A General Algorithm for Object Localization

Dylan Callaway

Mechanical Systems Control Lab, UC Berkeley - Fall 2019

Abstract

The purpose of this work was to develop a general approach to the detection and localization of objects in three-dimensional space; with the ultimate goal being the identification of obstacles in the path of industrial mobile robots. Several methods for object detection were surveyed and a point-cloud-based approach utilizing only a stereo camera was chosen. The algorithm was built around this option as it provided the most generality and ease of implementation.

Introduction

There are numerous cases where the deployment of robots may be beneficial in industrial settings. Carrying heavy or cumbersome loads like in Figure 1, or transporting more simple to move objects without human intervention like in Figure 2.

Figure 1: Waypoint Robotics [9].

Figure 2: Mobile Industrial Robots [6].

Most current implementations use a combination of autonomous and guided control, with the ultimate goal being the removal of human operators to as great an extent as possible. In order to achieve this goal, the ability of the robots to avoid obstacles such as people, infrastructure, other robots, and debris is paramount.

The objective of this paper is to present a survey of several methods that may be used to accomplish this task, outline the preferred one for the application,

and demonstrate the implementation on a comparable robot, like that shown in Figure 3.

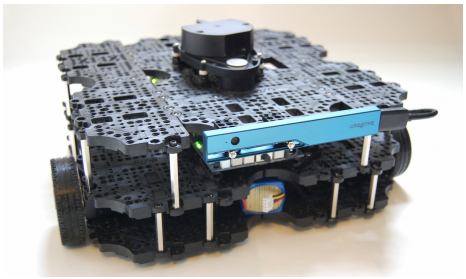

Figure 3: Robotis TurtleBot3 Waffle.

Survey of Methods

The aspects that are most critical to the success of any object detection algorithm within the scope of this project are generality, or the ability of the algorithm to be used in a multitude of settings without additional effort; low computational overhead, as the algorithm will be implemented on a relatively light hardware platform; and extensibility, or the ability of the algorithm to have features added to it, in order for its performance to be improved on with time.

The first category of algorithms surveyed for this purpose are those which use machine learning (ML). Specifically, ML algorithms which use convolutional-neural-networks (CNNs) to segment and classify objects in a 2D image similar to the overview shown in Figure 4.

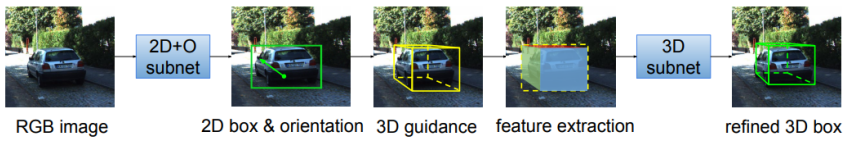

Figure 4: Example of CNN object detection from [1].

The segmented, 2D pixels can then be matched to their 3D point-cloud counterparts and a description of an object in 3D can be generated. This approach has two main drawbacks. First, machine learning algorithms must be trained on a large set of data in order to properly classify anything. This means that there must be prior knowledge of the objects the robot will come into contact with, such that the algorithm may be trained on them beforehand; thus reducing the generality of the algorithm. Second, because of the complexity of CNNs, particularly when intertwined with other processing algorithms, their implementation

in a system which needs to react to the data they provide quickly requires significant computational overhead; beyond that which most single-board-computers (SBCs), like that found on the TurtleBot3, can provide. However, machine learning algorithms are very extensible, and do provide the best performance when the prerequisite knowledge and computational requirements are met. Several examples of methods of this type that were used to complete this survey can be found at [1][10][5].

Another approach that may be taken in order to solve this problem is the use of markers or other identifying features that can be attached to objects in order to detect them like that shown in Figures 5 and 6.

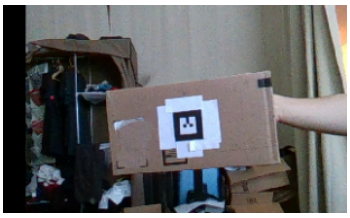

Figure 5: Marker on a box [2].

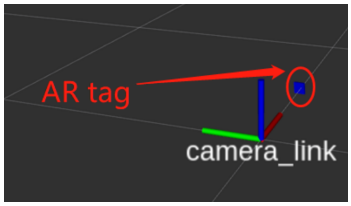

Figure 6: Translation to 3D space [2].

There are three major drawbacks with this approach. First, the markers themselves must be generated and an algorithm developed for their detection. This is not detrimental to this method as there are several open-source packages that do this well. Second, the markers must be physically placed on the objects, which may require significant effort for many objects and thus reduces the generality of this approach. Third, and most importantly, the object position with respect to the markers must be encoded into the general detection algorithm so that when the marker positions are detected, the position of the object can be inferred. These three aspects combined reduce the generality of this approach significantly, as all objects in the field-of-view (FOV) must be marked and their relative positions to said markers encoded. This approach also lacks extensibility, as it relies solely on the marker detection for performance; a task which is already done with great success. However, the computational overhead required to perform such detection is extremely minimal. Most marker detection schemes are very efficient, and to infer the object position from that data only requires matrix addition and a pre-encoded transformation. An example of this method of detection can be found at [2].

The final approach surveyed, and the one which was ultimately selected, is the aggregating of nearby points in a 3D point cloud into distinct clusters. This method can be seen graphically in Figures 7 and 8.

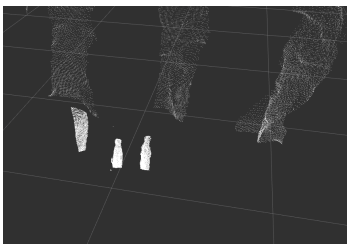

Figure 7: Raw point cloud.

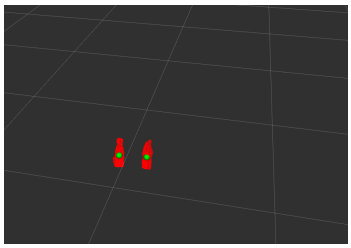

Figure 8: Two sets of aggregated points.

This approach was selected for three primary reasons. First, it is completely general; any reasonably sized object[1] that is in the field of view of the camera, and not occluded, can be detected with no prior knowledge or effort. Second, it utilizes only a stereo camera, which is comprised of a pair of monochrome cameras, meaning the amount of data entering the processing pipeline is small, even compared to typical RGB image processing. Finally, this approach provides many options and paths for extensibility in the form of greater sensor fusion with common sensors like RGB cameras and more complex ones like LIDAR, and the ability to implement machine learning algorithms on the segmented data, which can be used to gather further information about the objects. These reasons, combined with the fact that packages and documentation for the efficient segmentation of 3D point clouds are readily available, make it a clear choice for this application.

Development of the Algorithm

For this project we define coordinate frame, B, attached to the robot and inline with the camera frame such that x is orthogonal to the center of the camera lens, z is in line with Earth's gravity, and y is defined then by the right-hand-rule as shown in Figure 9.

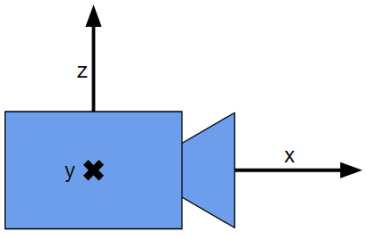

Figure 9: Coordinate frame used in the algorithm.

[1]There are limits to the minimum size of object the algorithm can detect based on the resolution and sensitivity of the camera. Objects of less than about $5cm^3$ of volume cannot be captured with much confidence and are thus ignored.

After each frame is captured, the algorithm will act according to Figure 10.

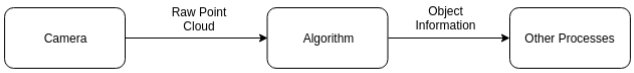

Figure 10: Algorithm outline.

Once the camera sends the raw point cloud data to the algorithm, it is stored in $\boldsymbol{P}$, a $3 \times n$ matrix where n is the number of points in the cloud. Each row represents the x, y, and z coordinates of each column, $\boldsymbol{p}_i$, the i^{th} point such that

$$\boldsymbol{P} = \begin{bmatrix} \boldsymbol{p}_0 & \boldsymbol{p}_1 & \dots & \boldsymbol{p}_i & \boldsymbol{p}_{i+1} & \dots & \boldsymbol{p}_n \end{bmatrix} = \begin{bmatrix} x_0 & x_1 & \dots & x_i & x_{i+1} & \dots & x_n \\ y_0 & y_1 & \dots & y_i & y_{i+1} & \dots & y_n \\ z_0 & z_1 & \dots & z_i & z_{i+1} & \dots & z_n \end{bmatrix}$$

Once $\boldsymbol{P}$ is populated for the current frame, the first step is to remove any points that are sufficiently far enough away from the robot to be safely ignored. Therefore, all points that satisfy one, or more, of the following

$$\boldsymbol{p}_i > \boldsymbol{TOL} \quad or \quad \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix} > \begin{bmatrix} TOL_x \\ TOL_y \\ TOL_z \end{bmatrix}$$

are discarded from $\boldsymbol{P}$ in order to reduce the amount of data that needs to undergo more complex processing. Where TOL_x, TOL_y, and TOL_z are the maximum distance in x, y, and z a point may be from the camera.

Once these points are discarded, $\boldsymbol{P}$ is then searched for clusters of points that satisfy the equation

$$ax + by + cz + d = 0$$

where $a \approx 0$, $b \approx 0$, $c \approx 1$, and $d \in \mathbb{R}$. This is approximately the equation of a plane parallel with the ground in the B coordinate frame. The points that satisfy this equation are typically the ground, and are thus subtracted from $\boldsymbol{P}$. Removing the ground plane in this fashion is necessary so the segmentation process does not falsely identify these clusters of points as obstacles.

With the ground removed, the aggregation of points which will be deemed obstacles begins. The segmentation is done according to the following algorithm from [7] as implemented in the PCL library [8].

Each $\boldsymbol{p}_i$ is organized into a k-dimensional tree where the linked points are the nearest other points to $\boldsymbol{p}_i$. An example structure for p_0 is shown in Figure 11.

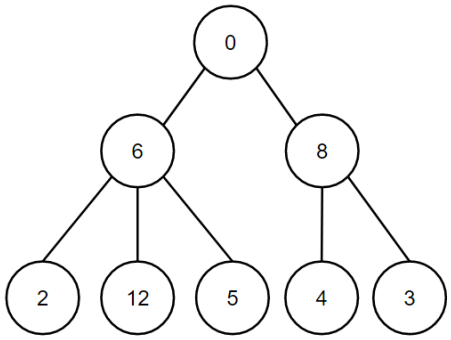

Figure 11: k-dimensional tree example.

Then, each $\boldsymbol{p}_i$ is checked for neighbors within a euclidean radius of r_{th}. All points that are inside of r_{th}, and that have not already been processed, are added to a list of points considered to be part of that cluster, and removed from the list of points that still need to be checked. This process is repeated for all $\boldsymbol{p}_i \in \boldsymbol{P}$. Once all of the points in $\boldsymbol{P}$ have been checked and added to a cluster, this part of the algorithm terminates.

The clusters are then further filtered based on the number of points they contain to remove any noise or unrealistic clusters. Clusters are removed from the list of clusters if they satisfy either

$$N_j^c < TOL_N^- \quad or \quad N_J^c > TOL_N^+$$

where N_j^c is number of points in the j^{th} cluster and TOL_N^- and TOL_N^+ are the lower and upper bounds on the size of the clusters, respectively.

After each cluster, $\boldsymbol{c}_j$, is filtered for size, the remaining group of clusters are stored in vector $\boldsymbol{C}$. The bounding box of each object is then found by taking the minimum and maximum x, y, and z position all of the points in each cluster. The centroid of each object is also computed according to

$$\boldsymbol{g}_i = \frac{\sum_{i=0}^{N_j^c} \boldsymbol{p}_i}{N_j^c} \quad \forall c_j \in \boldsymbol{C}$$

In order to maintain consistency for different processes operating on the robot, the position and bounding box of each object is also represented in an Earth-fixed coordinate frame. The transformation is applied according to

$$[\boldsymbol{s}_{PE}]^E = [\boldsymbol{s}_{PB}]^E + [\boldsymbol{s}_{BE}]^E = [\boldsymbol{T}]^{EC}[\boldsymbol{s}_{PB}]^C + [\boldsymbol{s}_{BE}]^E$$

where $[\boldsymbol{s}_{PE}]^E$ is the displacement of the point with respect to the robot starting location, E, in the earth fixed frame, E. All of the displacements, $[\boldsymbol{s}_{AB}]^C$, can be interpreted as the displacement of point A with respect to point B in the frame C. The transformation matrix from the camera frame to the Earth-fixed frame, $[\boldsymbol{T}]^{EC}$, is calculated from knowledge of the robots translations and

rotations [3]. This information is encoded in the quaternion of rotation between the two frames, $\{q\}^{EC}$, which is defined as follows

$$\{q\}^{EC} = \begin{bmatrix} q_0 \\ [\boldsymbol{q}_v] \end{bmatrix} = \begin{bmatrix} cos(\frac{1}{2}\rho^{EC}) \\ [\boldsymbol{n}^{EC}]^E sin(\frac{1}{2}\rho^{EC}) \end{bmatrix}$$

From $\{q\}^{EC}$ the transformation matrix can be calculated according to

$$[\boldsymbol{T}]^{EC} = e^{-[\boldsymbol{\Lambda}^{EC}]^E} \quad and \quad [\boldsymbol{\lambda}^{EC}]^E = \rho^{EC}\boldsymbol{n}^{EC}$$

where $[\boldsymbol{\Lambda}^{EC}]^E$ is the second order tensor representation of $[\boldsymbol{\lambda}^{EC}]^E$ defined as

$$[\boldsymbol{\Lambda}^{EC}]^E = \begin{bmatrix} 0 & -\lambda_2 & \lambda_1 \\ \lambda_2 & 0 & -\lambda_0 \\ -\lambda_1 & \lambda_0 & 0 \end{bmatrix}$$

With the determination of the position and size of each obstacle in the Earth-fixed coordinate frame, the algorithm is complete for that image cycle, and restarts when the next image of raw point cloud data is received.

Implementation

The algorithm described in the previous section is implemented on a TurtleBot3 Waffle using a combination of C++, ROS, and Python. The Point Cloud Library [8] was selected to perform the point cloud manipulation as it can be easily integrated into ROS and C++, and is well documented and maintained. The camera chosen for this application was the Intel Realsense D435i as it has a wider field of view than the D415i.

A caveat to the transformations mentioned above is that the camera and robot frames do not line up initially, so a second transformation matrix is required such that

$$[\boldsymbol{T}]^{EC} = [\boldsymbol{T}]^{EB}[\boldsymbol{T}]^{BC}$$

In order to improve the speed of the algorithm, the camera only captures depth data at the lowest resolution possible, 480×270. This reduces the amount of data that must be processed by all steps of the algorithm, thus improving speed significantly.

The algorithm is broken down into separate ROS nodes that may be run on separate devices for ease-of-use or efficiency. The relatively simple task of filtering the raw point cloud for distance is done on a TurtleBot3, as it can be completed quickly with limited hardware. The point cloud segmentation and transformations are done in separate nodes, but on the same hardware, an MSI laptop, in this case.

A plotting tool was developed using Python to display the detected objects in the Earth-fixed frame, along with a point representing the robot. The system was tested with several small to medium sized objects as shown in Figures 12 and 13.

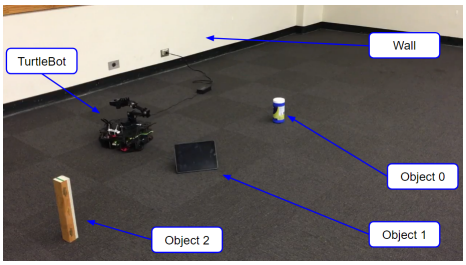

Figure 12: Diagram showing obstacles and TurtleBot.

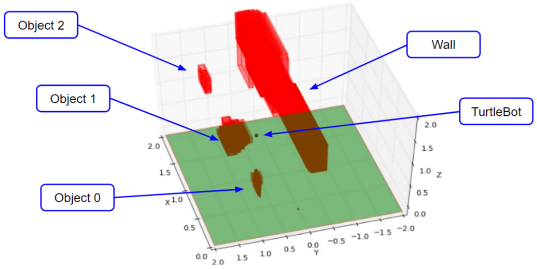

Figure 13: Diagram showing obstacles and TurtleBot in plotter.

To demonstrate typical performance of the algorithm, three images at subsequent time steps are shown in Figures 14 to 17.

Figure 14: $t \approx 1s$.

Figure 15: $t \approx 15s$.

Figure 16: $t \approx 30s$.

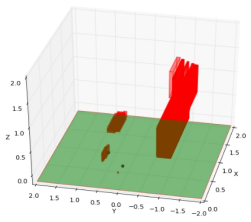

Figure 17: $t \approx 1s$.

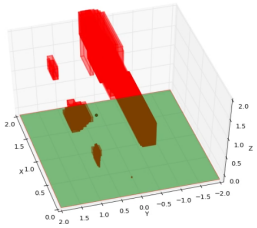

Figure 18: $t \approx 15s$.

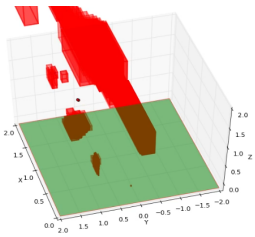

Figure 19: $t \approx 30s$.

Areas for Development

There are two key areas that would yield the most substantial improvements to the performance of the algorithm.

First, as can be seen in Figures 17 to 19, the bounding boxes are not entirely fixed around the objects they represent. There is a significant amount of noise in the raw centroid and bounding box measurements that could be greatly reduced with filtering. A Kalman filter was briefly implemented according to [4] as follows

$$\hat{x}_p(k) = A(k-1)\hat{x}_m(k-1) + u(k-1)$$
$$P_p(k) = A(k-1)P_m(k-1)A^T(k-1) + Q(K-1)$$
$$P_m(k) = (P_p^{-1}(k) + H^T(k)R^{-1}(k)H(k))^{-1}$$
$$\hat{x}_m = \hat{x}_p + P_m(k)H^T(k)R^{-1}(k)(z(k) - H(k)\hat{x}_p(k))$$

where A, the system dynamics model, is defined as

$$A = \begin{bmatrix} 1 & 0 & 0 & \Delta t & 0 & 0 \\ 0 & 1 & 0 & 0 & \Delta t & 0 \\ 0 & 0 & 1 & 0 & 0 & \Delta t \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

which is the forward-Euler numerical integration of the constant velocity of a given object with respect to time.

However, the implementation allowed for filtering of only one object, which significantly reduced the overall capability of the algorithm. It is possible to filter multiple objects, but the software infrastructure to correctly index each object, so as to keep track of their positions and velocities over time, was not able to be designed and built during the course of this project.

Second, the typical frame rate during testing with a similar number of objects as shown in Figures 14 to 19 was about 2FPS. This frame rate is relatively low and could be improved on by increasing the efficiency of the code, something that is not typically done at this stage in development, unless absolutely necessary. The more problematic issue with frame rate comes when objects get very near to the camera, causing $\boldsymbol{P}$ to become populated with the maximum number of points. This causes the algorithm to slow down significantly, to $\frac{1}{6}$FPS in some cases. The simplest way to remedy this issue would be to down-sample the raw data if the point cloud becomes overpopulated and the points are sufficiently close to the camera.

Conclusion

This project sought to develop a general algorithm for obstacle detection in small robots. It was successful in three key ways. First, the algorithm was

fully implemented on the TurtleBot3 and its performance demonstrated with various household objects. Second, the algorithm is very general, in that any physical object may be placed in the field-of-view of the camera and its position and extreme boundaries will be known; no prior knowledge or effort is required. Finally, the algorithm is extensible, with the addition of filtering, the robustness of the algorithm can be improved greatly, and additions like machine learning or fusion with other sensors can increase the knowledge gained about each object greatly. Along with the successes there is also a drawback to the algorithm, in that the computational overhead is not as low as desired. As mentioned previously, the frame rate is relatively low and would likely not be sufficient for industrial applications. The robot would lack the ability to react quickly to fast moving objects, which could be a safety concern. It is possible to improve the speed, but it was deemed out of the scope of this project.

This work could not have been completed without the enormous amount of help and guidance provided by Jessica Leu throughout, or without the opportunity from Professor Masayoshi Tomizuka and the Mechanical Systems Control (MSC) Lab at UC Berkeley.

References

[1] Buyu Li, Wanli Ouyang, Lu Sheng, Xingyu Zeng, Xiaogang Wang. GS3D: An efficient 3D object detection framework for autonomous driving. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1019–1028, 2019.

[2] Jieming Li. Exploring Robust Object Tracking – Visualizations, Noise Filtering and AR Tag Utilization in mobile robot platform. 2019.

[3] Mark Mueller. ME136 Course Notes. UC Berkeley College of Engineering, 2019.

[4] Mark Mueller. ME233 Course Notes. UC Berkeley College of Engineering, 2019.

[5] Charles R. Qi, Or Litany, Kaiming He, and Leonidas J. Guibas. Deep Hough Voting for 3D Object Detection in Point Clouds. *arXiv e-prints*, page arXiv:1904.09664, Apr 2019.

[6] Mobile Industrial Robots. Mobile industrial robots (mir) launches mir finance – a "robot as a service" (raas) leasing program, 2019.

[7] Radu Bogdan Rusu. Semantic 3d object maps for everyday manipulationin human living environments. *KI - Künstliche Intelligenz - German Journal of Artificial Intelligence*, 2009.

[8] The Point Cloud Library (PCL). www.pointclouds.org, 2019.

[9] Jason Walker. Waypoint launches MAV3K, a 3000lb payload omnidirectional, autonomous mobile robot, 2019.

[10] Jesus Zarzar, Silvio Giancola, and Bernard Ghanem. PointRGCN: Graph Convolution Networks for 3D Vehicles Detection Refinement. *arXiv e-prints*, page arXiv:1911.12236, Nov 2019.