1 ABC

abc

1.1 Robot Global Vision Update

1.1.1 Introduction

Robot Global Vision Update is a component of Autonomous Robot system. This component is designed and implemented in python programming language, with missions determining open sights and identifying open points based on the map figure from the SLAM node (belonging to the Robot System component). After determining the best open point to go to, this component analyses and publish a message that is a sequence of points the robot has to go through to get to the next point.

1.1.2 Role

In essence, in the Autonomous Robot system, Robot Global Vision Update has the role of identifying and storing information from around environment, thereby making decisions about the next points that the robot needs to go to.
Specifically, after Robot Global Vision Update has been started, it will go through the following works below:

- Receive two values "x" and "y" from users that are used to represent the actual coordinates of the Goal.
- The goal coordinates will undergo a mapping step from the real coordinates (fixed) to the robot's relative coordinates (set when the ROS Bringup node starts). The mapping step is implemented in the Paper Finder component.
- After receiving the goal coordinates that have been mapped, this component begins to analyse and send the set of points, including the current coordinates and the coordinates of points that need to be traversed, to the Robot Motion Node component.
- After receiving the done moving message from Robot Motion Node, this Node will continue to calculate and return the next point to go. This process repeats until the robot reaches its goal.

1.1.3 Published - subscribed topics

Because Autonomous Robot system uses on a publisher-subscriber protocol, the Robot Global Vision Update is a node of the system, whith sends and receives messages between the nodes through publishing and subscribing to a specific topics.

About topics which registered to publish:

- /done_change_view: is the topic used to publishing/subscribing requests to recognize the actual coordinates from objects on the camera view. After Robot Global Vision Update initialized, it will send to this topic the goal coordinates entered by the user with the format "[{x coordinate}, {y coordinate}]".
- /cmd_moving: is the topic used to publishing/subscribing messages which is a set of points to move to. After determining the next point to go to, Robot Global Vision Update will send a message to this topic which has the format is a set of points with the first point being the current coordinates of the robot and the remaining points being the points that the robot has to go through.

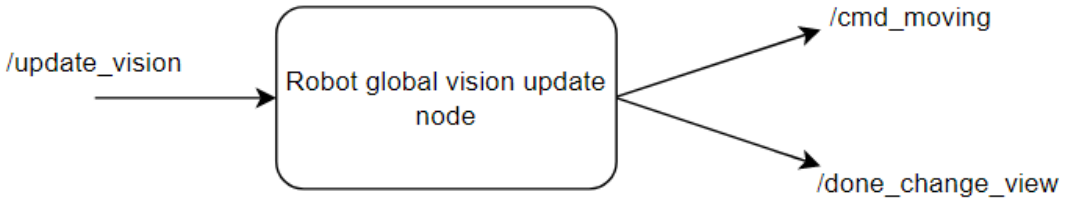


Figure 1: Topics are published or subscribed by Robot Global Vision Update

About topics that have been registered for subscription:

- /update_vision: is the topic used to publishing/subscribing a message containing the coordinates of a mapped destination or a confirmation message that notices the robot moved to the next point. Robot Global Vision Update subscribes to this topic to receive information about the mapped destination coordinates and passed a message to start the calculation and return the next set of points to go to.

1.1.4 Flow chart

abc

1.1.5 State machine

abc

1.1.6 Algorithms are applied

Since the Autonomous Robot system applies the method of moving according to the sets of points in unknown environment, it is very important to prioritize choosing safe open points. The open points must ensure the safety of the robot during and after moving. In addition, the travel distance must be optimized to save the robot's power energy.
Therefore, Robot Global Vision Update component applied algorithms to process data, involved in:

- Processing obstacles taken from SLAM map through Canny algorithm.
- Determining open sights and local open points at the current position of the Robot.
- Saving open points to visibility graph and reconstruct graph.
- Find next point to go to and approximately path to go to next point with has optimal distance.
- Remove the non-optimal points inside approximately path.

1.1.6.1 Processing obstacles taken from SLAM map through Canny algorithm

SLAM map is the image returned after the SLAM algorithm processes the data scanned by Lidar. We used Canny algorithm to analyse SLAM map and it return a list of obstacles, which are represented by a list of points that are the coordinates of the obstacle's vertices.
The algorithm determines the open area based on the obstacles inside robot vision (has value between $120mm - 3,500mm$, default is $1,000mm$). However, obstacles taken from the SLAM map is not a true polygon in the geometry theory, it has noisy vertices that form redundant edges, which affects the algorithm to enlarge the obstacles (Described in detail below).

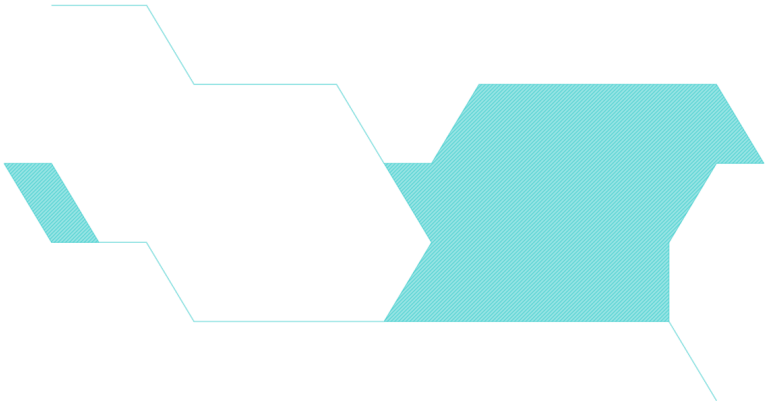


Figure 2: Obstacle which has redundant edges

Regarding the concept of an obstacles represented by a set of points, we convention that all obstacles applied in the program are polygon satisfying the following three conditions:

- There is no three collinear vertices lying on the same line. Experimentally, we saw that there are four possible cases of three collinear vertices.

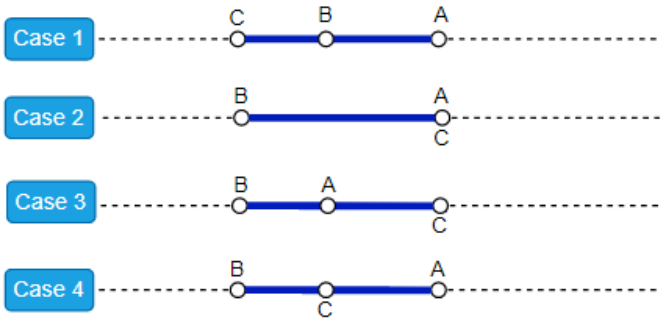
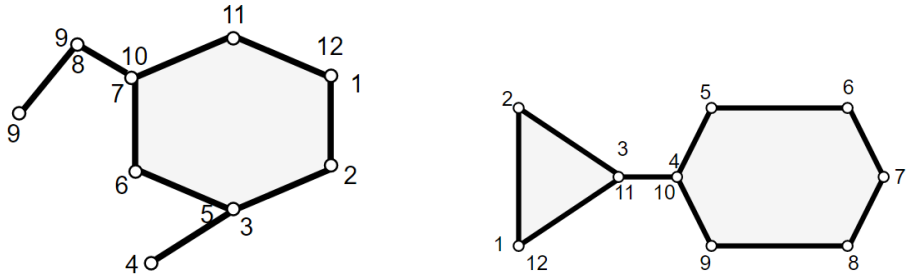


Figure 3: Four cases that 3 collinear vertices in order A→B→C

- There is no two sides cross each other. Experimentally, we find that this condition is always true for the obstacles after being processed by Canny algorithm.
- The obstacle is a polygon with no internal holes. This condition means that in a set of



(a) Obstacle is not true polygon (b) Obstacle is true polygon

Figure 4: List of obstacle with vertex numbers as the index in list

points of the obstacle, the begin and the end points always coincide.

Based on the conventions about obstacles that described above, we need to pre-process obstacles data before applying the obstacle zoom in algorithm. The algorithm which we implemented for obstacles pre-processing step named *Three collinear adjacent vertices removing.*

* **Three collinear adjacent vertices removing algorithm**
  Three collinear adjacent vertices removing algorithm is implemented to reconstruct obstacles data whenever it found three collinear adjacent vertices in the obstacle.
  In the first condition of an accurate polygon as described above, we clarified that if an obstacle is an accurate polygon, there are no three collinear adjacent vertices in the obstacle. That is why we need to implement and apply this algorithm to the Robot Global Vision Update component.
  In *Three collinear adjacent vertices removing* algorithm, we compare the length of three line segments formed by pair of two vertices in three adjacent vertices. If there exists a line segment that is equal to the sum of the other two line segments, then they are collinear. Based on two vertices which are the two ends of the longest line segment, we know the order of the three vertices and the vertex are between 2 other vertices. Since we consider every 3 vertices in a time, there are 4 cases as described in Figure {...}. However, We minimize to 3 cases because case 1 and case 2 of figure A have the same treatment in this algorithm.
  For example, if we assign current vertex is B, previous vertex is A and after vertex is C, by comparing the lenght of AB, BC and AC, we know that there 3 cases:

  - If $AB + BC = AC$, then B is between A and C. In this case, the algorithm will pop B out of vertices list of the obstacle.

    After processing, the list of vertices will change from $[..., A, B, C, ...]$ to $[..., A, C, ...]$.
  - If $AC + BC = AB$, then C is between A and B or C duplicates with A. In this case, the algorithm will add new vertex named B' such that BB' is perpendicular to AB and the length of BB' is equal to γ which is very small (fixed value).

    After processing, the list of vertices will change from $[..., A, B, C, ...]$ to $[..., A, B, B', C, ...]$.
  - If $AB + AC = BC$, then A is between B and C. In this case, the algorithm will add two new vertices named B' and A'. B' is the same as described in case 2 and A' is the

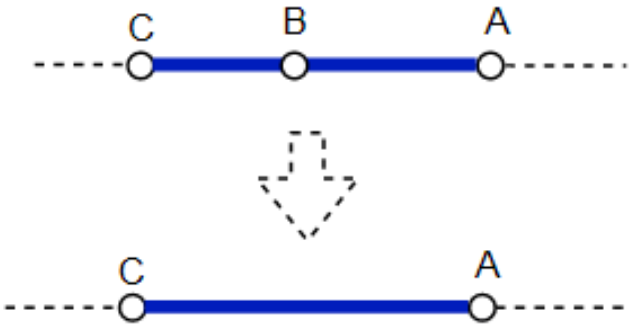


Figure 5: Case 1 solved by the algorithm in order A→B→C

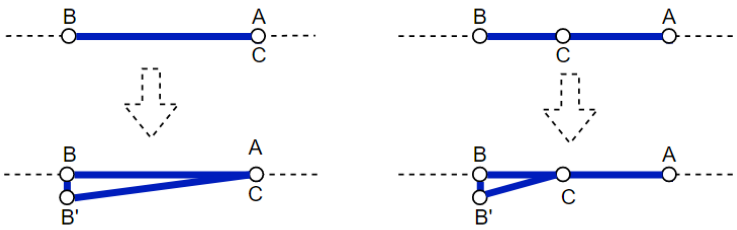


Figure 6: Case 2 solved by the algorithm in order A→B→C

same as A.

After processing, the list of vertices will change from $[..., A, B, C, ...]$ to $[..., A, B, B', A', C, ...]$.

– Otherwise, three adjacent vertices that are being considered, is not collinear.

In reality, three collinear adjacent vertices usually noisy signals when slam gets data from lidar. However, we implemented the algorithm to handle this case without discarding them. If we choose the way that removes the collinear adjacent vertices, there is a risk that the obstacle may be changed from reality, which affects the safety of the robot during movement. Instead, we add or remove vertices without changing the original structure. This way means that accepting noisy data is also an obstacle, but this noise data usually be outside the robot's vision, so we don't need to care about them.

∗ **Obstacles zooming-in algorithm**

The *Obstacles zooming-in* algorithm is the algorithm that zooms-in obstacles by value of robot radius (fixed value, default is $115mm$).

The Autonomous Robot system is constructed by path-planning method, which means the robot moves point by point in the set of points processed by the Robot Global Vision

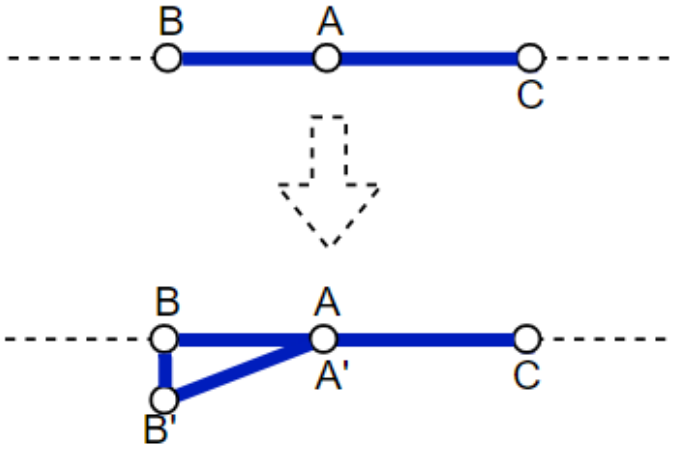


Figure 7: Case 3 solved by the algorithm in order A→B→C

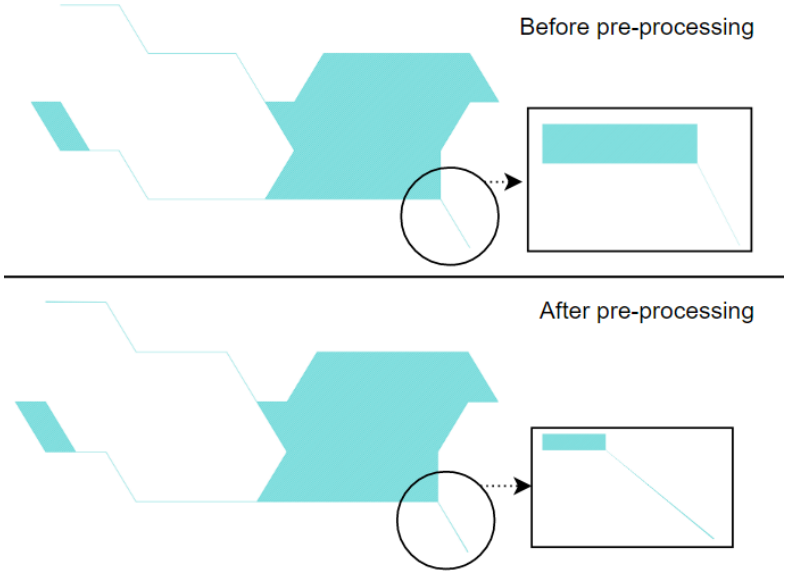


Figure 8: Obstacle before and after pre-processing step

Update component. In the open sights and open points finding algorithm (described below), open sights is determined by checking vertices of obstacles inside robot's vision. Therefore, if we don't zoom-in obstacles before finding open sights and open points, the algorithm can determine an open point that is too close to the obstacle. This happens because the algorithm only considers the robot as a point, despite in fact the robot is a block with a radius greater than 0. For this reason, the application of *Obstacles zooming-in* algorithm is necessary to ensure the selection of an open points, so that the robot will be safe to move

to that points.
For each edge of an obstacle, the algorithm finds normal vector and move the vertices in the direction of the vector found. The algorithm is always sure that that new obstacle cover old obstacle and all points on the polygon after zooming will be at a distance to old polygon greater than or equal to the robot radius.
In detail, The algorithm consists of these step:

– Check if the normal vector direction is out of the obstacle.

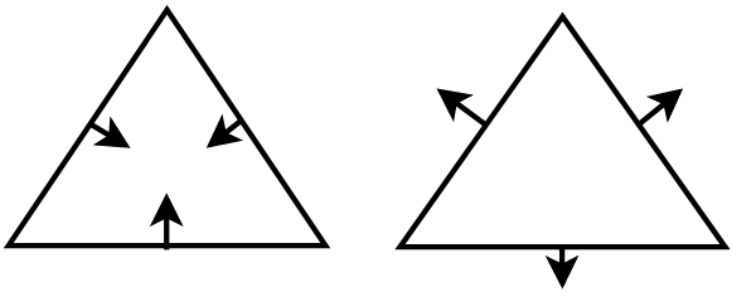


Figure 9: Normal vectors direction inside (left) and outside (rignt) of obstacle

We created an algorithm to check if the normal vector direction is out of the obstacle, that is named left hand direction algorithm.
The left hand direction algorithm is implemented based on the *Jordan Curve Theorem*. It says that "a point is inside a polygon if, for any ray from this point, there is an odd number of crossings of the ray with the polygon's edges". In the left hand direction algorithm, we find a point based on the normal vector of an edge of the obstacle, then we choose any ray peeking out from the selected point and check its number of intersection points with the obstacle. If intersection points is odd, which means current normal vector direction is inside ob the obstacle, the left hand direction algorithm will return False. Otherwise, it return True.

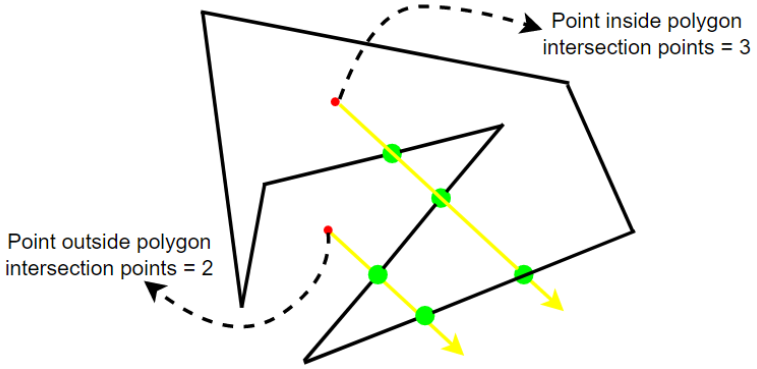


Figure 10: Jordan Curve Theorem

– After got the corrent direction of normal vector, the algorithm will traverse each pair of adjacent edges of the obstacle. In this step, there are two cases that need to be process:

  ◦ If two edges form a concave vertex, then the algorithm will find normal vectors for 2 edges, then move the two edges in the direction of the normal vector. Finally, the algorithm calculates the intersection point of two edges after moving and updating the new vertex is the intersection of those two edges.

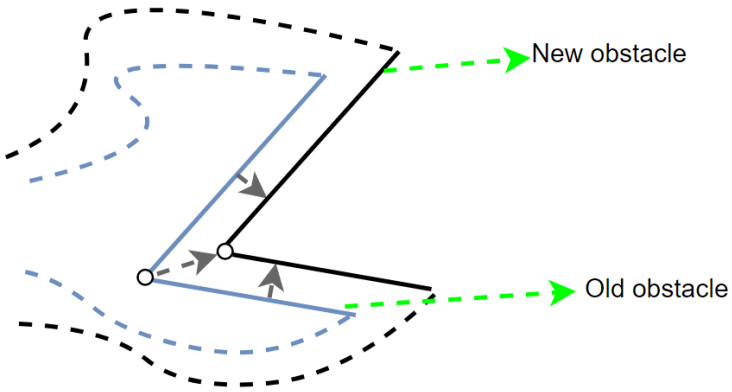


Figure 11: Jordan Curve Theorem

  ◦ If 2 edges form a convex vertex, then the algorithm will find normal vectors move the edges like the same with case 1 above. After that, it find an bisector of the vertex in the direction out of the obstacle, the length of bisector is equal to robot radius.
  The bisector is calculated based on vector of 2 edges. This is the reason we need to process the three adjacent collinear points before using the obstacles zooming-in algorithm. If there are 3 adjacent collinear points, then the normal vector of the 2 sides and the bisector have the same direction, so it is impossible to find the intersection between them.
  Finally, it calculate intersection points of 2 edges with a line perpendicular to the bisector which has distance of robot radius from the old vertex. The old vertex will be replaced by 2 intersection points calculated.

  After traversing all pairs of adjacent edges, the algorithm return an obstacles list that contain obstacles after zooming-in as the figure below.

As we see the obstacle after zooming-in, there exists the wrong edge insde obstacle. Therefore, to ensure that the algorithm for finding open sight and open points works as expected, the obstacle needs to go through a post-processing step to remove the vertices inside the obstacles.

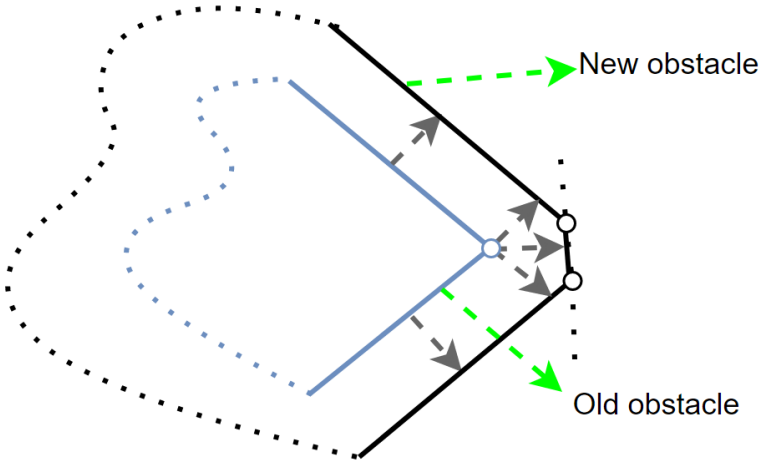


Figure 12: Jordan Curve Theorem

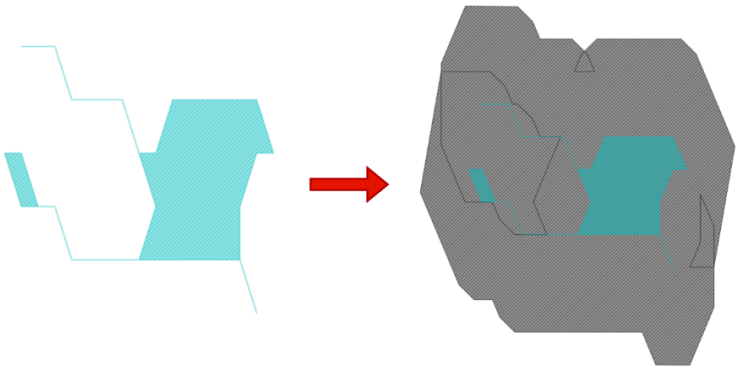


Figure 13: Jordan Curve Theorem

* **Obstacles post-processing algorithm** *Obstacles post-processing algorithm* is used to remove any edge inside obstacle.
The open sights and open points finding algorithm applied in case obstacle is correct polygon (as described in *Three collinear adjacent vertices removing* algorithm). As we saw the result after obstacle zooming-in step, the obstacle exists edges and vertices inside it. The cause of this one is that the obstacle actually has a lot of edges close together, so when applying the zooming-in algorithm, edges after zooming will intersect inside the new obstacle, thereby leading to the formation of vertices and edges inside it.

  The algorithm keep one edge has index $i : i + 1$ of the obstacle and traverse all remaining edges afterwards in list (which has index bigger than $i + 1$). If it found an edge index $j : j + 1$ which intersects an edge keeped, it would pop all point from $i + 1$ to j and append new vertex is an intersection point of 2 edges under review.

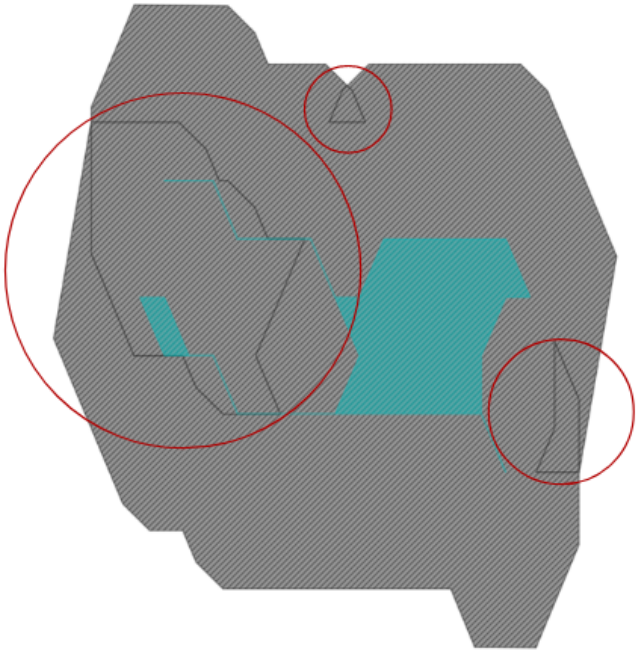


Figure 14: Zoomed-in obstacle has edges and vertices inside