Research on a Costmap that can Change the Turning Path of Mobile Robot

Hao Wen¹,a, Jiamin Yang ²,b
¹ School of Mechanical Engineering, Guangxi University, Nanning, Guangxi, China
² School of Mechanical Engineering, Guangxi University, Nanning, Guangxi, China
a email: 1711393009@st.gxu.edu.cn
b Corresponding author’s b e-mail: 1711307004@st.gxu.edu.cn

Abstract. In the field of mobile robots based on ROS (Robot Operating System) platforms, A*, Dijkstra, and BFS (Breadth First Search) are widely used to obtain the shortest path. This approach, however, often results in paths that are close to the obstacles located in corners. In practical applications, cumulative positioning errors in the navigation process are produced by aging components, skidding wheels and so on. In addition, the local path planning of the robot does not completely follow the global path planning, so the robot may easily cross the inflation of the costmap and collide with obstacles when turning. This paper proposes a method to change the turning path of the robot by adding virtual obstacles to the obstacles in map corners. A global path is generated by the path planning algorithm along the virtual obstacle so that the robot can travel along a smooth track that is far from the obstacle when turning. The proposed algorithm ensures that the robot remains far from physical obstacles and avoids collisions.

1. Introduction
Costmap and path planning algorithms are core technologies for mobile robots based on the ROS platform. In such scenarios, costmaps serve the entire navigation process, so path planning can only be realized by costmaps.¹ Hereafter, we refer to the global map constructed by the mapping algorithm as the static map. However, it is not sufficient for robot navigation to rely only on a static map. This is because the environmental information cannot be updated in real time, and obstacles that have been moved in or out of the scene may not be fed back to the robot promptly. To address this issue, an obstacle layer is introduced into the costmap. This layer is specially designed to update the environmental information in real time. Although an obstacle layer is introduced, the navigation of the robot might need further safety improvement. Since a cumulative position error exists, collisions are likely if the robot happens to be in direct contact with the obstacle.²

To address this problem, an inflation layer is introduced to the costmap. The inflation zone around the obstacle helps to prevent direct contact of the robot with the obstacle. The static map layer, obstacle map layer and inflation layer together constitute the original costmap. Some developers have designed extra functional layers based on the original costmap, e.g. Hallway Layer, Wagon Ruts Layer and Proxemic Layer, to address various practical issues.³ These extra layers effectively solve the prominent problems in the relevant industries.

In cases where the planned path is too close to the obstacles around the corner, common path planning algorithms such as A*, Dijkstra, and BFS encounter an issue in the navigation process.⁴ This is caused by the fact that the shortest path is often planned by the majority of algorithms as the optimal solution.
The shortest path, however, may not be practical as the cumulative position error in robots, and local path planning does not fully comply with global path planning. If the path is planned too close to the obstacles, then the robot might easily cross the inflation zone and hit the obstacle.\cite{5}

To solve this problem, some scholars proposed a solution to improve the path planning algorithm which can ensure not only the safe distance between the robot and the obstacles but also the algorithm efficiency (time complexity) and length of the planned path.\cite{6} Currently, common commercial robots are usually equipped with ARM-embedded processors with low power consumption, low cost and low frequency. To ensure that the planned path is as short as possible and to avoid obstacles, the algorithm shall inevitably become more complicated; thus, the robot will be hard to run and faces higher risks of lags or crashes.\cite{7}

To ensure that the algorithm is as simple as possible and to avoid obstacles, the length of the path shall inevitably be prolonged, which may greatly reduce the efficiency of the robot. Taking the Dijkstra algorithm as an example, the shortest distance from the current location to the destination in the map can be detected and searched based on this algorithm. The time complexity is $o(n^2)$. To make the planned path based on this algorithm far from obstacles, a genetic algorithm may be introduced; however, the following factors in path planning shall be taken into consideration:

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{r} & Crossover Rate \\
\hline \textbf{ps} & Point Scale for Encoding \\
\hline \textbf{ip} & Number of Intermediate Points \\
\hline \textbf{gn} & Number of Algorithm Evolution \\
\hline \textbf{sd} & Safe Distance of Single Point on Path \\
\hline \textbf{sp} & Number of Search Points on Path \\
\hline
\end{tabular}
\end{center}

Therefore, the equation for the time complexity of the new algorithm is $T = o(n^2) + o(rspsipgnsdsp)$. Compared with the calculation before improvement, the algorithm complexity has been significantly increased. For larger and more complicated situations, the calculations will become more difficult. Although there are many similar solutions, this is actually a balance among the safe distance, path length and algorithm complexity.

Of course, the inflation radius of the costmap can also be prolonged to avoid robot collisions in some solutions. The prolonged inflation radius can ensure a safer distance between the robot and obstacles as a whole, so collisions will not occur due to the long distance during navigation. However, the space for robot movement is not always wide enough, such as an indoor environment crowded with furniture or a logistic environment full of goods and shelves. If the inflation radius is prolonged as a whole, then the

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{fig1.png}
\caption{Schematic diagram of the proposed scheme.}
\end{figure}
working area of the robot will inevitably be decreased to reduce the efficiency of multirobot collaboration. If we only extend the safe distance of the area where collisions most likely occur (the corners of obstacles), then we can ensure the safe operation of the robot while avoiding the least occupation of the robot’s working space. The solution proposed in this paper is designed according to the aforementioned idea, as shown in Figure 1 (A).

In this scheme, we define a new functional layer that adds a circular virtual obstacle at the corner of the actual physical obstacle. The virtual obstacle has the same cost as a physical obstacle. A global path is then generated by the path planning algorithm along the inflation zone around the virtual obstacle so that the robot can travel along a smooth track that is far away from the obstacle when turning. We show that the proposed method greatly reduces the collision probability.

According to this solution, only relevant information is needed from the static map during the robot startup, and no additional computing resources are occupied during subsequent operations. Therefore, this solution can ensure a safe distance without an additional computational burden on the robot by means of the most primitive and shortest path-planning algorithm. Meanwhile, it can be seen from Figure 1(B) that the method of adding virtual obstacles partially will ensure less occupation of the robot’s working space compared with the method of prolonging the inflation radius as a whole. Please refer to Table III in the Experiment chapter for detailed statistics. To realize this method, two core issues should be addressed:

1) How can virtual obstacles be synchronized with the costmap?
2) How can the obstacles’ inflection point coordinates be located in the map?

We address the above issues in the following. In Section 2, we introduce how to synchronize the virtual obstacle to the costmap. Then, in Section 3, we propose a scheme to locate the obstacles’ inflection point coordinates in the map. We then verify the efficiency of the proposed method in Section 4 using simulations and experiments.

2. The Virtual Obstacle
In this section, we provide a brief description of the costmap, and then, we elaborate on our proposed virtual obstacle concept.

2.1. The Costmap Operation Mechanism
A functional block diagram of the costmap is presented in Figure 2, which illustrates the relationship between the main interfaces and plug-ins. [8]

![Fig. 2 The relationship between the interfaces and plug-ins in the costmap.](image)

In ROS, each map layer is presented in the form of a plug-in, which can be modified and compiled independently. In Figure 2, Costmap2DROS is an interactive interface provided to the users. LayeredCostmap is also a mechanism for loading various plug-ins. [9] The map information is also stored in the respective plug-ins. Other blocks are also defined as follows:

StaticLayer: This block mainly addresses the static map generated by the mapping algorithm.
ObstacLayer: This block mainly addresses the obstacle information produced by the robot in the moving process.\[10\]

InflationLayer: This mainly addresses the obstacle inflation information on the robot navigation map. Since ObstacleLayer and StaticLayer have their respectively maintained maps, they inherit from CostmapLayer and Costmap2D, where a parent class for storing maps is provided by Costmap2D. Furthermore, several functionalities are offered by CostmapLayer to manipulate the maps. A real map is not maintained by InflationLayer; therefore, it inherits the map from Layer in conjunction with the CostmapLayer.\[11\] Furthermore, a mechanism is provided by Layer to update the Master costmap.

In our proposed method, we place a virtual obstacle in the map at the corner of the physical obstacle. Therefore, a new plug-in that manages map information must be created. We refer to this plug-in as CornerLayer. Using this plug-in, the shape of the virtual obstacle is directly defined, and the virtual obstacle coordinates are calculated. The real map is not maintained by this plug-in; therefore, its operation is similar to InflationLayer, which is directly inherited from Layer.

2.2. SimpleLayer
To facilitate the secondary development of the costmap, a plug-in called SimpleLayer is provided by the ROS platform. This is a simple functional layer that can synchronize a virtual obstacle in the shape of a small dot located 1 m immediately ahead of the robot.\[12\] The plug-in is also directly inherited from the layer. In our proposed method, we utilize SimpleLayer to realize CornerLayer.

Two functions are critical in the costmap update process: updateBounds() and updateCosts(). These exist in each plug-in. The function updateBounds() retrieves the update zone and area of each plug-in sequentially to calculate the total predicted update range of the master costmap. Function updateCosts() writes the updated content of each plug-in to the master costmap upon updating updateBounds().\[13\]

A virtual obstacle \( P(\text{mark}_x, \text{mark}_y) \) with a dot shape is also designed by SimpleLayer and located one meter ahead of the robot. Because the object information is not derived from a static map or sensor, its coordinates and coverage area (shape) information can be directly written into updateBounds(), i.e., it can be updated by the master costmap. The robot coordinates and azimuth information \((\text{origin}_x, \text{origin}_y, \text{origin}_\text{yaw})\) are also obtained from the ROS. Therefore, the coordinate information of the obstacle is

\[
\text{mark}_x = \text{origin}_x + \cos(\text{origin}_\text{yaw}).
\]
\[
\text{mark}_y = \text{origin}_y + \sin(\text{origin}_\text{yaw}).
\]

Since the object is a dot with a single pixel, there is no need to specifically define its shape. The object designed by SimpleLayer is defined as an obstacle; therefore, if the cost of the object is set to LETHAL_OBSTACLE under updateCosts(), it can be presented in the form of an obstacle in the master costmap. In this paper, our objective is to change the turning trajectory of the robot through a virtual obstacle. Therefore, the single-dot model of the obstacle needs to be completely modified.

2.3. Design Of The Virtual Obstacle
To ensure the smoothness of the robot trajectory, we propose a virtual obstacle approximated as a circle, as shown in Figure 3.

![Fig. 3 The design of the virtual obstacle.](image-url)
A regular N-polygon with \( n \) edges on the left is placed in Figure 3 and its central coordinates are \( O(x, y) \). The distance from its center to any vertex coordinates on the graph is \( z \), where \( A(a, b) \) is the coordinate of an arbitrary vertex. By using the number of edges, \( n \), of regular N-polygons, the included angle \( \theta \) corresponding to each edge is obtained as:

\[
\theta = 2 \times \pi / n.
\]  

(3)

Using the central coordinate \( O(x, y) \) of the regular N-polygon and the distance \( z \) from the center coordinates to the coordinates of any vertex on the graph, the coordinate of point \( A \) is obtained by the sine and cosine functions:

\[
a = x + z \times \cos \theta.
\]  

(4)

\[
b = y + z \times \sin \theta.
\]  

(5)

Using the same method, the coordinates of the remaining vertices on the regular N-polygon are then obtained as shown in Figure 4.

In the graph, the label of the current vertex is represented by \( j \). When \( j \) equals 1, as shown in Figure 4 (A), the included angle \( \theta \) is \( 2\pi/n \), signifying that \( 360^\circ \) is equally divided by the angle into \( n \) parts, and the corresponding angle of each part is the included angle corresponding to each edge of the regular \( n \)-polygon. Therefore, the coordinates of the first vertex can be obtained using formulas (4) and (5). When \( j \) equals 2, as shown in Figure 4 (B), the included angle \( \theta \) is twice that of \( 2\pi/n \). In this coordinate system, the coordinates of the second vertex can be obtained by using the sine function and cosine function of the included angle. Following the same line of argument, the \( j \)th vertex is as shown in Figure 4 (C), and the included angle \( \theta \) is \( j \times 2\pi/n \). Similarly, the coordinates of the \( j \)th vertex are obtained by using the sine and cosine functions of the included angle. This process can be realized utilizing a cyclical function, as follows.

Fig. 4 The vertex coordinates of the regular polygons.

![Fig. 4 The vertex coordinates of the regular polygons.](image)

Fig. 5 Regular polygons are calculated using cyclic functions.

![Fig. 5 Regular polygons are calculated using cyclic functions.](image)
In Figure 5, the parameter label in the parameter list is represented by \( i \). In the costmap, more than one identical virtual obstacles are often needed, therefore, when there are multiple pieces of information in the parameter list, regular polygons need to be generated one-by-one.

To conclude, a regular polygon can be produced by using only such parameters as \( [x, y, z, n] \), where \( n \) is the number of edges of the regular polygon. By increasing \( n \), the graph becomes infinitely close to a circle. In this setting, \( z \) represents the distance between the central coordinates and the coordinates of any vertex on the graph, which is the radius of the graph. The value of \( z \) determines the size of the virtual obstacle. In this setting \( (x, y) \) is the central coordinate of a regular polygon, which can be understood as the specific location of a virtual obstacle in the costmap. The method of finding its coordinates is presented in the next section.

We redesign the shape of the obstacle in the SimpleLayer to make it closer to the circle by setting the number of edges to 36 by default. Figure 6 shows the virtual obstacles generated based on different values for the model parameters.

![Fig. 6 Comparison of SimpleLayer obstacles before](A) ![Fig. 6 Comparison of SimpleLayer obstacles before](B)

Figure 6 shows two costmaps in which the purple areas represent the obstacles, the blue area is the inflation zone, and the black object is the robot. Figure 6 (A) shows a small dot directly in front of the robot, which is a virtual obstacle synchronized by using the original SimpleLayer. Figure 6 (B) indicates several purple areas of different sizes, which are virtual obstacles synchronized with the improved SimpleLayer, representing a practical case much more effectively than the dots in the original SimpleLayer.

3. Coordinate Detection Of Inflection Point Of Obstacle

If the turning path of the mobile robot is to be changed, in addition to adding the virtual obstacles to the costmap, another key point is to find the inflection point coordinates of the physical obstacles. Corners are similar to angular points, in practice however detecting them just by relying on corner detection algorithms and/or feature point detection algorithms is not viable. The corner region is one of the most protruding positions in an object with regard to its parts. To detect such a position, we use the convex hull algorithm.

3.1. Inflection Point Detection Based On The Convex Hull Algorithm

The convex hull is a geometric concept that is defined as follows. Suppose \( S \) is an arbitrary subset of the Euclidean space, \( \mathbb{R}^n \), where the convex set containing the smallest \( S \) is called the convex hull of \( S \). In a two-dimensional plane, the smallest convex polygon formed by a subset of a given point set, \( Q \), is called a convex hull polygon. In a convex hull polygon, all the points satisfying \( Q \) are on, or within, the polygon (see, Figure 7 for an example in a two-dimensional plane).[14]

![Fig. 7 A schematic of a convex hull polygon in a two-dimensional plane.](image)
In the static maps, each obstacle can be regarded as a set of multiple pixels. Involving these pixels in the calculations however results in high computational complexity.\cite{15} Because the most protruding pixel of the obstacle only appears on the contour, the contour of the obstacle can be extracted by using a contour detection algorithm. This allows us to store the main pixels that constitute the contour in a matrix. This matrix is then processed using the convex hull algorithm.\cite{16} Finally, the coordinates of the target pixels are obtained, as shown in the convex hull solving process as depicted in Figure 8.

Fig. 8 The process of obtaining the convex hull of a plane figure.

The protruding pixels detected in Figure 8 are the inflection points for which we are looking. To prove the practicability of this method, two obstacles are intercepted from the static map and the above method is used to detect their inflection points. Meanwhile, the traditional Harris corner detection algorithm and the feature point detection algorithm FAST are applied for comparison. The detection results are shown in Figure 9.

Fig. 9 Comparison of the detection results of three algorithms.

The detection results in Figure 9 suggest that in cases where we are facing a figure with many gaps, the location of the inflection point is not only spotted by Harris corner detection algorithms, but also a large number of noninflection point coordinates are detected. These redundant points, however, meet the definition of an angular point by the algorithm, and there is no way to filter them by adjusting the thresholds. The same problem is faced by the FAST algorithm, wherein this case there are more irrelevant points. Ultimately, only the detection results of the convex hull algorithm are completely in conformity with our requirements for inflection points.

For a smooth irregular figure, the Harris algorithm is completely at a loss for the arc region, and there are still a large number of irrelevant points for the FAST algorithm. Finally, only the detection results of the convex hull algorithm are in accord with our requirements. Therefore, compared with the traditional corner detection algorithm and feature point detection algorithm, the convex hull algorithm is better suited for the inflection point detection of independent obstacles in the static maps.

3.2. Research On The Inflection Point Detection Method Applicable To The Map Boundary
The boundary of the map is composed of the wall itself and the obstacles placed against the wall. In this
case, the convex pixels on the image contour should correspond to positions such as the wall corner, while the concave pixels on the image contour are turned into inflection points.\cite{17} Therefore, the convex hull algorithm is no longer suitable for inflection point detection of the map boundary. To detect the concave pixels on the image contour, we first apply the concave shell algorithm, which breaks the irregular figure contour, and then store the coordinate information of each turning point on the broken line.\cite{18} as shown in Figure 10.

![Fig. 10 Comparison of convex hull and concave shell algorithms.](image)

Suppose that Figure 10 is a map boundary, where A1, A2 and A3 are qualified inflection points. If the convex hull algorithm is applied, then the detection results are C1, C2, C3 and C4, which do not include the targeted points. By using the concave shell algorithm instead, the detection results are the points marked on the image. Although the coordinates of the target points are included, there might also be a large number of irrelevant points. To accurately detect the inflection points of the map boundary and eliminate the redundant irrelevant points, we design a novel algorithm based on the convex hull and contour detection algorithms. We refer to this algorithm as TurningPoint-W (near the wall). The steps of the proposed algorithm are as follows.

1) The convex hull polygon of the target image is generated and filled by the convex hull algorithm, which is marked as img1.

2) Using the contour detection algorithm, the contour map of the target image is generated and filled, marked as img2.

3) The difference between the two images is presented by using the function absdiff(), and the result is marked as img3.

4) img3 is the concave part of the boundary of the map, which may be an obstacle placed against the wall or perhaps an irregular wall. Using the method in the previous section for the treatment of img3, a set of points, including the coordinates of inflection points, is obtained. The set of points is marked as $M$, as shown in Figure 11.

![Fig. 11 Inflection point detection of the map boundary.](image)

As shown in Figure 11, point set $M$ contains the target inflection point set, $N$, i.e., (A1, A2, A3), as well as the irrelevant point set, $W$, i.e., (B1, B6). There is an intersection between point set $W$ and the convex hull contour, while point set $N$ exists in the concave region of the image, thus no intersection exists between
N and the convex hull contour. Using the interrelation between point sets, the following mechanism is proposed to eliminate the irrelevant points:

5) Using the contour detection algorithm to calculate img1, point set O is obtained, and the coordinates of each pixel that form the convex hull contour are included in point set C.

6) Point set M and point set O are converted from the matrix to a list format; meanwhile, a new list P is created; set $P = M$;

7) Set point $i \in M$, point $j \in O$, $W$ is an appropriate threshold, $T = [(i_x - j_x)^2 - (i_y - j_y)^2]^{1/2}$ is calculated and if $T < W$, $i$ is deleted from the list $P$, until all points in $M$ are compared with those in $W$;

8) Through the judgment and screening in “7)”; the point set in $P$ includes the inflection points of the map boundary we are looking for.

Figure 12 is a static map without independent obstacles. Using the algorithm of TurningPoint-W to detect its inflection point, the result is shown in Figure 12.

![Fig. 12 Detection of inflection points on static map boundary using TurningPoint-W algorithm.](image)

As shown in Figure 12, the TurningPoint-W algorithm effectively detects the inflection point of the static map boundary. However, to realize the obstacle inflection point detection of the entire static map, a combination of the convex hull algorithm in Section A and the TurningPoint-W algorithm in this section is required. We refer to this combined algorithm as TurningPoint.

3.3. The TurningPoint Algorithm

TurningPoint combines the convex hull and TurningPoint-W algorithms. The precondition of realizing these two algorithms is image contour detection. The image contour consists of multigroup three-dimensional matrices in which each matrix element is a key point coordinate. Contour detection is carried out from outside to inside, and the outer contour is the parent contour of the inner contour. Use [0] as the parent contour index and ["1 - n"] as the child contour index. Because the research object of this paper is the map boundary and independent obstacles, the contour is divided into two layers, and no new
subcontour is established for the independent obstacle. A flowchart of the TurningPoint algorithm is illustrated in Figure 13.

To test the algorithms, several representative images are drawn to compare the detection efficiency of the corner detection algorithms, concave shell algorithm, convex hull algorithm and TurningPoint algorithm. The results are illustrated in Figure 14.

![Corner detection](image1)
![Concave shell](image2)
![Convex hull](image3)
![TurningPoint](image4)

(A)

![Corner detection](image5)
![Concave shell](image6)
![Convex hull](image7)
![TurningPoint](image8)

(B)

![Corner detection](image9)
![Concave shell](image10)
![Convex hull](image11)
![TurningPoint](image12)

(C)

Fig. 14 Comparison of the inflection point detection for different algorithms.

As shown in Figure 14, the corner detection algorithm is not sensitive to arcs, and the inflection point cannot be detected in the face of smooth and irregular objects. The convex hull algorithm also fails to correctly recognize the boundary inflection points. This means that in practical applications, such as obstacles placed against the wall and/or irregular walls, their inflection points cannot be detected by the convex hull algorithm. The concave shell algorithm, however, effectively detects every turning point on the figure contour. However, some points do not belong to the inflection point; thus, their inclusion may reduce the robot’s working area. Among the investigated algorithms in Figure 14, the TurningPoint algorithm accurately detects all the inflection points without returning any irrelevant points. It is therefore concluded from the above tests that compared to the corner detection algorithm, concave shell and convex hull algorithms, the TurningPoint algorithm is more effective for obstacle inflection point detection.

4. Turning Simulation And Experiment Of Mobile Robot
By adding the TurningPoint algorithm to the improved SimpleLayer, the new plug-in CornerLayer is capable of automatic detection of the corner coordinates of the static map and of adding circular virtual
obstacles to the corner. To test the actual effect of the virtual obstacles, in this section, we use simulations and experiments.

4.1. Simulation Verification Of CornerLayer

To verify the costmap navigation effect following the use of the CornerLayer plug-in, we constructed the scene demonstrated in Figure 15 using the Gazebo physical simulation platform.

This scene is composed of several small squares. We then use Python to compile a simple autonomous navigation program that allows robots to navigate freely in the scenes. Meanwhile, a circle of red sensors is considered around the simulated robot. During collisions between the robot and an obstacle, a set of data is returned by the robot, including the coordinates of the collision and topic sequence. According to the topic publication frequency, the collision time can be calculated, while statistics can tell the cumulative collision times within the unit time. The cumulative error of the robot is related not only to its hardware but also to the density of the obstacles. In this paper, by introducing appropriate noise to the robot sensor data, the error of the robot under different obstacle densities is simulated (see Figure 16).

Fig. 15 Gazebo simulated scene.

Fig. 16 Robot cumulative error for different obstacle densities.
Figure 16 shows the error of the robot’s x-coordinate, y-coordinate and direction angle Ø under different obstacle densities. It can be seen that the sparser the obstacle density, the larger the robot’s cumulative error. It is further seen that the robot’s cumulative error increases with time. Therefore, we adopt a moderate obstacle density in this simulation. Figure 17 presents the original costmap corresponding to the Gazebo scene and the costmap after adding virtual obstacles at the corner.

We further introduce noise to the robot sensor and let it navigate autonomously in the scene. The experiment time for each group is 5 minutes, and the radius of the virtual obstacle is gradually increased from 0. The experimental collision data in each group are shown in Figure 18.

![Fig. 17 Comparison of the two costmaps.](image1)

![Fig. 18 The collision frequency chart.](image2)

| Table 2 Statistical Table Of Average Collision Times |
|---------------------------------------------|-----|
| 0 cm Small virtual obstacle radius:          | 7   |
| 5 cm Effective prevention and control        | 7.4 |
| 10 cm Large virtual obstacle radius:         | 6.2 |
| 15 cm Ineffective prevention and control     | 3   |
| 30 cm                                        | 0   |
### Table 3 Occupied Rate Of Robot’s Effective Working Area

| Virtual/Inflation radius | Increase the Inflation radius | Add virtual obstacles |
|--------------------------|-------------------------------|-----------------------|
| 0 cm                     | 0%                            | 0%                    |
| 5 cm                     | 2.93%                         | 0.16%                 |
| 10 cm                    | 6.23%                         | 0.68%                 |
| 15 cm                    | 9.35%                         | 1.48%                 |
| 30 cm                    | 19.81%                        | 5.92%                 |

To eliminate accidental factors, five experiments are conducted on the virtual obstacle of each size. Table 2 shows how many collisions occur between the robots and the virtual obstacles of different radii. To avoid repeated feedback from the sensors, feedback values are screened only near the same coordinates within the adjacent time. Through a comparison of the experiments, it can be seen that although the cumulative error of the robot increases with time, the number of collisions between the robot and the obstacle is significantly reduced due to the gradual increase of the virtual obstacle radius. In Table 3, the different occupancy rates of the robot’s effective working area are recorded in the current simulation environment for the virtual obstacles and inflation area with the same radius. According to the statistics, we can see that the robot’s working area occupied by virtual obstacles is always far smaller than that occupied by the inflation area. Therefore, we conclude that CornerLayer not only effectively prevents the robot from making a turn from collisions but also reduces the occupation of its working area as little as possible.

### 4.2. Experimental Verification Of Cornerlayer

A Raspberry Pi 3B is adopted for this experiment as the host computer of the robot. This microcomputer integrates an ARM-architecture 64-bit 1.2-GHz quad-core processor plus 1 GB RAM, 16 GB ROM, WiFi, Bluetooth and several other communication interfaces. It is mainly in charge of data processing and information transmission. The lower part of the robot is a chassis-driven circuit board that integrates a motor-driven interface, radar interface, power input interface, IMU pose module, etc. The lower machine is mainly responsible for controlling the bottom sensors and transmitting data. Moreover, a SLAM RPLIDAR A1 2D laser radar, two motors containing encoders, and power supply are included in the robot. Figures 19 and 20 show pictures of the robot and the experimental scenes, respectively.
In this experiment, Ubuntu16.04 and ROS Kinetic are loaded on a Raspberry Pi 3B through the PC terminal. Instructions are transferred by the PC to the Raspberry Pi via WiFi. Similarly, real-time data are transmitted by the Raspberry Pi to the PC via WiFi. We use the Next rviz visualization tool to monitor the data in real time. The costmaps before and after the CurveLayer plug-in are added and used to navigate. The results are shown in Figure 21.

![Fig. 21 Comparison of the navigation trajectories of two costmaps.](image)

The blue objects represent the robots, red lines represent the planning paths, and the purple area indicates the physical and virtual obstacles. Before the virtual obstacles are added, the planning path is too close to the inflation zone on the fringe of the obstacles when turning. If the cumulative error is large, then the risk of the robot bumping into the obstacles is also high. By adding virtual obstacles to CornerLayer, the corridor area is not excessively occupied. If the path planning algorithm plans along with the virtual obstacles, then the turning path is rendered more smoothly and stays farther away from the physical obstacles. This effectively reduces the robot’s collision possibility. The two costmaps corresponding to the real moving trajectories of the robot are depicted in Figure 22.

![Fig. 22 Comparison of the real moving trajectories of the robot.](image)

(A) No virtual obstacles
Fig. 22 Comparison of real trajectories of the robot based on two costmaps.

It is observed in Figure 22 that for navigation via the original costmap, the robot is traveling close to the obstacles at turning and may slightly run into the obstacles. Using the costmap for navigation after adding the virtual obstacles with CornerLayer, the robot stays away from the physical obstacles while turning, and the traveling process becomes safe. In commercial applications, product designers can adjust the radius of the virtual obstacles in line with the actual situation of the products and customer demands to change the turning path of the robot and avoid collisions.

5. Conclusions And Outlook
To prevent collisions between a robot and obstacles while turning, we proposed adding virtual obstacles to the costmap to modify the turning path of the robot. The main contributions of this paper are as follows:

1) The obstacle shape of SimpleLayer was redesigned so that it can effectively affect the planning path of the robot.
2) With the contour detection algorithm and convex hull algorithm acting as the foundation, the TurningPoint algorithm was proposed, which is capable of effectively identifying the position of the inflection point of the obstacles in the static map.
3) By integrating the improved SimpleLayer and TurningPoint algorithm, CornerLayer was formed. This new plug-in can automatically detect the inflection point coordinates of the static map and further synchronize the virtual obstacles in the corresponding position in the costmap to adjust the turning path of the robot.

This research can availably solve the problem that A*, Dijkstra, and BFS (Breadth, First Search) and other common path planning algorithms stick to obstacles on the turning point, and reduce the risk of collision effectively when the robot turns in practical application. Above problems are solved by introducing "virtual obstacles". At present, in the field of mobile robot navigation, problems such as avoiding the robot from entering a specific area, allowing the robot to move within a specific range, and allowing the robot to maintain specific behavior characteristics can also be solved by using virtual obstacles. It is hoped that a new way to solve the similar problems can be provided in this research.

Acknowledgments
This work was supported by the Guangxi Department of Science and Technology Department (Project number: AE30100009).

References
[1] D. V. Lu, “Contextualized robot navigation,” Ph.D. thesis, Dept. of Computer Science, Washington University, Washington, USA, 2014.
[2] D. V. Lu, D. Hershberger, and W. D. Smart, Layered costmaps for context-sensitive navigation, *IEEE/RSJ International Conference on Intelligent Robots and Systems*, Chicago, IL, USA, Sep, 14-18, 2014.
[3] D. Claes, and K. Tuyls, "Multi robot collision avoidance in a shared workspace." *Autonomous Robots*, vol. 42, no. 8, pp. 1749-1770, Apr. 2018.
[4] J. C. Mohanta, and A. Keshari, “A knowledge based fuzzy-probabilistic roadmap method for mobile robot navigation.” *Applied Soft Computing*, vol. 79, pp. 391-409, Jun. 2019.
[5] B. K. Patle, D. R. K. Parhi, A. Jagadeesh, and S. K. Kashyap, “Application of probability to enhance the performance of fuzzy based mobile robot navigation.” *Applied Soft Computing*, vol. 75, pp. 265-283, Feb. 2019.

[6] A. Rudenko, T. P. Kucner, C. S. Swaminathan, R. T. Chadalavada, K. O. Arras, and A. J. Lilienthal, THÖR: Human-Robot Navigation Data Collection and Accurate Motion Trajectories Dataset. *IEEE Robotics and Automation Letters*, vol. 5, no. 2, pp. 676-682, Apr 2020.

[7] G. Kahn, A. Villaflor, B. Ding, P. Abbeel, and S. Levine, Self-supervised deep reinforcement learning with generalized computation graphs for robot navigation. *IEEE International Conference on Robotics and Automation (ICRA)*. Brisbane, QLD, Australia, May, 21-25, 2018.

[8] Y. Cheng, and G. Y. Wang, Mobile robot navigation based on lidar, *Chinese Control And Decision Conference (CCDC)*. Shenyang, China, Jun, 9-11, 2018.

[9] T. Sotiropoulos, H. Waeselynck, J. Guiochet, and F. Ingrand, Can robot navigation bugs be found in simulation? an exploratory study. *IEEE International Conference on Software Quality, Reliability and Security (QRS)*. Prague, Czech Republic, Jul, 25-29, 2017.

[10] B. Hoover, S. Yaw, and R. Middleton, CostMAP: an open-source software package for developing cost surfaces using a multi-scale search kernel. *International Journal of Geographical Information Science*, vol. 34, no. 3, pp. 520-538, Oct, 2019.

[11] D. Claes, D. Hennes, and K. Tuyls, Towards human-safe navigation with pro-active collision avoidance in a shared workspace, *Workshop On-Line Decision-Making Multi-Robot Coordination (IROS)*, pp. 1-8, Oct, 2015.

[12] P. Regier, S. Oßwald, P. Karkowski, and M. Bennewitz, Foresighted navigation through cluttered environments, *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Daejeon, South Korea, Oct, 9-14, 2016.

[13] S. Mentasti, and M. Matteucci, Multi-layer occupancy grid mapping for autonomous vehicles navigation, *AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. Torino, Italy, Italy, Jul, 2-4, 2019.

[14] R. L. Graham, and F. F. Yao, "Finding the convex hull of a simple polygon," *Journal of Algorithms*, vol. 4, no. 4, pp. 324-331, Dec. 1983.

[15] D. T. Lee, On finding the convex hull of a simple polygon, *International journal of computer & information sciences*, vol. 12, no. 2, pp. 87-98, Apr. 1983.

[16] N. L. F. Garcia, L. D. M. Martinez, A. C. Poyato, F. J. M. Cuevas, and R. M. Carnicer, Unsupervised generation of polygonal approximations based on the convex hull, *Pattern Recognition Letters*, vol. 135, pp. 138-145, Jul, 2020.

[17] S. Suzuki, and K. Abe, "Topological structural analysis of digitized binary images by border following," *Computer Vision Graphics & Image Processing*, vol. 30, no. 1, pp. 32-46, Apr. 1985.

[18] J. Seo, S. Chae, J. Shim, D. Kim, C. Cheong, and T. D. Han, Fast contour-tracing algorithm based on a pixel-following method for image sensors, *Sensors*, vol. 16, no. 3, pp. 353-379, Mar, 2016.

[19] T. Matić, I. Aleksić, Ž. Hocenski, and D. Kraus, Real-time biscuit tile image segmentation method based on edge detection, *ISA transactions*, vol. 76, pp. 246-254, May, 2018.