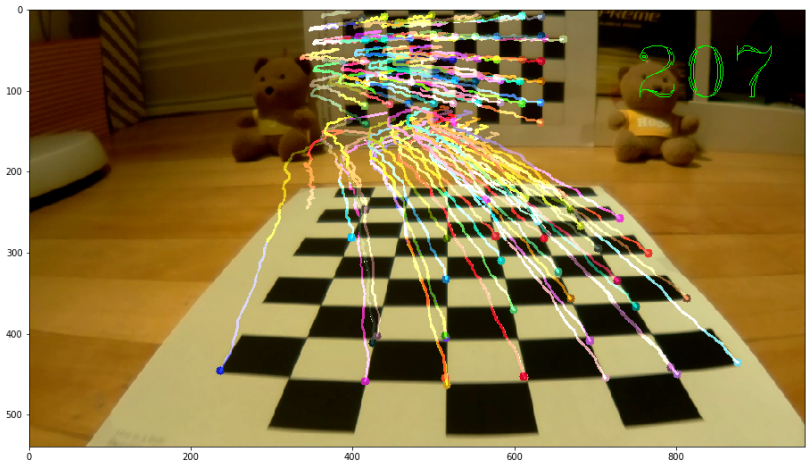


Figure : Poursuite

On constate que c'est une bonne poursuite, il détecte continuellement les mêmes points avec succès, sans les ne perdre ni suivre à les autres. Car cette vidéo est courte et suffisamment claire (moins floue); le robot roule lentement (la vitesse égale **90**[1]); l'intensité de la lumière est fondamentalement la même; de plus nous utilisons **2** damiers pour réduire la difficulté de poursuivre.

En fait, dans un environnement de SLAM en temps réel, il n'est pas toujours facile de poursuivre, car les conditions ne sont pas toujours comme celles ci-dessus, de plus dans nos expériences, limitées par les conditions matérielles (puissance de calcul, acquisition lente de la caméra, etc.), deux problèmes majeurs se posent : 1. L'image est plus floue, 2. Le changement entre les deux images est trop grand. Ci-dessous est un échec de la poursuite sur deux images adjacentes.

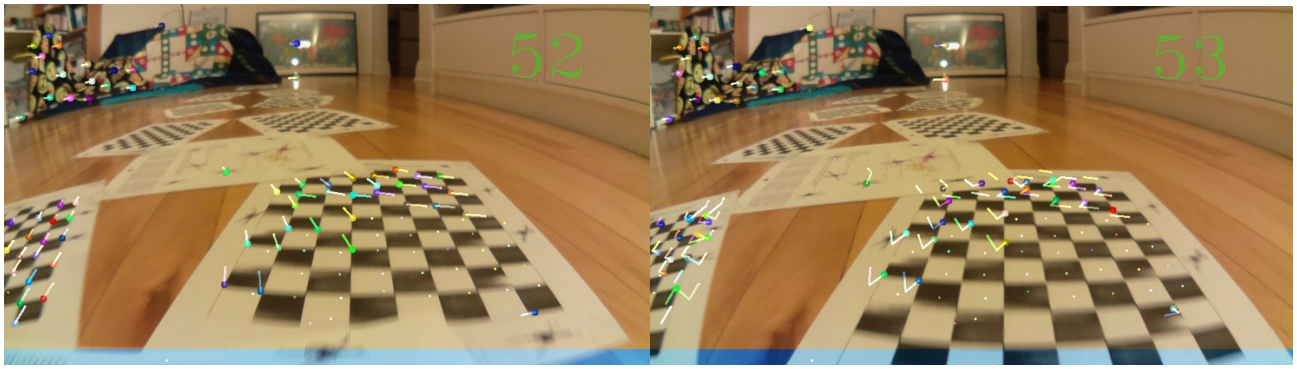


Figure : L'échec de la poursuite

Pour améliorer les performances de la poursuite en temps réel, j'ai effectué quelques réglages:

1) Réduire la vitesse du robot;
2) Ne suivre pas les points trop proches devant la caméra, car c'est trop flou et la disparité est grande;
3) Disperser les points de la poursuite;
4) Déterminer la distance de l'euclidien entre deux points de match, abandonner au-delà d'un seuil;
5) Ajuster les paramètres de la fonction de poursuite du OpenCV.

D'ailleurs, je pense que le motif du damier est trop régulier, ce qui augmentera la probabilité d'erreurs de la poursuite, car les schémas sont trop similaires, notamment lorsque la distance de déplacement entre deux images provoque le chevauchement de grilles de damier.

Les paramètres du OpenCV doivent être ajustés de manière appropriée. Après avoir beaucoup d'essayage sur notre robot, je choisis les paramètres suivants pour la poursuite en temps réel:

Pour le fonction *goodFeaturesToTrack* :

```
maxCorners = 100,
qualityLevel = 0.1, # la proportion de choix de points
minDistance = 5,  # afin de rendre les points assez dispersés, ne doit pas être trop petite
blockSize = 5 # le taille du point
```

Pour le fonction *calcOpticalFlowPyrLK* :

```
winSize  = (15,15), # la fenetre taille pour faire match le point, entre [7,15] est bon.
maxLevel = 2,
criteria = (cv.TERM_CRITERIA_EPS | cv.TERM_CRITERIA_COUNT, 10, 0.03)
```

Notez qu'on doit sauvegarder tous les points de chaque image et marquer **1/0** dans un vecteur d'état pour chaque point de chaque image s'il est encore en bon état ou non (par exemple un point est dehors de l'image présente, l'état = **0**) pendant une période de la poursuite. De plus, il faut assurer que le nombre de points ne peut pas être inférieurs à **8**, c'est pour calculer la matrice d'essentielle et décomposer ***R, t*** (Algorithme des **8** points).

[1] La vitesse du robot est défini par *Adafruit-Motor-HAT-Python-Library*, c'est un nombre entier entre [0,255].

Pour un robot autonome, l'une des aptitudes de base est de continuer à rouler droit, mais en raison des différences variés, le robot ne peut pas se déplacer complètement en ligne droite et nous pouvons corriger cet écart en jugeant son angle de rotation.

Pour notre expérimentation, deux moteurs contrôlent sa propre roue de chaque côté. L'*Adafruit-Motor-HAT-Python-Library* fournit deux paramètres de la correction (*right_trim* et *left_trim*) pour les vitesses de moteur gauche et droite, donc j'ai volontairement réglé le paramètre du moteur droit un peu plus (*right_trim*=**20**), de sorte que lorsque le robot commence à conduire, le grand angle tourne à gauche. Notre objectif est d'ajuster en temps réel ces deux paramètres de moteurs toutes quelques images(frames) afin d'atteindre l'objectif que le robot se déplace finalement en ligne droite.

Le déplacement du robot n'est pas assez stable en raison de différences variés comme le plancher lisse, la condition différence entre les deux côtés du robot, et surtout la roue universelle très instable, donc il ne faut pas ajuster le robot après chaque image. De plus pour mesurer les points, il y a une contradiction de la triangulation (va expliquer davantage dans la section de mesurer la profondeur), donc je choisis d'ajuster le robot une fois après chaque poursuite de **10** images. D'ailleurs, étant donné que la capture de la caméra a un délai (environ **6** images) et que la poursuite ne sert à rien quand le robot calcule ***R, t*** et ajuste la direction.

Donc la logique principale est comme : poursuivre **10** images; pauser le robot; rectifier les points de première image et de dernière image; calculer ***E*** et ***R*** et ajuster le robot; attendre que la caméra lise le délai de **6** images; puis démarrer le robot pour qu'il entre le cycle suivant. Le figure au-dessous est l'organigramme de tâche « Corriger le trajet du robot en ligne droite en temps réel »:

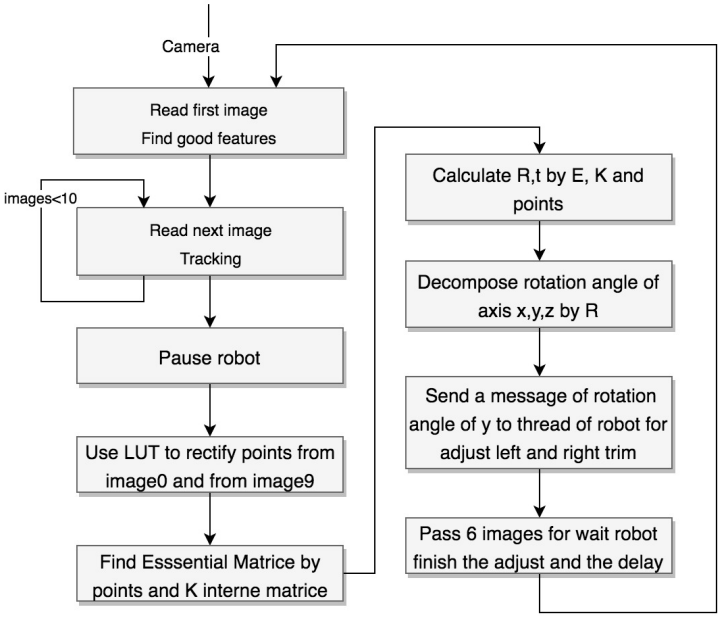


Figure : L'organigramme de corriger le robot en ligne droite

Calculer Essential Matrice et Rotation Matrice:

Je utilise *findEssentialMat(points1,points2,K)* et *recoverPose(E,points1,points2,K)* du *OpenCV* pour calculer ***E***, ***R*** et ***t*** , puis je écris la fonction *rotationMatrixToEulerAngles* pour décompose ***R*** aux angles de rotation de axis ***x, y, z*** .

Relation entre la rotation de *y* et l'ajustement de la vitesse:

Si $\mathbf{angle(y)} > \mathbf{0}$, le robot tourne à gauche, on doit donc augmenter *left_trim* et réduire *right_trim*. Selon le test, je conçois les formules de conversion suivante:

$$\Delta v(left) = 100 * \Delta radian(y)$$
$$\Delta v(right) = -100 * \Delta radian(y)$$
$$left\_trim = clip(left\_trim + \Delta v(left), 0, 20)$$
$$right\_trim = clip(right\_trim + \Delta v(right), 0, 20)$$

La dernière formule permet d'assurer que les valeurs de ces **2** paramètres (*right_trim* et *left_trim*) sont comprises entre **0** et **20**. Si $\mathbf{angle(y)} < \mathbf{0}$, vice versa.

Le résultat du test est très bon, les valeurs du *left_trim* et *right_trim* sont ajustée de **0** et **20** à **12** et **12** pendant quelques poursuites, après cela, le écart de gauche à droite est toujours compris entre **[3, -2]**. Le robot tourne à gauche du grand angle du début jusqu'à la dernière ligne droite.

IV. CORRIGER LA DIRECTION DU ROBOT EN TEMPS RÉEL

Cette tâche est en fait similaire à la précédente, mais je veux seulement essayer de garder le robot dans la même direction après chaque poursuite, c'est aussi très significatif. Pour faire ce test, on n'ajuste pas les paramètres de *trim*, on ajuste directement la direction du robot après chaque poursuite, mais l'organigramme principale est la même.

Pour la relation de la rotation d'axis $\boldsymbol{y}$ et la tourne du robot , je conçois une fonction d'affine :

$$turn\_seconds = sign\big(\Delta radian(y)\big) * \min\left(\text{max\_value}, \text{c} * \text{abs}(\Delta radian(y))\right)$$

Selon mes tests, quand le $\mathbf{speed = 90}$, alors $\mathbf{max\_value = 0.2}$ et $\mathbf{c = 4.4}$. Si $\mathbf{turn\_seconds > 0}$, l'ajustement va tourner le robot à droite et vice versa.

Le résultat du test est aussi très bon, le robot se conduit comme une séquence de l'onde, mais la direction générale est correcte.

V. MESURER LA PROFONDEUR

La contradiction de la triangulation :

La mesure de la profondeur sur certains objets est aussi importante au sujet de SLAM, on va essayer de mesurer les points caractéristiques pour simplifier cette tâche.

Après avoir $\boldsymbol{K}$, $\boldsymbol{R}$ et $\boldsymbol{t}$, avec **2** pairs de points homogènes, selon la géométrie épipolaire, on peut utiliser la mesure de triangulation pour obtenir les 3d coordonnées de points et aussi leur profondeur d_1 et d_2:

$$d_1\widehat{\boldsymbol{p}_1} = d_2\boldsymbol{R}\widehat{\boldsymbol{p}_2} + \boldsymbol{t}$$
$$d_1\widehat{\boldsymbol{p}_1} \times \widehat{\boldsymbol{p}_2} = 0 = d_2\widehat{\boldsymbol{p}_1} \times \boldsymbol{R}\widehat{\boldsymbol{p}_2} + \widehat{\boldsymbol{p}_1} \times \boldsymbol{t}$$
$$\widehat{\boldsymbol{p}} = \boldsymbol{K}^{-1}\,\widetilde{\boldsymbol{p}}$$

OpenCV nous a offert la fonction de *triangulatePoints(externe_matrice1(**R**, **t**), externe_matrice2(0,0), hat_points1,hat_points2)* pour calculer le point de coordonnée 3d, donc je l'ajoute après la décomposition de $\boldsymbol{R}$, $\boldsymbol{t}$.

Encore, il ne faut pas calculer $\boldsymbol{R}$, $\boldsymbol{t}$ pour chaque image, car premièrement, la quantité de calcul est trop grande, Cela affectera la tâche précédente. Ensuite, comme le montre la figure ci-dessous, lorsque $\boldsymbol{t}$ est grand (à droite de le figure), $\Delta\boldsymbol{d}$ (causé par bruit de pixel, l'instabilité de mécanisme du robot, la déviation de l'algorithme pour match les points, etc.) seront considérablement réduites, ce qui signifie que lorsque la translation est plus grande, la mesure de la triangulation sera plus précise à la même résolution de la caméra. Cependant nous ne pouvons pas faire $\boldsymbol{t}$ trop grand, car cela conduira à l'échec du match des points caractéristiques. Ce qui est la contradiction de la triangulation.

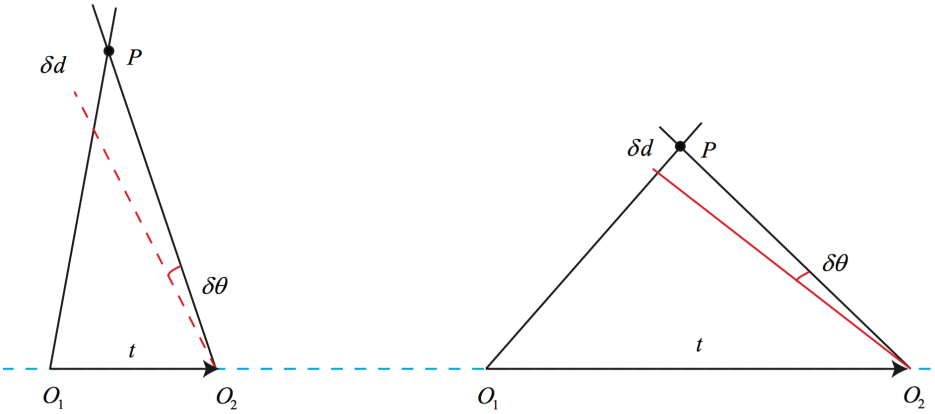
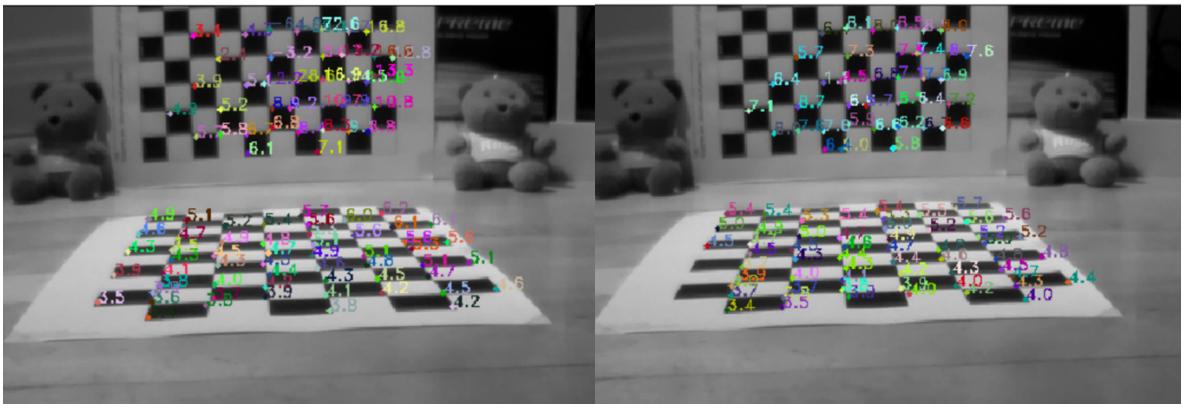


Figure : La contradiction de la triangulation

Notez qu'il y a des versions différentes pour solver équations $\min||Ax||$ dans algorithme des 8 points pour trouver $\boldsymbol{E}$, il affectera le calcul de la profondeur. J'ai essayé *RANSAC* et *LMEDS*. Il faut définir plusieurs paramètres pour les deux comme *confidence* etc. , l'avantage de *LMEDS* est qu'il n'est pas nécessaire de spécifier le seuil d'erreur *ransacThreshold* (permettre de distinguer entre *inlier* et *outliner*), car *LMEDS* peut obtenir la solution optimale de manière itérative.

Je les ai testé plusieurs fois en utilisant seulement les valeurs de paramètres par défaut (*ransacThreshold=**1**,confidence=**0.99***). Nous pouvons comparer les différences de profondeur entre les points dans les deux images ci-dessous, à mon avis *RANSAC* est un peu meilleur pour ce projet, mais ce n'est pas absolu.



LMEDS *RANSAC*

Figure : Comparaison entre *LMEDS* et *RANSAC*

La ligne de base (Baseline) :

Pour un système de caméra monoculaire, il manque le baseline dans un système de stéréo, donc on sait seulement la forme du triangle précédent, mais on ne sait pas l'échelle, s'il est grand ou petit. Donc les profondeurs des photos ci-dessus n'est pas réelle, nous ne pouvons que connaître la différence entre ces chiffres.

Pour obtenir le baseline, j'ai tendance à calculer en fonction de la vitesse du robot et de le **fps** de la caméra, au lieu d'utiliser des informations spécifiques dans l'environnement (telles que la reconnaissance de formes, estimée par la taille d'objets connus comme des voitures ou des personnes; ou par le damier placé), car la manière la plus large est que le robot exécute la tâche **SLAM** dans un environnement inconnu.

Afin de simplifier mesurer le baseline, j'utilise la même vitesse que la tâche précédente, puis je filme une vidéo sur la règle pendant que le robot roule, ensuite j'obtiens la distance entre les deux images, qui devient le baseline le plus élémentaire.

Première image(frame) de la vidéo : ***21cm, 17f, 3.67s*** Dernière image(frame) de la vidéo :***121cm, 95f, 23.71s***

Figure : Vidéo pour mesurer le baseline

$$baseline = \frac{121 - 21}{95 - 17} = 1{,}2820512821\ cm/frame$$

Selon le baseline obtenu, on peut calculer l'autre baseline entre 10 images dans les tâches précédentes, puis après avoir ***R***, ***t***, on corriger simplement le ***t*** matrice, enfin avec le ***t*** corrigé, on obtient la profondeur selon l'algorithme triangulaire. Notez que le baseline exacte de **10** images doit être obtenue en calculant la longueur de la corde selon la rotation ***R***, mais puisque ***R*** n'est pas très grand pour ***10*** images, je la multiplie simplement par **10** pour la raison de simplicité.

L'image suivante montre les profondeurs calculées à l'aide du baseline dans la tâche **2** en temps réel. On constate que l'échelle correspond à la taille réelle. Il existe de nombreux points que leur profondeur sont assez précis, notamment ceux du livre. Mais il existe aussi des profondeurs bizarres, cela est généralement dû à des erreurs dans la poursuite de ces points.

Figure : Les profondeurs corrigés

VI. Conclusion

Globalement, plusieurs objectifs du SLAM ont été bien atteints dans mes expérimentations. Il est important de souligner que la poursuite est une étape très critique et n'a pas toujours des résultats de grande qualité, et que sa qualité affecte directement d'autres tâches suivantes.

De plus, ce projet m'a permis de connaître le SLAM et d'appliquer beaucoup de connaissances de vision 3d. Le SLAM est une direction de recherche passionnante dans le domaine de vision 3d et son application est vaste.

Annexe

IFT6145-projet-report-ZhibinLu.pdf :	Le rapport de ce projet;
code/calibration.py :	Le calibrage de la caméra et le *LUT* fonction;
code/vision3d.py :	Les fonctions fondamentaux de vision 3D;
code/robot_line.py :	Toutes les tâches à propos de ce projet;
output_keep_line.avi/.log/dir :	Le résultat de tâche « Corriger le trajet du robot en ligne droite en temps réel »;
output_correct_direction.avi/.log/dir :	Le résultat de tâche « Corriger la direction du robot en temps réel »;
Robot.py :	Fonctions vient de *Adafruit-Motor-HAT-Python-Library* pour contrôler le robot.

RÉFÉRENCES

[1] Wikipedia, " Triangulation (computer vision)"
[2] Richard Hartley and Andrew Zisserman (2003). Multiple View Geometry in computer vision. Cambridge University Press. ISBN 978-0-521-54051-3.
[3] Doxygen, "Epipolar Geometry", https://docs.opencv.org/3.4/da/de9/tutorial_py_epipolar_geometry.html, Jan 2019
[4] Doxygen, " Optical Flow", https://docs.opencv.org/3.4/d7/d8b/tutorial_py_lucas_kanade.html, Jan 2019
[5] " Camera Calibration and 3D Reconstruction", https://docs.opencv.org/3.0-beta/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html#int%20recoverPose(InputArray%20E,%20InputArray%20points1,%20InputArray%20points2,%20OutputArray%20R,%20OutputArray%20t,%20double%20focal%20,%20Point2d%20pp%20,%20InputOutputArray%20mask)
[6] " Motion Analysis and Object Tracking", https://docs.opencv.org/2.4/modules/video/doc/motion_analysis_and_object_tracking.html?highlight=calcopticalflowpyrlk#calcopticalflowpyrlk
[7] Adafruit.com, "Simple Raspberry pi robot", https://learn.adafruit.com/simple-raspberry-pi-robot/software