Trajectory planning with real time vision-based obstacle detection

by

Xuemeng Li

B.A.Sc., Simon Fraser University, 2017

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

MASTER OF ENGINEERING

in

The Faculty of Applied Science

(Electrical and Computer Engineering)

THE UNIVERSITY OF BRITISH COLUMBIA

(Vancouver)

April 2020

The following individuals certify that they have read, and recommend to the Faculty of Graduate and Postdoctoral Studies for acceptance, the thesis entitled:

Trajectory planning with real time vision-based obstacle detection

submitted by **Xuemeng Li** in partial fulfillment of the requirements for the degree of **Master of Engineering** in **Electrical and Computer Engineering**.

Examining Committee:

Maryam Kamgarpour, Electrical and Computer Engineering, UBC
Supervisor

Mahdi Yousefi, Avestec Technologies Inc.
Supervisor

Abstract

Autonomous aerial vehicles are broadly used to assist human in dangerous or complex monitoring tasks, such as inspection of hard to reach high voltage power lines and oil pipes, monitoring and distinguishing wildfire, or improving precision farming. An autonomous navigation system can support drones or robots to move towards targeted positions without any external control. A fully functional autonomous system requires an integration on a wide range of algorithms, including trajectory planning algorithms, object detection with image processing and robust control algorithms for path following.

In this project, an implementation of trajectory planning with object detected from the video with depth information has been described. The location and size of the targets are determined through image processing and convex optimization with the depth information collected from Intel Real Sense Camera. Then the trajectory planning function based on Rapid-exploring random tree (RRT) algorithm will plan the path on the map which combined the detected obstacle according to the obstacle detection function. This process would ensure drones or robots reaching the target domain safely.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

Short	Long
RRT	Rapid-exploring Random Tree
RGB	Red green blue additive color model
ROS	Robot Operating System
SDK	Software development kit
FOV	Field of View

Acknowledgements

I would like to thank the supervision of Dr. Maryam Kamgarpour and Dr. Mahdi Yousefi.

Chapter 1

Introduction

Autonomous aerial vehicles and robotics are broadly used to provide human assistance on dangerous or complex tasks. Fields such as oil, mining, wildfire extinguishing, and precision farming are requiring more robotic involvement nowadays. To better adapting to duties' requirements, an autonomous navigation system is essential for the robots or drones to safely accomplish the work with minimal human control. In this project, an integration of the obstacle detection and trajectory plan has been implemented as the start point for the autonomous navigation system for future integration on robot's hardware. One of the main algorithms used in this project is the Rapid-exploring random tree (RRT) algorithm for path planning. Initially brought up by Steven M. LaVell and James J. Kuffner Jr (LaValle and Jr., 2001), the RRT algorithm could efficiently search nonconvex and high-dimensional spaces by constructing space-filling tree from samples drawn randomly in the search space and inherent grows towards large unsearched areas. The algorithm has been broadly discussed and implemented in education materials for robotics such as Python Robotics (Sakai et al., 2018). Many variants and improvements for motion planning have been studied and implemented. There are also comparison researches on improving the RRT algorithms (Iram Noreen, 2016). With all these variants, the original RRT algorithm holds its tidy and simplicity which is good to be used as a start point. Therefore, in this project, original RRT algorithm has been implemented as object-oriented with MATLAB. In original RRT algorithm, the next point of the tree is generated based on the direction to random point, selected step size and whether is on the obstacle.

The intel RealSense Camera D400 series is a stereo vision depth camera system (rea, 2019). The small size and ease of integration of the camera sub system provides integration flexibility on wide range of products including drones, robots, virtual reality, PC peripherals and home surveillance. Works have been done for image processing with RealSense Camera. For example, Chang utilized the OpenCV DNN object detection package within RealSense

Camera to outline the obstacles in RGB content in real-time (Chang, 2018). To better utilize the depth information, Song and Xiao implemented depth maps for object detection and designed a 3D detector to overcome the difficulties for recognition based on rendering hundreds of viewpoints of a CAD model to obtain synthetic depth maps (Song and Xiao, 2014). In this project, Intel's RealSense Depth Camera D435i has been used for object detection because of its handy size, ease of integration with cross-platform open source Intel RealSense SDK application and stereo depth camera which designed to maintain its calibration throughout lifetime.

In the following sections of this report, detailed implementation process and discussion on the results will be described. For algorithm implementation, there will be explanations on how to generate an RRT tree for trajectory plan, how to identify the obstacle with the depth video file from RealSense camera and how to integrate the image processing with the trajectory plan. The accuracy of the measurement for the camera will also be discussed. And in the section after, detailed assessment of the work including error source, advantages and disadvantages, and future steps for improvements will be discussed.

Chapter 2

Algorithm and Implementation

The implementation of the project has been done in MATLAB. Rapid-exploring Random Tree Algorithm has been implemented for trajectory plan and object detection algorithm has been built with depth information extracted from RealSense Camera. The overall implementation and the instruction has been committed to GitHub at https://github.com/luckymeng7/EECE597.

2.1 Rapid-exploring Random Tree Algorithm for Pathway Planning

Rapid-exploring Random Tree (RRT) algorithm was introduced by Steven M. LaVell and James J. Kuffner Jr (LaValle and Jr., 2001). To efficiently search nonconvex and high-dimensional spaces, the algorithm constructs a space-filling tree from samples drawn randomly in the search space and inherently grows towards large unsearched areas.

The output of the RRT algorithm is a tree(T) that contains all of the randomly sampled nodes(q) with connection to their inherit points. With object-oriented programming, each of the sampled node has been assigned as an instantiation of *Node* class with properties including index, parent index, position and path to node. The output tree is an instantiation of *RRT* class which is defined to contain all generated nodes, with properties including total number of the nodes, initial node, positions of all existing node and a list of all existing nodes. A new node will be sampled in each iteration and the RRT tree will be updated to include the new node if the path to this new node has avoided the obstacle. To generate a new node(qnew), a random point(qrand) will be firstly generated within the map range. Then it will be connected to the nearest node(qnear) on the tree. On the connection line, a

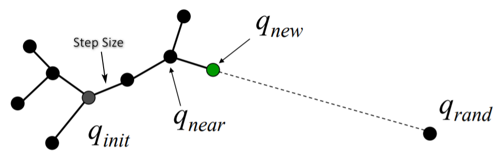


Figure 2.1: New Node Generation(Choset, 2015)

point with max-step distance (step size) to the nearest point will be selected as the new node if it is confirmed as clear with obstacle free function. Figure 2.1 shows the generation of one new node to the tree. The algorithm will use the initial point as the first *Node* of the tree, and update the RRT tree until the distance between the latest generated node and destination point is less than the step size.

For a general configuration space, the algorithm in pseudocode is as follow (Iram Noreen, 2016):

```
Algorithm T=(V,E) ← RRT*(qini)
T ← InitializeTree();
T ← InsertNode( , qini, T);

    for i = 1 to K do
        qrand ← RAND_CONF()
        qnearest ← NEAREST_VERTEX(qrand, T)
        qnew ← STEER (qnear, qrand, Δq)
          if Obstaclefree(qnew) then
            qnear ← Near( D, qnew, T);
        qmin ← Chooseparent(qnear, qnew, qnearest);
        T ← InsertNode(qmin, qnew, T);
        T ← Rewire(T,qmin, qnew, qnear);

    return T
```

- "←" denotes assignment. For instance, "largest ← item" means that the value of largest changes to the value of item.
- "return" terminates the algorithm and outputs the following value.

The inputs for the algorithm are initial position, destination position, max

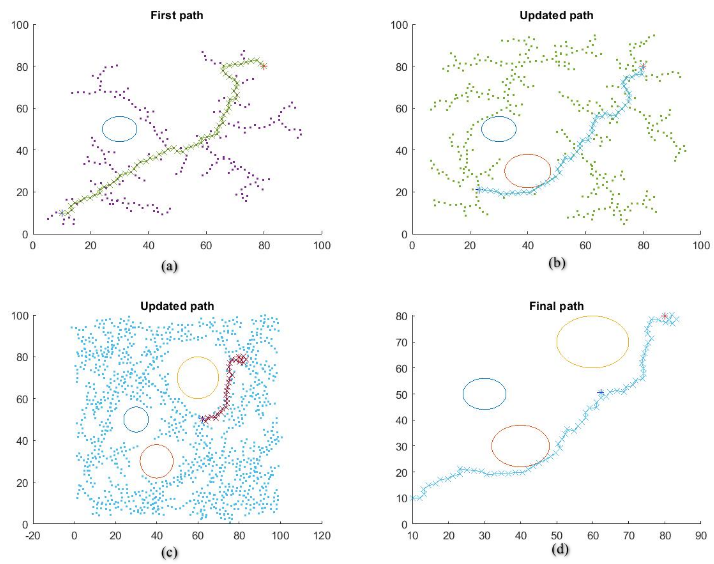


Figure 2.2: Path updated when new obstacle detected

step size, obstacle size and position and the overall map size. The output for the function is a planned path and the overall tree.

Real-time obstacle detection has also been implemented. As the robot moving forward, the function would recalculate the tree when it detects a new obstacle shows up. The current position would be taken as the new start point and the destination position would still be the same. Figure 2.2 shows path updated twice after initial path planned. This figure is generated by running script pathPlan_main.m in folder “PathPlanRRT” on GitHub.

At the beginning, a original RRT path was calculated from the start point to destination point. Figure 2.2 (a) shows the original path with one obstacle detected in the filed. Then in figure 2.2 (b), a new obstacle was added to the canvas after the robot moved along the original path for 10 steps. The current position at the time of (b) for the robot was the 10th Node on the original path. A new RRT tree was calculated to update the path which avoided the new obstacle. Similarly, in figure 2.2 (c), a new obstacle was added to the canvas after the robot moved along the updated path for another 30 steps. Another new RRT was calculated to avoid the new obstacle. Then the robot would move on with the updated path to the

Figure 2.3: Obstacle detected in depth field

destination point. Figure 2.2 (d) shows the actual path of the robot moving from start to end point when new obstacles showed up along the way.

The density of the tree shows the total generated *Nodes*. Mode random points generated to get the path leads to more *Nodes* on the tree. Small step size or long distance between the start and end point would increase the number of calculating iterations for the RRT. Other variants such as the position of obstacles and the distance between obstacles would also influence the number of the total *Nodes*.

2.2 Image Processing for Obstacle Detection

With the video stream captured with Intel RealSense camera, the depth information could be used for object detection.The output file type of the RealSense camera is bag file. Bag is file format in ROS for storing ROS message data (Saito, 2015). An example of the bag file is located at https://github.com/luckymeng7/EECE597/tree/master/Videos. With MATLAB's ROS toolbox, the bag video was imported and processed as depth frames and RGB frames. A function has been implemented to identify the obstacle based on the depth information. On each of the frames, a mask will be created in which all the pixels that has depth value within the sensitive detection range will be set to 100 and the rest of pixels will be set to 0. The comparison between regular RGB image and detected obstacle area are shown in figure 2.3

In order to project the obstacles properly onto the map where the trajectory is planned, top views for each frame are created by switching the depth value with the height index. By looping through the columns number of the depth image, using the smallest depth value of each column as the row number and assign row number of the related point to the value of that pixel for the top

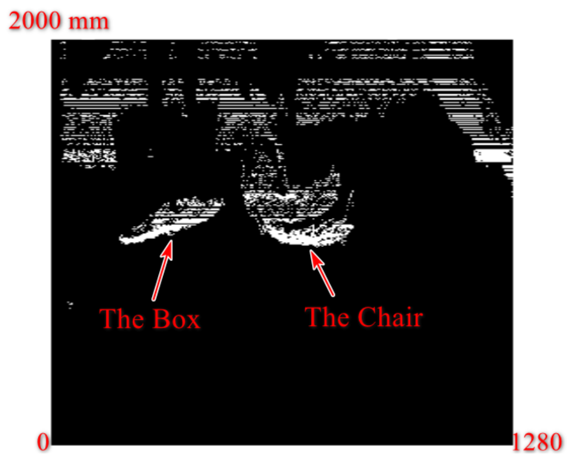


Figure 2.4: Top view based on depth information

view. The reason of using the top view for the obstacles is that the obstacles on map in figure 2.2 align as the same horizontal level as the robot. The x and y axes in the map is the width and distance value in robot's camera. A distance threshold is used to set the limitation of the depth size. For example, only objects within two meters to the camera would be captured as object and saved in top view as figure 2.4.

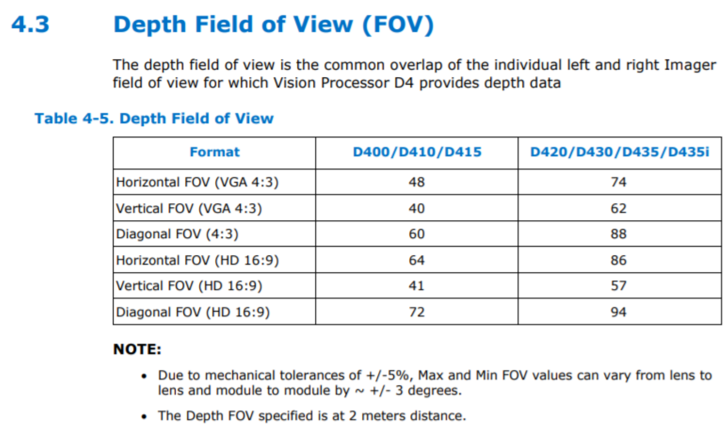
Figure 2.3 and 2.4 are generated by running script videoProcess_main.m in folder "ObjectDetection" on GitHub. Output videos are created by combining frames of top views or obstacle masks on the RGB frames.

2.3 Measurement and Calculation of the Object Size

Based on the datasheet of the RealSense camera (rea, 2019), the horizontal field of view (FOV) for depth image is 74 degree and vertical field of view for depth image is 62 degree.

Based on the definition of FOV in figure 2.6, the actual width and height based on the pixel value could be calculated as:

$$Actualwidth = \frac{widthpixel}{resolutiononwidth} \cdot depth \cdot \tan(\frac{74}{2}) \cdot 2$$



4.3 Depth Field of View (FOV)

The depth field of view is the common overlap of the individual left and right Imager field of view for which Vision Processor D4 provides depth data

Table 4-5. Depth Field of View

Format	D400/D410/D415	D420/D430/D435/D435i
Horizontal FOV (VGA 4:3)	48	74
Vertical FOV (VGA 4:3)	40	62
Diagonal FOV (4:3)	60	88
Horizontal FOV (HD 16:9)	64	86
Vertical FOV (HD 16:9)	41	57
Diagonal FOV (HD 16:9)	72	94

NOTE:

- Due to mechanical tolerances of +/-5%, Max and Min FOV values can vary from lens to lens and module to module by ~ +/- 3 degrees.
- The Depth FOV specified is at 2 meters distance.

Figure 2.5: Depth Field of View from camera manual (rea, 2019)

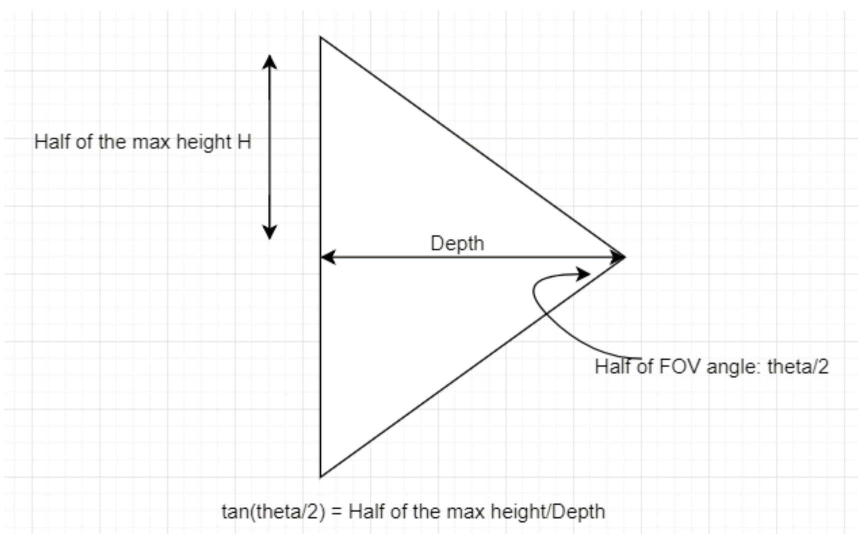


Figure 2.6: Definition of field of view

$$Actualheight = \frac{heightpixel}{resolutiononwidth} \cdot depth \cdot \tan(\frac{74}{2}) \cdot 2$$

In order to use actual size as base point to do the calibration, I transferred the above equations into:

$$\tan(\frac{74}{2}) \cdot 2 = \frac{rp \cdot L1}{l1 \cdot D1}$$

Where,

- rp is the resolution on width or height direction,
- L1 is the actual width or height,
- l1 is the pixel number on width or height,
- D1 is the depth distance.

With another transformation for the above equation, we will get:

$$\frac{L1}{l1 \cdot D1} = \frac{L2}{l2 \cdot D2} = C$$

With the above function, we could calculate the constant **C** with the acutal length L1 measured by ruler, the length in pixel on the RGB frame l1 and the depth value D1 from the depth frame of the first object. Then, L2 could be calculated for the second object based on C, the length in pixel on the RGB frame l2 and the depth value D2 from the depth frame. To increase the accuracy of the calculation for L2, multiple L1 objects are measured. Then L2 are calculated with the average of C that calculated from first group of objects.

The tables 2.1 and 2.2 are the measurements of the first group of objects to calibrate the measurement with resolution 1280x720 for depth image and RGB image. Regular-shape objects are chosen for the calibration to increase the accuracy. The measurement of actual height and width are done with a ruler on the object and the measurement of height and width in pixel is measured on the RGB image directly. The depth value of the center point of each object has been used as the depth value for the overall object. The unit for actual length are millimetre.

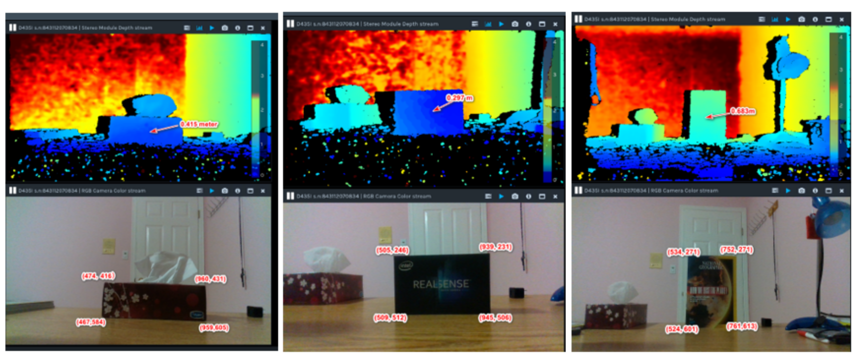
Figure 2.7 are the images for the above objects (1,2,3 from left to right):

Table 2.1: Measurements for calibration (Height)

Item	Depth(mm)	Height(Actual)	Height(Pixel)	Ch
1	415	75	171	0.00106
2	297	90	270	0.00112
3	683	252	336	0.00109

Table 2.2: Measurements for calibration (Width)

Item	Depth(mm)	Width(Actual)	Width(Pixel)	Cw
1	415	225	489.0	0.00111
2	297	142	435.0	0.00109
3	683	173	227.5	0.00111



(1) (2) (3)

Figure 2.7: Images for calibration

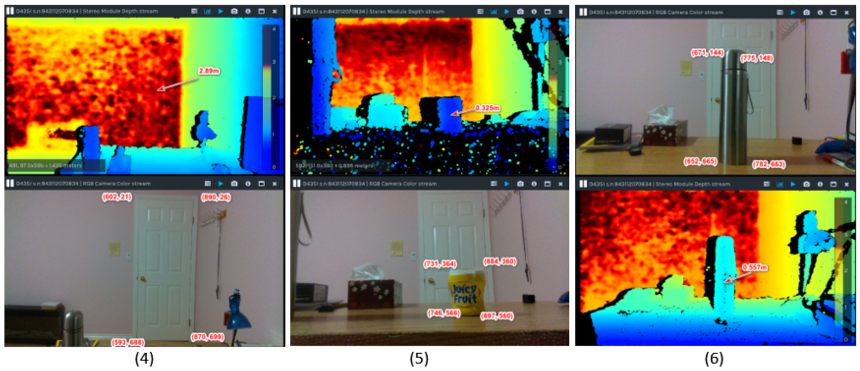


Figure 2.8: Images for measurement test

Table 2.3: Information from image

Item	Depth(mm)	Height(Pixel)	Avg. Ch	Width(Pixel)	Avg. Cw
4	2890	670	0.00109	225	0.0011
5	325	201	0.00109	152	0.0011
6	557	518	0.00109	117	0.0011

To calculate with equation

$$\frac{L1}{l1 \cdot D1} = \frac{L2}{l2 \cdot D2} = C$$

, the following three objects in figure 2.8 are measured (4, 5, 6 from left to right):

The comparison of calculation and measurement are as tables 2.3, 2.4 and 2.5, where

$$L1 = C \cdot l1 \cdot D1$$

According to the datasheet (rea, 2019) of RealSense Camera, there is a mechanical tolerance of +/-5% for the camera. Therefore, the error measured is within the tolerance. As we could see, the error will be relatively smaller if the shape of the object is rectangle and facing the camera perpendicularly. Physical measurement on object with anomalous shape will introduce more error compare to rectangles. Also, to simplify the calculation, depth value of the center on the object was taken to calculate the constant C and later the actual height or width. However, the depth value at the center of the

Table 2.4: Measurement Result and Error Rate on Height(mm, percentage)

Item	Calculated Height	Actual Height	Error Rate
4	2110.0	2100	0.48
5	71.2	80	11.00
6	314.5	325	3.20

Table 2.5: Measurement Result and Error Rate on Width(mm, percentage)

Item	Calculated Width	Actual Width	Error Rate
4	898.0	920	2.4
5	54.3	60	9.5
6	71.7	82	12.6

water bottle is different from the depth value for the edge. This deviation on depth value will introduce more error. Objects not facing the camera perpendicularly will also introduce errors with the misalignment for the width and height.

To increase the accuracy for the calculation, two approaches could be done in the future. Firstly, instead of using the depth value at the center of the object to represent the distance to the whole object, we could calculate average depth of the object and use it to calculate the dimension of the object. The other method is to calculate the width and height with the coordination in 3D which combines both depth and RGB information.

2.4 Combination of Obstacle Detection and Trajectory Plan

In order to integrate the detected obstacle onto the trajectory plan, the top view of detected obstacle will be projected on the map canvas after re-sizing and rotation. Assuming the initial direction of the camera is from start point to destination point. Also, the current map size is on a dimension of 200x200 pixel, the top view of the obstacle will be re-sized to meet the mapping scale and rotated based on camera orientation. An obstacle mask based on the top view frames will be shifted onto the canvas with the bottom

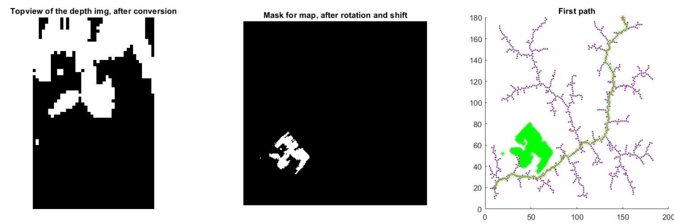


Figure 2.9: Detect Obstacle on Map

center point of the mask coincide with the camera's current position, which is the initial position at beginning. When new obstacles detected as the camera moving forward, the trajectory plan function will re-calculate the path with generating a new RRT tree. Figure 2.9 shows the obstacle mask generated with the top view frame from section 2.2 and an RRT path is calculated with obstacle mask projected on map canvas. The scale for the map canvas is 1:31.25, which means that 31.25 pixels on the map represent 1 meter.

Once the camera starts to move based on the generated path, the object detection and path recalculate will continue. After the start point, the direction of the camera will be changed to the vector difference between current coordinate and previous coordinate. The newly generated mask will be re-sized, shifted and rotated to the new position and direction. If the obstacles condition has been changed since last projection, a new RRT tree will be generated and the camera will move along the new path.

In order to increase the safety and avoid the drone being trapped in the anomaly shaped obstacle, a minimum volume ellipsoid has been generated to cover all of the obstacle points as shown in figure 2.10. The algorithm to find the ellipsoid is discussed in the textbook "Convex Optimization" [(Boyd and Vandenberghe, 2009)and the function usage in MATLAB is described in [(Mutapcic, 2019)

Because of the long RRT calculation time and video processing time, the object will be checked every 5s when the robot's moving speed is 0.1m/s

2.5 Update the path as obstacle moving

Two scenarios have been demonstrated to show that the implementation could recalculate the path when the camera detects the object change. Mul-

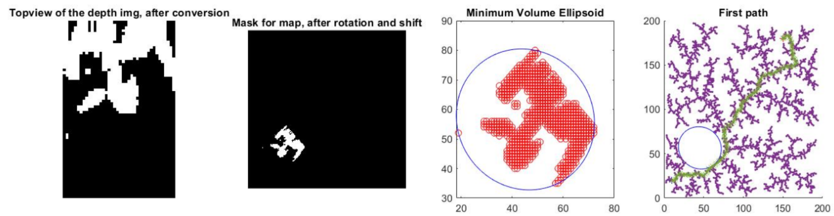


Figure 2.10: Detected obstacle with minimum volume ellipsoid covered

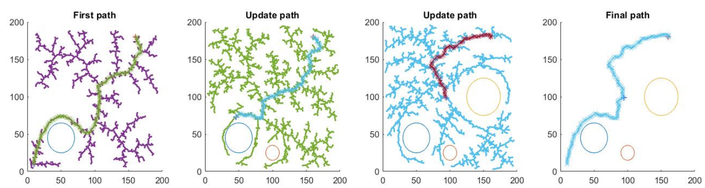


Figure 2.11: Path updated when new objects detected

tiple obstacles would show up in the global map as the robot moving along the path. Also, the original obstacles would change the position from time to time. The program would recalculate the RRT tree when changes happen to the environment.

Ideally, the path would only be recalculated when new obstacles are added in the map as shown in figure 2. However, in reality, with the vibration and noise from the camera, the informaiton of all the detected obstacle will change every time checking the camera. If the function is set to recalculate the RRT whenever detecting the change in obstacle, it would keep recalculating. Therefore, in the implementation, an update rate as has been defined to set the time between the updates of the camera streaming. In the example demo, the update rate has been set to 20 iterations. The updated camera video will be checked when the robot moves along the previously defined path for 20 steps. With the scale as 1:31.25 and step size as 2, the camera will be checked everytime when the robot moves for 1.28 meter. This update rate is defined in integrate_main.m in folder “Integrate” on GitHub and could be modified as needed.

Figure 2.11 shows the path updated when new objects detected. And figure 2.12 shows the path updated when a single objet changes position as the robot is moving.

The update for the input of obstacles are done by streaming the video sep-

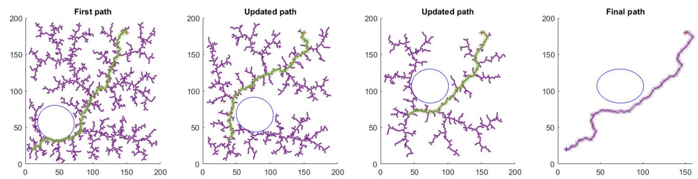


Figure 2.12: Path updated when object moved

arately at different time point. Then feeding these videos are fed into the system which would recalculate the path. This process proves the concept of the capability on updating the obstacles in real time. An actual real time object detection during path planning could be implemented with RealSense SDK in the future.

Chapter 3

Discussion

With the implementation from this project, a trajectory planning methodology that is capable of detecting the obstacle with camera has been implemented. In this section, analysis on the RRT algorithm, error rate and source for the measurement with RealSense camera, limitation for the real-time implementation and future work will be discussed.

The advantage for RRT algorithm is that a collision avoidance path from initial point to the destination would for sure be created, as long as it exists. However, as the algorithm will generate random points to build the tree exhaustively, the computation time will increase dramatically when the obstacles are close to the destination point, which showed in figure 2.2. It is important to choose a proper value for the step size between each node on RRT. If the step size is too small, there will be too many nodes to be generated between the two points on the map. This would increase the calculation time. However, if the step size is too large, chances for finding a proper path between obstacles would be decreased, especially for those anomaly shape paths between two obstacles.

The objects captured by the camera were measured. The measurement result defines the scale for transferring the obstacle to the map canvas. In order to increase the accuracy for the measurement, several reference objects have been used for calibration. However, the error rate for the measurement ranges from 0.48% - 12.6%. We noticed that when the object to be measured has a flat surface whose distance to the camera is about the same across the surface and has a regular shape such as rectangle, the error rate would be lower. And when the object has an irregularly or convex shape such as cylinder, the error rate would be higher. The reason is that the depth information for the flat surface is mostly uniform, but the depth information on the cylindrical surface varying. Current calculation for measurement is based on the depth info at the center point of the object. Therefore, there will be a bias on the depth value. Similarly, the error from varying depth value would be introduced if the surface is not facing the camera perpendicularly. To solve this problem, we could use 3D coordination which combined both RGB and depth information for the measurement.

A limitation for the current implementation is the path planning is not online. Currently, the scripts are importing the recorded video for image processing and object detection. Then the processed results were imported as obstacle information to the pathway planning algorithm. The video reading process take more than 10s to finish and this set a bottleneck in implementing it in real-time. As the SDK for RealSense camera are implemented with C++, the development environment should be changed to C++ for further implementation.

In the future, there are two main directions for this project to be continued. One is the implementation of the object detection in real-time with OpenCV packages provided by RealSense. With the RealSense SDK, saving the internal processed result of the video streaming is no longer needed. The depth information detected in real-time will be input directly to the object detection function, converted to top view and then added on the pre-defined map for the next path planning calculation. The other direction for the future work is the control algorithm for path following. A feedback control algorithm should be implemented to support the drone to follow the planned path properly. At the end, a real-time system combining the functions of obstacle detection, path planning and path tracking would fully support the atonomous navigation of the robotic system.

Chapter 4

Conclusions

The trajectory planning algorithm implemented in this project is capable to detect the obstacle in the camera and generate a pathway to avoid the collision. RRT algorithm has been integrated with the image processing on the video captured by RealSense camera. In the future, control algorithm for path following and object detection in real-time with OpenCV packages should be implemented

Bibliography

(2019). *Intel RealSense D400 Series Product Family.*

Boyd, S. and Vandenberghe, L. (2009). *Convex Optimization.* Cambridge University Press.

Chang, C. Y. (2018). Realsense opencv dnn object detection. =https://github.com/twMr7/rscvdnn.

Choset, H. (2015). Robotic motion planning: Rrt's. *Visited on*, pages 09–27.

Iram Noreen, Amna Khan, Z. H. (2016). A comparison of rrt, rrt* and rrt*-smart path planning algorithms. *IJCSNS International Journal of Computer Science and Network Security*, 16(10):20–27.

LaValle, S. M. and Jr., J. J. K. (2001). Rapidly-exploring random trees: Progress and prospects. pages 293–308.

Mutapcic, A. (2019). Minimum volume ellipsoid covering a finite set.

Saito, I. (2015). Bags. =http://wiki.ros.org/Bags.

Sakai, A., Ingram, D., Dinius, J., Chawla, K., Raffin, A., and Paques, A. (2018). Pythonrobotics: a python code collection of robotics algorithms.

Song, S. and Xiao, J. (2014). Sliding shapes for 3d object detection in depth images. 13th European Conference on Computer Vision.