Motion Mapping in an Unknown World

EECS 467: Final Report

Atishay Singh
Michael Rakowiecki
Peter Wrobel
Vivian Nguyen

***Abstract*— One of the goals of robotics is to have real-world human task support. To accomplish this support, robots must be able to understand dynamic environments. The human world is full of motion, including walking people, moving pets, and moving chairs. In our work, a small mobile robot, the MBot Mini, used RGB-Depth Simultaneous Localization and Mapping (RGB-D SLAM) from the open-source ORB-SLAM3 library to create a point-cloud representing its environment. The Random Sample Consensus (RANSAC) algorithm then segmented walls and the ground from our point-cloud. We modified the ORB-SLAM3 library to allow for the identification and tracking of features that are in motion relative to the environment. Once we had processed the point cloud, the object velocity and location would be transmitted to the MBot, which would use a PD controller to adjust its velocity to follow the moving object. Due to the computational complexity of the full system, components were required to be run separately from each other. This project serves as a strong proof-of-concept for point-cloud based dynamic object tracking. Following work will focus on optimization to allow real-time use.**

Introduction

The team's goal was to build on top of the existing open-source ORB-SLAM3 library. ORB-SLAM3 refers to an algorithm that solves the Simultaneous Localization and Mapping (SLAM) problem by using ORB features, unique image features that are detectable regardless of rotation or distance. ORB-SLAM3 also supports using depth images to assign distances to detected ORB features, a feature we utilized throughout this project. The ORB-SLAM3 algorithm produces a point-cloud, or a 3D map of ORB features. Typically, points detected in ORB-SLAM3 are discarded if their position changes significantly relative to other points in the frame. However, we have modified the library to retain these points, allowing us to detect and track dynamic objects in the robot's field of view. In addition, we developed a point-cloud segmentation algorithm using the Efficient RANSAC Point Cloud Shape Detector [5] to detect static features in the MBot Mini's environment. These features include the ground plane and wall planes.

Figure 1 gives a visual representation of how data are transferred between a host computer, which we and robot. There are two computers on the robot itself, a Raspberry Pi 4 and a BeagleBone Blue (BBB), along with an additional Intel Realsense camera that we used for gathering image data. The camera sends the RGB and Depth images to the Raspberry Pi, which subsequently is sent to the Host Computer for ORB-SLAM3 point-cloud generation. The Host Computer uses this image data for plane segmentation and to determine a dynamic object. The dynamic object information is sent to the BeagleBone Blue board, which interprets this data into motor commands and sends these commands to the motors on the robot.

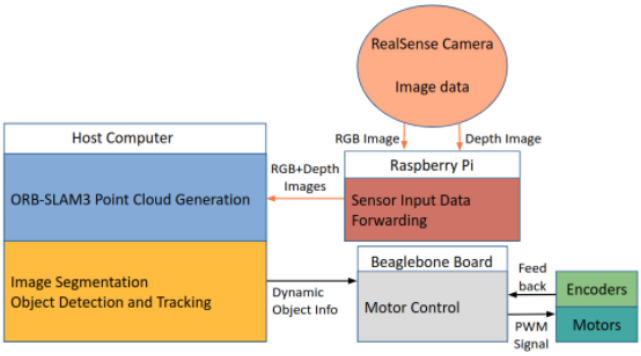

Fig. 1: This flow chart explains how data are transferred between host machine and robot.

RGB-D SLAM Mode

In the pursuit of generating a more dense and accurate point-cloud, we decided to use ORB-SLAM3's RGB-D mode. To use this mode, there needs to be an RGB (or grayscale) image sent to the SLAM algorithm as well as a corresponding depth map for that same frame. RGB-D SLAM provides a superior point-cloud when compared to monocular SLAM because the associated depth information gives a distance to each pixel, increasing the amount of information the algorithm has to work with.

A. Approach

To run RGB-D mode, a camera capable of producing both RGB and depth images is required. To remain within the project's financial constraints, an Intel RealSense D435 camera was purchased. A large majority of RGB-D cameras are not designed to interface easily with Raspberry Pis. This oversight led to a period of difficulty when trying to have the camera operate properly on the MBot. We learned that the sub-optimal USB 2.0 port was required to run our camera. Typically, a USB 3.0 port is preferred because it

allows for higher transfer speeds.

In the current beta v0.3 version of the library, RGB-D support was recently added in September 2020. The feature does not appear to be completely tested. When we attempted to run the example code to try the mode, we were receiving segmentation faults. However, our group was successful in writing a real-time, LCM-based program for RGB-D mode.

When using the D435 camera to stream both color and depth images, the images need to be as synchronized as possible to give the SLAM algorithm depth values that actually correspond to the images being processed. If the streams are too out-of-sync, then the generated point-cloud may be nonsensical clusters of points, or the point-cloud may be not generated at all. From testing, the camera streams were found to be synchronized well at a frame rate of 6 frames per second (FPS). Using the next highest setting (15 FPS), the depth images would remain the same for multiple frames. This is believed to be an issue related to using the Raspberry Pi.

Mobile Object Detection and Description

The primary goal in this project was to detect moving objects and calculate their motion vector. Since the detection was to be integrated into the ORB-SLAM3 library, the objective was to accomplish this using image ORB features and point-cloud information created by ORB-SLAM3.

A. Approach

To identify which features in a stream of images belonged to moving objects, we worked off of an algorithm described by [7]. The paper utilizes a method of finding inconsistencies in Map Points (ORB features points assigned a 3D coordinate) between two frames that would be caused by motion of an object. The algorithm works by creating a fundamental matrix using 8 random points, assuming the random points are stable (this assumption relies on statistically significantly more feature points that are static vs dynamic). The created fundamental matrix F is then applied to all feature points m like so:

$$d_i^j = (m_j^{`T} \cdot F \cdot m_j)^2$$

The points with the highest distance d are eliminated, along with the clusters they make up, when creating the static background. In our case, these points would not be eliminated, but rather used to identify the dynamic objects we are seeking.

The ORB-SLAM3 library has a similar filtering process. applying two filters in the algorithm's tracking thread. The first filter, the rotation filter, projects Map Points that were visible on the previous frame onto the next frame. The Map Points that did not fit into a certain bounding box, or have a different rotation than the rest of the Map Points, are filtered out from the current frame.

The second filter occurs during camera pose estimation. Since the Map Points has established 3D positions, a camera pose is optimized where the difference with the received and calculated depth of the Map Points on the current frame are minimized (if in Stereo/RGB-D mode, as in our case). The points where this difference was too high are cast as outliers, and subsequently removed. The two filtering processes are shown in relation to each other in Algorithm 1 - $SearchByProjection$ filters out the rotational inconsistencies, while $PoseOptimization$ filters out the position outliers.

if $state == OK$ **then**
  $SearchByProjection$: Project Map Points in $Fprev$ onto $Fcur$ (previous, current frames)
  if Less than 20 matches **then**
    $state \leftarrow LOST$
    $return$
  else
    $PoseOptimization$: Calculate new camera pose
    Discard outliers found by $PoseOptimization$
  end if
end if

Algorithm 1: ORB-SLAM 3 Tracking Algorithm. $state$ refers to whether the camera pose has been localized.

B. Algorithm

The ORB-SLAM3 algorithm provides us with two sets of outliers, which will be referred to as Rotation Outliers and Position Outliers. While an outlier was originally erased, it is now used to initialize a potential point on a moving object, referred to as a candidate Mobile Point. The Mobile Point's 3D position is calculated using its depth, position on image, and the camera pose.

In the next frame received, the $SearchByProjection$ algorithm is applied to find matches for potential Mobile Points. If a match is found, the match's 3D position is calculated after the camera pose is established; now the Mobile Point contains two points. Reiterating this process over and over can give a history of a candidate Mobile Point's positions.

Candidate Mobile Points are 'interviewed' to become valid Mobile Points once they are old enough. The criteria they must meet are:

1) Seen in at least last 6 of 8 frames
2) Has a non-zero velocity (filters stationary candidates)
3) Has a consistent velocity vector (filters cases of aliasing)

Afterwards, valid Mobile Points of similar velocity and position can be grouped into a 'Mobile Object', where the object's velocity and position is simply the aggregate of

the contained Mobile Points' velocities and positions. A visualization can be seen in Fig. 2.

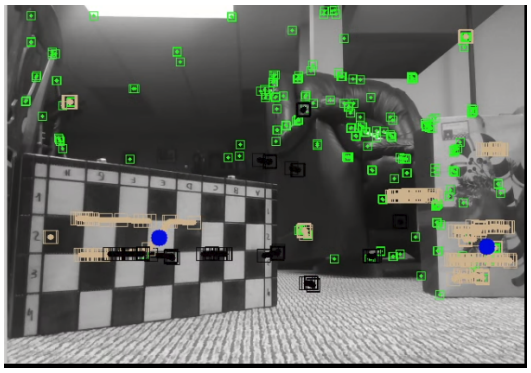

Fig. 2: Visualization of Algorithm. In the camera, green dots represent ORB-SLAM3's Map Points, while tan and black trails of points represent a valid Mobile Point's history. A blue dot represents a Mobile Object, in our case a chess board and present box moving right

C. Results

Using the algorithm described above, we successfully detected moving objects and provided information on their location and velocities under certain conditions.

1) Sensitivity

Moving objects were identified and marked quite well, given they had a significant amount of features. To quantify the extent of success, a test was performed where the supplemented ORB-SLAM3 algorithm would detect and mark a single mobile object going 1 cm/s perpendicularly to the robot's motion while the robot drove at a rate of 2 cm/s. The speeds chosen ensured the robot sent over clear images. The robot then continued to move around the room for another minute or so.

This test was applied 10 times under different conditions. One varying condition was initializing candidate Mobile Points with both Rotation and Position Outliers vs Position Outliers alone, while the other condition was the threshold distance between ORB point descriptors that qualified the pair as a match. A low threshold meant that features were only matched if they had very similar descriptions, while a high threshold allowed for more lax matching.

The three measurements of performances were recall, precision (any), and precision (top). Recall refers to the ratio of times the proper object was identified. The precision (any) count refers to the ratio of successful trials where no false positives were detected, while the precision (top) measurement refers to the ratio of successful trials where the moving object contained the most Mobile Points when compared to false positives. The results are found in Table I.

Outliers, Threshold	Recall	Precision (any)	Precision (top)
Rot. + Pos., High	1.00	0.50	0.80
Rot. + Pos., Low	0.90	0.90	0.90
Opt., High	0.60	0.50	0.60
Opt., Low	0.30	0.30	0.30

TABLE I: Recall and Precision on detecting a single moving object.

The test showed that conditions where both Rotation and Position Outliers were used under a strict matching threshold were best suited for the environment. The high level features allowed for high sensitivity and a consideration of rotating moving objects, while the low matching distance threshold prevented aliasing in matching and creating Mobile Objects from depth measurement noise.

2) Mobile Object Speed

The algorithm had a difficult time identifying objects moving at a reasonable speed relative to the camera. The extent of this was tested by having an object move at a different speeds at two different distances from a stationary robot. The results are reported in Table II, where the number of times the object was detected in 10 trials is reported as a percentage.

Object Speed	1m Away	3m Away
1 cm/s	100%	90%
2 cm/s	90%	90%
5 cm/s	10%	50%
10 cm/s	0%	20%

TABLE II: Detection vs Object Speed at different distances.

The results show a sharp cut off at 5 cm/s when the object is 1 meter away. In the trial where the detection was successful, the tracking was lost within a second. There are two likely factors behind the sharp decrease in performance. The first is that the ORB features move too far per frame to be detected by the bounding box around outlier's position in previous frame. Another potential factor is image blurring, caused by the camera's slow frame rate. This blur results in ORB features being distorted or undetectable, causing candidate Mobile Points to fail to find a match between frames.

When the object is a farther distance from the camera, the potential issues mentioned above are subdued. However, there are diminishing returns — the farther the object is from the camera, the lower resolution ORB features are detected and are therefore more likely to be unmatched in subsequent frames. The depth information on the ORB points detected also become increasingly noisier the farther they are from the camera, resulting in depth hopping between frames, causing the candidate Mobile Point describing the point to have an inconsistent velocity.

3) Limits

It is imperative to state that the way the algorithm was applied was computationally expensive, and resulted in the ORB-SLAM3 algorithm to run quite slow. In conditions where three or more objects were detected, with many Mobile Points needing updates, the resulting lag of sometimes a full second caused the program to be significantly out-of-sync with the stream of real time images received. Theoretical solutions exist where the algorithm could be optimized to reduce performance, such as threading, a limit on objects, not testing every frame, or only initializing Mobile Points in a section of the image with optical flow.

The algorithm also falls short in tracking objects moving at a relative speed greater than a few centimeters a second, as described previously. Therefore, tracking objects in real time is limited, as a low frame rate combined with quick objects result in blurry movement and unidentifiable ORB features.

Finally, the algorithm, as well as all of ORB-SLAM3, is contingent on the idea that there are significantly less detectable ORB features in moving objects than the static surroundings. When this doesn't hold, the camera pose estimate can potentially be estimated using the points in the object rather than the surroundings. The result can be seen in Fig. 3.

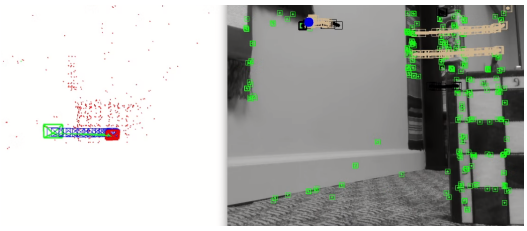

Fig. 3: Case where moving chess board is treated as static by the ORB-SLAM3 algorithm, resulting in mapping stationary robot as moving, and identifying a static scarf (top left) as a Mobile Object.

Interacting with Moving Objects

After gaining the ability to detect a moving object in the environment, the next natural step is to try to interact with it in some way. Our object detection and motion tracking code is designed to communicate with our BBB via LCM. Information regarding the dynamic object's velocity, angle, and location relative to the MBot is relayed in the message. The rest of the described code in this section is run on the BeagleBone to save processing power for the Raspberry Pi and host computer.

A. Operating Modes

There are two overarching states associated with the robot's motion. The first is 'Explore', where the robot is simply moving around in its environment to build up its map and learn more about its environment. The second state is 'Follow', where the robot attempts to face, move toward, and follow a dynamic object that it recognizes. Within these two states, there are smaller states that dictate how the robot moves when exploring versus following. Algorithm 2 contains a description of the robot's motion state machine.

B. Explore Mode

Upon initialization, the robot remains stationary. This is to ensure safety and predictable behavior. When an LCM message is received from the host laptop, there is a boolean variable that states whether a dynamic object exists. When no dynamic object is found, the MBot enables Explore mode. Simply put, the robot drives forward with open-loop PWM control, until the LCM messages raise a flag stating an approaching obstacle. The robot will then turn to the right until the LCM messages lower the obstacle flag.

The algorithm is simple, and allows for quick testing. In the future, it may be replaced with something more sophisticated, such as A* path planning during the 'Explore' mode.

C. Follow Mode

If the LCM messages received by the BeagleBone say that a dynamic object has been spotted, then 'Follow' mode is engaged. In this mode, the algorithm makes use of additional information provided by the LCM message. The additional information is the (x,y) location, angle, and velocity of the object relative to the MBot. From this given information, angular displacement and distance setpoints are calculated. Two closed-loop controllers then run to minimize the MBot's error with respect to the setpoints. Encoder data are used to update the MBots's position estimate as the controllers run. This continues until the setpoints are reached or a new LCM message is received.

If a new LCM message is received, the setpoints are recalculated and the MBot's encoder counts are re-zeroed. This ensures that the controller gets to work with information that is aligned with the SLAM map each time a new LCM message is received. This is done to mitigate drift when following an object.

Point-Cloud Segmentation

While the object detection and motion tracking algorithm provides a solid basis for the control laws written for this project, relying on the motion tracker alone could result in severe issues during the operation of the MBot Mini. For one, while the robot is moving, it could mistake static obstacles or walls as moving objects, which would then result in the MBot Mini following the wrong object. In addition, failing to take these static obstacles and structures into account could result in not recognizing an obstacle or incorrectly predict the velocity of a moving object upon collision with an obstacle,

```
Require: LCM Input from ORB-SLAM3 Side
  if State == Explore then
    if Clear Ahead of Robot then
      Substate = Forward
    else
      while Area Ahead Blocked do
        Substate = Turn Right
      end while
    end if
  end if
  if State == Follow then
    Use motion vectors of
    robot and object to
    calculate lateral and
    longitudinal setpoints

    Run closed-loop control
    on setpoints until error
    is less than a threshold ϵ
  end if
```

Algorithm 2: Robot Motion Decisions

```
Require: mappoints - Matrix of generated Map Points
Require: planes - Pointer to list of detected planes

  Pointcloud pc
  for row in mappoints do
    Point point = Point(row[0], row[1], row[2])
    Add point to pc
  end for
  n = 0
  while n ¡ maximum number of iterations do
    Create random subsets of points in pc
    for subset in pc do
      Find plane of best fit for subset using gradient descent

      Compute number of points that fit on plane
      if subset fits well to a plane then
        Add computed plane to planes
        Shift fitted points to end of pc
      end if
    end for
    n++
  end while
  return Number of unfitted points
```

Algorithm 3: RANSAC and Plane Detection Algorithm

both issues that could result in the robot crashing into an obstacle or moving object. To account for these potential issues, we decided to implement a point-cloud segmentation algorithm to identify and label ground and wall planes in the point-cloud map and in the image frame.

A. Approach

At a high level, our approach involves extracting the Map Points generated by ORB-SLAM3 and running a RANSAC algorithm over these points to identify and label clusters of points that resemble planes. The generated planes are then labeled as static plane objects, representing either the ground plane or a wall in the environment. The RANSAC algorithm then separates the planar points from the non-planar points, allowing the object detection algorithm to process the points not associated with a static obstacle. We used an external library, as described in the "Efficient RANSAC for Point-Cloud Shape Detection" paper [5], to handle the RANSAC algorithm and shape detection, modifying this library to communicate with the ORB-SLAM3 algorithm using LCM.

Every time the point-cloud map is updated, a list of active Map Points are extracted through the ORB-SLAM3 algorithm. This list of Map Points is converted to an array of float values containing the coordinates of each Map Point in the frame. This array, as well as the number of extracted Map Points, is wrapped in an LCM message and sent to the RANSAC algorithm for processing. The RANSAC algorithm then parses the matrix, populates an internal point-cloud object, and attempts to fit the points to one or more planes, as described in Algorithm 3.

Once the points are processed through the algorithm above, the point-cloud will have been arranged in such a way that the outlier points are isolated from the other points within the point-cloud object. Using this information, as well as the number of unfitted points returned by the algorithm, we extract the fitted points from the point-cloud object, saving the coordinates of each point to an array of float values. This array and the number of fitted points are wrapped in an LCM message and sent to the ORB-SLAM3 algorithm, where they are converted to point objects and drawn in both the point-cloud map and the image frame.

While it was possible to detect non-planar shapes with the RANSAC library we were using, we found that detection for non-planar objects tended to run slower than detection for planar objects and that the RANSAC algorithm occasionally failed to return any results when attempting to detect non-planar objects. As such, we made the assumption that the ground and walls in our environment would be planar in nature. This assumption held true for the environments in which we tested the object detection algorithm. However, in its current state, the point-cloud segmentation algorithm may not perform well in environments with non-planar floors or walls.

B. Performance

In order to assess the feasibility of the point-cloud segmentation algorithm described above, the algorithm was tested by manually creating a series of points resembling one or the planes for use as inputs. Based on this data, we were to test how the time taken to detect all the planes in a point-cloud

scales with the number of points per plane and the number of planes in the point-cloud. The results of this testing are depicted in the graphs below.

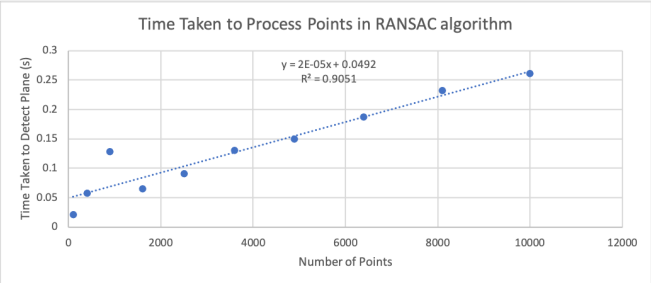

Fig. 4: This graph depicts the roughly linear relation between number of points in a plane and time taken to detect the plane.

As seen in Figure 3 above and Figure 4 below, the time complexity of the RANSAC algorithm is linear with both the number of points per plane and the number of planes per point-cloud. It takes roughly 0.25 seconds to detect a plane containing 10,000 points, which is much larger than the planes we expected to detect during the operation of our robot. As such, we expected the algorithm to run smoothly in real time, although it would be somewhat slow when attempting to identify many large planes. Based on this information, we proceeded to integrate the RANSAC algorithm with ORB-SLAM3 by setting up LCM communication between the two libraries.

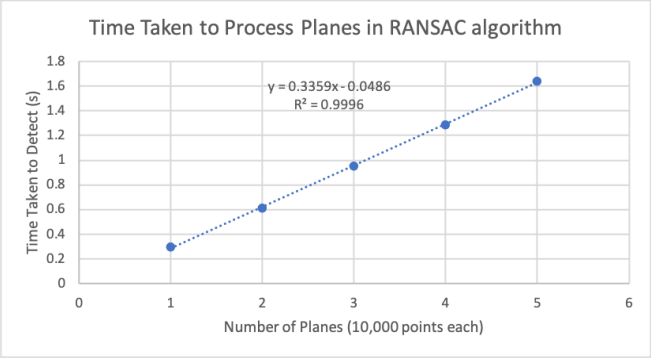

Fig. 5: This graph depicts the linear relation between number of planes and time taken to detect all the planes.

Upon testing the image segmentation algorithm in real time, we found that the RANSAC detector was very slow and that it exhibited very strange behavior in real time. For one, it took several minutes to process a single plane containing roughly 2,000 points, if it did at all, despite the fact that it had managed to parse a similarly sized plane relatively quickly during our initial testing. In addition, since the RANSAC detector processed points incredibly slowly, there would be instances in which both ORB-SLAM3 and the RANSAC detector would throw segmentation errors simultaneously due to the RANSAC detector's inability to handle LCM messages in real time. As such, we were unable to fully integrate the image segmentation algorithm into the final product, despite the fact that both the ORB-SLAM3 algorithm and the RANSAC segmentation algorithm were capable of running independently.

C. Difficulties

The greatest difficulty faced in developing the point-cloud segmentation algorithm was getting the algorithm to operate in real time. As discussed above, we never managed to fully integrate the RANSAC detector into the object detection algorithm due to the RANSAC detector's inability to process points in real time. If there was more time to develop the point-cloud segmentation algorithm, we would likely focus on developing a lighter weight RANSAC algorithm that meets our needs better than the solution described here.

While working on communication between the RANSAC detector and the ORB-SLAM3 algorithm, both programs would experience a segfault error after two or three LCM messages were delivered. Upon further testing, this was largely due to the RANSAC detector receiving multiple LCM messages too quickly for it to process. As such, we added a flag in the ORB-SLAM3 code that indicated whether or not a response from the RANSAC detector was received, and only allowed sending LCM messages if a response to the last message was received. This solved the segmentation fault error, although it did not solve our latency issues.

Next Steps

Although the end of the semester is quickly approaching, the next phase of this project would be to optimize the implementation, so that the object detection and plane segmentation will be able to run at a higher frequency. Currently, the ORB-SLAM3 algorithm, the RANSAC algorithm, and the algorithm for object following have been separate entities and run on different computers. Each of these algorithms have been too computationally expensive to run on the same machine, or at the same time. Since external libraries were used, the only plausible way to resolve this problem would be to find smaller, less expensive libraries, or gain access to more computing power, neither of which are plausible in the scope of this class.

Certification

I participated and contributed to team discussions on each problem, and I attest to the integrity of each solution. Our team met virtually and physically as a group at least two times a week throughout November and December.

Peter Wrobel
Vivian Nguyen
Atishay Singh
Michael Rakowiecki

References

[1] Sudeep Pillai and John J. Leonard *Monocular SLAM Supported Object Recognition* http://www.roboticsproceedings.org/rss11/p34.pdf

[2] Runzhi Wang, Wenhui Wan, Yongkang Wang and Kaichang Di *A New RGB-D SLAM Method with Moving Object Detection for Dynamic Indoor Scenes* https://www.mdpi.com/2072-4292/11/10/1143

[3] Bin Yang, Wenjie Luo, and Raquel Urtasun *PIXOR: Real-time 3D Object Detection from Point Clouds* https://openaccess.thecvf.com/content_cvpr_2018/papers/Yang_PIXOR_Real-Time_3D_CVPR_2018_paper.pdf

[4] Ramy Ashraf Zeineldin and Nawal El-Fishawy *Fast and accurate ground plane detection for the visually impaired from 3D organized point clouds* https://www.researchgate.net/publication/307572427_Fast_and_accurate_ground_plane_detection_for_the_visually_impaired_from_3D_organized_point_clouds

[5] Ruwen Schnabel, Roland Wahl, and Reinhard Klein *Efficient RANSAC for Point-Cloud Shape Detection* http://www.hinkali.com/Education/PointCloud.pdf

[6] *RANSAC Library for Plane Detection* https://github.com/alessandro-gentilini/Efficient-RANSAC-for-Point-Cloud-Shape-Detection

[7] Runzhi Wang, Wenhui Wan, Yongkang Wang, and Kaichang Di *A New RGB-D SLAM Method with Moving Detection for Dynamic Indoor Scenes* https://www.mdpi.com/2072-4292/11/10/1143