# Implementation notes about Tangent Bug, Trajectory Tracking, Potential Fields and Wave Front

Arthur H. D. Nunes[1], and Álvaro R. Araújo[1]

***Abstract*— In this work we implemented four robot motion planning algorithms to a differential drive robot: (i) Tangent Bug; (ii) Trajectory Tracking; (iii) Potential Fields; (iv) Wave Front. The implementations were made in StageROS ROS simulator with a ROS in Docker architecture. We described the implementation guidelines and present the results.**

***Index Terms*— Implementantion notes, ROS in Docker, tangent bug, potential fields, wave front.**

## I. INTRODUCTION

In this work we implemented four robot motion planning algorithms:

- Tangent Bug;
- Trajectory Tracking;
- Potential Fields;
- Wave Front.

We considered a differential drive robot that can move in a two-dimensional workspace with obstacles. The goal of all algorithms is to guide the robot from a initial configuration to an goal configuration. Furthermore, in trajectory tracking, the goal configuration may vary with time to generate a closed curve. The implementations were made in StageROS ROS simulator with a ROS in Docker architecture.

In the following section, II, we describe the simulation architecture and the four motion planning problems definitions. Then, in Section III we describe the implementation guidelines and strategies that we used to achieve the completion of them. Next, in Section IV we show the simulation results. Finally we conclude in Section V.

## II. SETUP

### A. Simulation Architecture

The simulations presented in this work was based in Robot Operating System (ROS). Inside ROS, we used StageROS simulator which allows us to control a differential drive robot in a planar space with obstacles. This simulator is good for multiple robots, beginners at ROS and for simpler simulation scenarios. As we are not interested in complex physic interactions, irregular workspaces, non-euclidian spaces or tri-dimensional simulations, our case fits in the third selection, where we want a basic planar simulator to test the four mentioned motion planning algorithms.

Furthermore, we also used Docker Containers as part of the simulation architecture. Docker is a software that performs Operating System level virtualization to deliver software in packages. The intention to use ROS inside Docker is to minimize the effort of setting up the simulation environment by creating a container model, called container image. By creating our custom ROS image, one can assure that the software will incorporate its dependencies in it and therefore installations and configuration are no longer required, making a *ready-to-use* environment.

In order to access GUI provided by the container, we use a noVNC approach that is already configured inside the container image. With the noVNC running, just opening the web browse is enough to access the container GUI. Figure 1 shows the basic background for a ROS in docker simulation. Another key point to mention is that Docker containers can also be used to deploy applications, using the same container images and therefore the same software version.

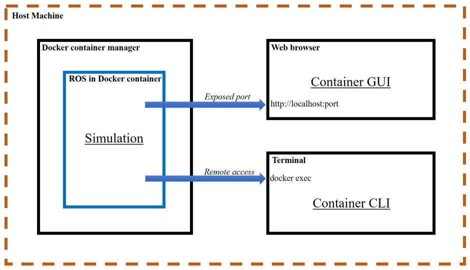

Fig. 1. Diagram to show ROS in Docker simulation architecture.

### B. Problem Definitions

*1) Tangent Bug:* The Tangent Bug is an algorithm to drive an robot from an initial configuration to one target configuration while avoiding obstacles, [1]. In this work, we will address the problem of guiding one differential drive robot from one two-dimensional position to another. In order to do that, we need an range sensor and an position sensor/estimator.

The implemented algorithm here can be split in three different states:

- First, the robot is in *motion-to-goal* state, where it blindly follows the target. When it encounter an obstacle border between itself and the target, it switches to the next state. An obstacle border is defined by a sequence of "continuous" measured points, where the first and the last point are discontinuities. Those discontinuities points in the range sensor measurement are labeled as $O_i$.
- Then, the next state that it switches to we called *motion-to-oi*. In this second state the robot follows an chosen $O_i$ as it was following the target. When the robot finds a local minimum in the obstacle border that is being followed it switches to the third and final state.

- Finally, the *follow-wall* state is described by a movement that intends to follow the obstacle border until it can safely decrease again the distance to the target and therefore switch back to the first state.

*2) Trajectory Tracking:* Define the Trajectory Tracking problem addressed in this work as follows:

Make a differential drive robot follow a closed curve. This curve is described by parametric equations that corresponds to a virtual target varying in time. Hence, the robot must converge to and follow this target.

*3) Potential Fields:* The Potential Fields navigation method apply to a rich class of robots and produce a great variety of paths, aplying to a general class of configuration spaces, including those that are multidimensional and non-Euclidean, [6].

The Potential Fields method is based in a potential function, that is a differentiable real-valued function. The value of a potential function can be viewed as energy and hence the gradient of the potential is force. The gradient is a vector, which points in the direction that locally maximally increases the function. A gradient vector field, as its name suggests, assigns the gradient of some function to each point.

The potential function approach directs a robot as if it were a particle moving in a gradient vector field. Gradients can be intuitively viewed as forces acting on a positively charged particle, the robot, which is attracted to the negatively charged point, the goal. In this case, obstacles also have a positive charge which forms a repulsive force directing the robot away from obstacles. The combination of repulsive and attractive forces hopefully directs the robot from the start location to the goal location while avoiding obstacles. So potential functions can be viewed as a landscape where the robots move from a high-value state to a low-value state. The robot follows a path downhill by following the negated gradient of the potential function, and following such a path is called "gradient descent".

*4) Wave Front:* The Wave Front navigation method can only be implemented in spaces that are represented as grids, but affords a simple solution to the local minimum problem in robot motion planning. Also for this method it is necessary to know the map that will be explored beforehand, [1].

Consider a two-dimensional space: initially, the planner starts with the standard binary grid of zeros corresponding to free space and ones to obstacles. The planner also knows the pixel locations of the starting robot position and goal position. The goal pixel is labeled with a two. In the first step, all zero-valued pixels neighboring the goal are labeled with a three. Next, all zero-valued pixels adjacent to threes are labeled with four. This procedure essentially grows a wave front from the goal where at each iteration, all pixels on the wave front have the same path length, measured with respect to the grid, to the goal. This procedure terminates when the wave front reaches the pixel that contains the robot starting location.

## III. METHODOLOGY

Here we will address the problems in a different order as they appeared in the previous Section, I. The new order is meant to construct a logic that uses one solution as part of the next ones and gradually evolve in problem complexity.

### A. Trajectory Tracking

The first problem addressed is the Trajectory Tracking. Here, we will split the solution in two parts: (i) Define and create the target; (ii) Define and implement the low-level controller to follow the given target.

The target can be a function of time that returns a two-dimensional position. The chosen function for this work describes curves named as Rose Curves, [2]. The Rose Curves equation is shown in eq. (1).

$$q_{target}(t) = \begin{bmatrix} A\cos(k\omega t)\cos(\omega t) \\ A\cos(k\omega t)\sin(\omega t) \end{bmatrix}, \tag{1}$$

where $A, k, \omega$ are parameters passed by the user at runtime.

$A$ describes the amplitude of the curve. $k$ is the one that actually defines the rose-shape of the curve, where $k = 0$ is the trivial shape not addressed in this work. $\omega$ is to adjust the target time variation in order to accomplish target speeds small enough so the robot can follow. It is important to emphasise that $\omega$ do not determine the target speed itself, but it helps to adjust it. The total speed will be a function of all three parameters.

Figure 2 shows four examples of Rose Curves with same $A$, one period and for different $k$ values.

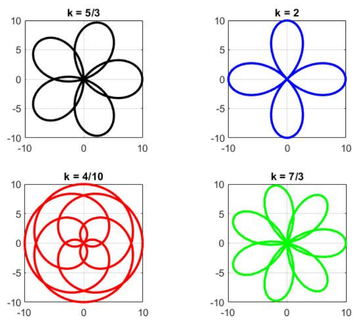

Fig. 2. Rose Curves examples

As we are going to show next, for the low-level controller we also need the target speed, which is given by the first order derivative of the target. Even though it can be analytically defined, we are going to use the discrete bilateral definition shown in eq. (2)

$$\dot{q}_{target} = \frac{q_{target}(t + dt) - q_{target}(t - dt)}{2dt}, \tag{2}$$

where $dt$ is the differential time in analytics formulations or the time increment in the simulation matters.

Both (1) and (2) can be easily computed during the simulation.

Next, we need to define the controller that will guide the differential drive robot to $q_{target}$. In other words, we need to find control inputs that will make $q(t) \to q_{target}(t)$ as $t \to \infty$. To do that we will use a non-linear control technique named feedback linearization, [4]

The vector $q(t) = [x_1(t)\ x_2(t)]^T$ stands for the two-dimensional position of the controlled point somewhere in the robot's rigid body. Usually the controlled point is the geometric center of the robot. If we pick a circular shaped body and choose its center as the controlled point we will inherit a big disadvantage. Due the non-holomic constraint of the differential robot model, it is impossible to make its center follow and arbitrary velocity vector that is not aligned with the robot's orientation.

Therefore, we aim to control another point that is located $d$ units in front of the robot's center. We will call this point as *controlled point*. Referring to that point, in eq. 3 we show the state-space equations for $q(t)$.

$$\dot{\begin{bmatrix} x_1 \\ x_2 \end{bmatrix}} = \begin{bmatrix} \cos(\theta) & -d\sin(\theta) \\ \sin(\theta) & d\cos(\theta) \end{bmatrix} \begin{bmatrix} v \\ \omega \end{bmatrix}, \tag{3}$$

Where $\theta$ is another state related to robot's orientation and $v, \omega$ are the control inputs relative to linear and angular velocity.

The intention is find input the signals $[v\ \omega]^T$ that make $\dot{q}$ follows an auxiliary control law $F \equiv [F_{x1}\ F_{x2}]^T$. It is easily shown that (4) suits this case.

$$\begin{bmatrix} v \\ \omega \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -d\sin(\theta) \\ \sin(\theta) & d\cos(\theta) \end{bmatrix}^{-1} \begin{bmatrix} F_{x1} \\ F_{x2} \end{bmatrix}, \tag{4}$$

Since we are not controlled the center, so $d > 0$, the matrix is non-singular. Although, one need to take into account that as $d$ decreases the matrix becomes more bad conditioned. So it is not a good choice to control a point too much close to its center. To avoid that, we will define the controlled-point at the robot border.

Finally, the auxiliary control law can be determined as an proporcional law with feedforward action, as expressed in (5).

$$F = k_e(q_{target}(t) - q(t)) + \dot{q}_{target}(t),\ k_e > 0 \tag{5}$$

Both $F$ and $[v\ \omega]^T$ can be easily computed. The implementation proposal for this problem is completed.

## B. Potential Fields

The simplest potential function is the attractive/repulsive potential. The intuition behind the attractive/repulsive potential is straightforward: the goal attracts the robot while the obstacles repel it. The sum of these effects draws the robot to the goal while deflecting it from obstacles. The potential function can be constructed as the sum of attractive and repulsive potentials:

$$U(q) = U_{att}(q) + U_{rep}(q) \tag{6}$$

To calculate the attractive potential it is necessary to know the robot position in relation to the goal, and for the repulsive potential it is necessary to have information about the nearest obstacle point to the robot. Therefore, it is needed a proximity sensor installed in the robot, also location data about the robot and the goal.

The simplest choice for $U_{att}$ is the conic potential, since we need a function that monotonically increases with the robot distance in relation to the goal. That function would be $U_{att}(q) = \sigma d(q)$, with $\sigma$ being a parameter used to scale the effect of the attractive potential and $d(q)$ a function that relates the distance between the robot and the goal configuration. But this kind of implementation may deal with chattering problems, because there is a discontinuity in the gradient at the origin.

For that reason we need a function that is continuously differentiable, and also monotonically increases with the robot distance in relation to the goal. In that scenario the simplest function is one that grows quadratically with the distance such as:

$$U_{att}(q) = \sigma d^2(q) \tag{7}$$

It is also important to limit that function, since if the distance between the robot and the goal is too high there's a chance the repulsive potential cannot overcome it to make the robot change its direction. For that reason, the chosen function was equation 8, with the $d_{thres}$ value limiting the potential:

$$U_{att}(q) = \begin{cases} \frac{1}{2}\sigma d^2(q),\ d(q) <= d_{thres}, \\ d_{thres}\sigma d(q) - \frac{1}{2}\sigma d^2(q),\ d(q) > d_{thres} \end{cases} \tag{8}$$

In the case of the repulsive potential we desire that the closer the robot is to an obstacle, the stronger the repulsive force should be. Therefore, the repulsive potential is usually defined in terms of the distance to the closest obstacle. Labeling this distance as $D(q)$, we can make use of an equation such as:

$$U_{rep}(q) = \begin{cases} \frac{1}{2}\eta(\frac{1}{D(q)} - \frac{1}{D_{thres}})^2,\ D(q) <= D_{thres}, \\ 0,\ D(q) > D_{thres} \end{cases} \tag{9}$$

where $D_{thres}$ allows the robot to ignore obstacles sufficiently far away from it, and $\eta$ is a parameter used to scale the effect of the repulsive potential.

To find the closest obstacle point to the robot we need to perform an exhaustive search in the range sensor array and find the measurement with minimum distance. For all measured points, we can convert the data measured in relation to the robot frame to the inertial frame according to a homogeneous transformation, [6]. In order to do that, consider the relation expressed in (10) and the following: (i) $[x_{obst}\ y_{obst}]^T$ is the obstacle point position in the inertial frame; (ii) $[x\ y\ \theta]^T$ is the robot two-dimensional pose; (iii) $s, \theta_s$ is the sensor point measurement that describes a point at a distance $s$ at an angle $\theta_s$ in the robots frame.

$$\begin{bmatrix} x_{obst} \\ y_{obst} \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & x \\ \sin(\theta) & \cos(\theta) & y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s\cos(\theta_s) \\ s\sin(\theta_s) \\ 1 \end{bmatrix} \tag{10}$$

Even though there is no direct need to apply this homogeneous transformation to calculate the gradient of the potential function, this step facilitates the computation of the gradient and reduces the computational cost of the method.

With those considerations made, the gradients can be expressed as in equations 11 and 12, with $q_{diff}$ being equal to $(q - q_{obst})$:

$$F_{att}(q) = -\nabla U_{att}(q)$$
$$F_{att}(q) = \begin{cases} -c(q - q_{goal}), \ d(q) \leq d_{thres} \\ -\frac{d_{thres}c(q-qgoal)}{d(q)}, \ d(q) > d_{thres} \end{cases} \tag{11}$$

$$F_{rep}(q) = -\nabla U_{rep}(q)$$
$$F_{rep}(q) = \begin{cases} \eta(\frac{1}{D(q)} - \frac{1}{D_{thres}})\frac{1}{D(q)^2}q_{diff}, D(q) \leq D_{thres} \\ \vec{0}, D(q) > D_{thres} \end{cases} \tag{12}$$

Finally, we used $F = F_{att} + F_{rep}$ as the auxiliary control input for equation (4) in the feedback linearization from the previous problem.

## C. *Tangent Bug*

The only change between the presented problems order and the order in this section is the Tangent Bug problem that was shifted by two positions. The reason for that is because the full Tangent Bug solution also incorporates both the feedback linearization from the Trajectory Tracking problem and the Potential Fields.

First, we will define the auxiliary control law for the feedback linearization in equation (4) for each state. Last, we will define how we compute the transitions between those states.

For *motion-to-goal* and *motion-to-oi* states we use an attractive field to drive the robot to the current goal added with an repulsive field that acts very close to the obstacle to prevent the rigid body of the robot from colliding with an obstacle border while doing those movements, those fields use the same formulation as in the previous problem.

For the *follow-wall* state we use an artificial vector field approach, [5]. In order to do that we need to find the vector $D$ that starts at the closest obstacle point to the robot and ends at the robots position. With the vector $D$ we can find the vector field for wall-following as described in equations (13).

$$E = D - \epsilon\frac{D}{\|D\|},$$

$$D_H = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} D,$$

$$G(E) = -\tfrac{2}{\pi}\arctan(k_G\|E\|), \ k_G > 0, \tag{13}$$

$$H(E) = \pm\sqrt{1 - G(E)^2},$$

$$F = G(E)\frac{E}{\|E\|} + H(E)\frac{D_H}{\|D\|}$$

For the first transition, from *motion-to-goal* to *motion-to-oi* we need to compute the continuity intervals. The classic continuous function definition rely on the lateral function limits being equal. Although, this is not applicable to discrete functions as the case of the range sensor array. In this case, we first assign "infinity" to each value that is equal to the maximum sensor range. Infinity is an abstraction that can be created as an object with defined operations in some programming languages. Other languages already have it. In every case, it can be substituted by a high value much bigger than the maximum range sensor. Therefore, each sequential measurements interval that doesn't contain "infinity" can be assigned as a continuity interval and just storing the first and last value is enough. Finally, with the continuity intervals we need to determine if there is at least one of them that intersects the segment between the robot and the target in order to activate the transition.

To determine if the continuity interval intersects the segment between the robot and the goal we need to first define two line segments. The first line segment is defined by the start and the end of an continuity interval, denoted as $O_i, O_{i+1}$. The second is defined by the target position, $q_{target}$, and a point inside the sensor range that is furthest from the target, $q_{aux}$, ie. a point in the line that conects robot-goal that are sensor max units behind the robot. Figure 3 shows an example where there is intersection. One can see the segment $O_i, O_{i+1}$ in orange and the segment $q_{target}, q_{aux}$ in green.

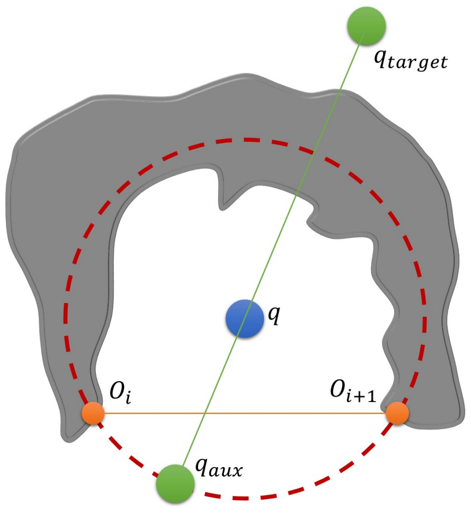

Fig. 3. Segment intersection example to check if there is an obstacle between robot and target.

Let $orient(p, q, r)$ be a function defined by (14). The function returns 1 if points $p, q, r$ are arranged clockwise in the plane, -1 if they are arranged counterclockwise and 0 if they are collinear.

$$sign(x) = \begin{cases} 1, \ x > 0, \\ -1, \ x < 0, \\ 0, \ x = 0 \end{cases} \tag{14}$$

$$orient(p, q, r) = sign(\ (q_y - p_y)(r_x - q_x) \\ -(q_x - p_x)(r_y - q_y)\ )$$

If the segments $O_i, O_{i+1}$ and $q_{target}, q_{aux}$ are not collinear, they will intersect if and only if condition (15) is true, [3].

$$orient(O_i, O_{i+1}, q_{target}) \neq orient(O_i, O_{i+1}, q_{aux}) \\ \text{AND} \\ orient(q_{target}, q_{aux}, O_i) \neq orient(q_{target}, q_{aux}, O_{i+1}) \tag{15}$$

When switched to *motion-to-oi* state, we need to decide which $O_i$ to follow. To do that we use the Manhatan norm heuristic, ie. we choose the $O_i$ that minimizes $d_{heuristic} = d(q, O_i) + d(O_i, q_{target})$. In each iteration we compare if the current $d_{heuristic}$ with the previous one. If the current is not smaller than the previous one, the robot has found an local minimum an it is time to switch to *follow-wall*. At this point we use the same orientation idea given by (14) but using the robot position, the followed $O_i$ and the closest obstacle point as the three-point input to determine the sign of $H(E)$ that will be used in equation (13). We also store the minimum $d(O_i, q_{target})$ found during this progress and call that value $d_{followed}$.

During *follow-wall* we compute a candidate for the stored $d_{followed}$. When we found a value that is appropriete to overwrite $d_{followed}$, ie. when we found a new $d(O_i, q_{target})$ smaller than the stored $d_{followed}$ is time to switch back to the first state.

When the robot reaches the target the program restarts and asks for a new target. When there is not solution, the robot remains in *follow-wall* state. By checking if the robot reached the same point that it started to *follow-wall*, the program can determine that the solution doesn't exist and also restart.

The implementation proposal for this problem is completed.

### D. *Wave Front*

For the Wave Front algorithm the first step is to analyze the map the robot is going to explore. After that, the next step is divide it in a grid, and finally assign the values to each grid cell using the rule described in II-B.4.

Differently than the common approach described in II-B.4, we assigned "infinity" instead of 1 to the pixels that contains obstacles. By doing that, there is no chance to the planner accidentally make the robot go from a pixel 2 (the goal) to a pixel 1. Also, we didn't stop the algorithm when it reaches the robot position. Instead, we used a recursive approach that starts at the goal and ends at the picture borders or obstacles. By making that, all the existing paths to the target, given the resolution will be found. Therefore, we can assure that if the robot is in a pixel with cost:

- 2: It reached the goal. End the program;
- 0: The planner didin't reach this pixel, ie. there is no path to the goal. End the program;
- $\infty$: It is in a pixel with assigned to an obstacle. End the program;
- $2 < X < \infty$: It should move to any neighbour pixel with value $X - 1$. Go to the next iteration.

The planner determines a path via gradient descent on the grid starting from the robot position. Actually we simply computed the auxiliary control law for feedback linearization in (4) as being: $F = [\pm k\ 0]^T$ OR $F = [0\ \pm k]^T, k > 0$, depending of the next pixel direction.

Assuming that the value of the start pixel is 15, as in Figure 4, the next pixel in the path is any neighboring pixel whose value is 14. Essentially, the planner determines the path one pixel at a time, and if there's more than one choice, the algorithm simply picks one of the available choices. The limits of the free space (and therefore the discretization) and continuity of the distance function ensure that construction of the Wave Front guarantees that there will always be a neighboring pixel in the grid whose value is one less than that of the current pixel and that this procedure forms a path in the grid to the goal.

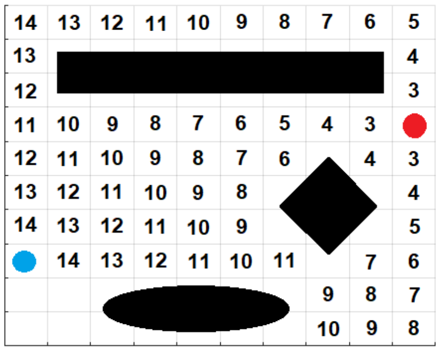

Fig. 4. Wave Front planner pixel assignment value example.

The Wave Front planner essentially forms a potential function on the grid which has one local minimum and thus is resolution complete. The planner also determines the shortest path, but at the cost of coming dangerously close to obstacles. But the true major drawback of this method is that the planner has to search the entire space to find a path to the goal.

Another implementation note to mention is that to avoid colliding or moving too close to the obstacles, we also assigned infinity to those pixels neighbours to obstacle pixels right before executing the planning. This comes into a cost of making the solution may not be found if the discretization doesn't allow at lease three pixels between obstacles.

## IV. RESULTS

### A. *Trajectory Tracking*

The simulations for the Trajectory Tracking problem showed visual convergence and the goal curve was drawn by the robot's path. One example can be seen in Figure 5 for $A = 15$, $\omega = 0.02$ and $k = 5/3$.

For big values of $A$ and $\omega$ the robot could not follow the target due its speed limit imposed by the simulator. It was also observed that for big initial position errors, the robot trajectory can have chattering until it reaches close enough to the target and start to follow a clean trajectory.

### B. *Potential Fields*

The simulations for the Potential Fields method successfully navigated the robot to the desired location when

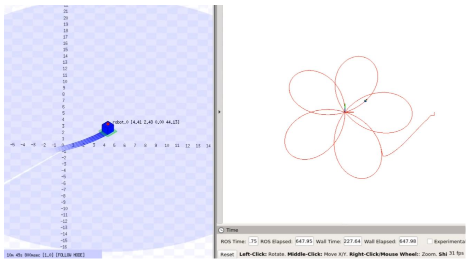

Fig. 5. Trajectory Tracking ROS simulation.

there was no local minimum problem in the ambient being explored by the robot. The attractive potential of the goal location combined with the repulsive potential of the obstacles constructed smooth trajectories during execution time in the simulations.

One example is shown in Figure 6, the robot avoided the obstacles maintaining a safe distance while approaching the target location continuously.

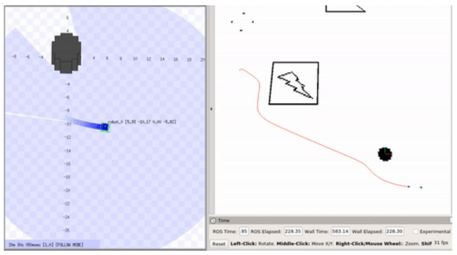

Fig. 6. Potential Fields ROS simulation.

### C. Tangent Bug

The simulations for the Tangent Bug problem successfully guided the robot to the target when there was solutions. In the case that the target was out of $\mathcal{W}_{free}$, the program advertised the impossibility to reach the target.

One example of reaching the target is shown in Figure 7 where one can see the three states working. First the robot follows the target in a straight line until it starts to sense the wall. When it does, it follows one $O_i$ in the lower part of the wall and keep seeking until it determines that it must follow the obstacle. After following the triangle-like border and the lower part the robot finds its way back to the target and follows until it reaches it.

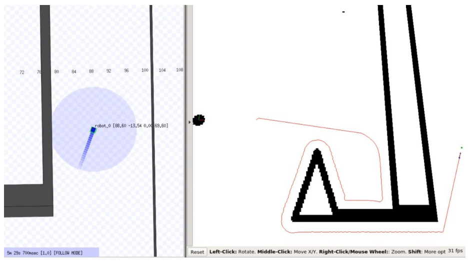

Fig. 7. Tangent Bug ROS simulation.

### D. Wave Front

The simulations for the Wave Front method successfully navigated the robot to the goal while avoiding the obstacles. One exmaple can be seen in Figure 8, it is possible to note the squares in green color which are the grid cells created by the Wave Front algorithm in order to create a path between the robot and the goal.

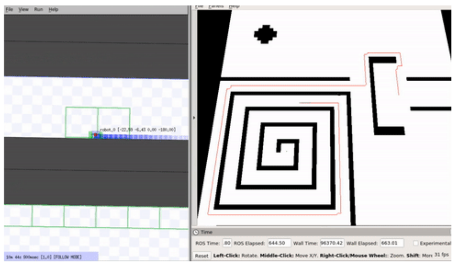

Fig. 8. Wave Front ROS simulation.

## V. CONCLUSIONS

In conclusion, for the Trajectory Tracking implementation, the parameters $A, k$ and $\omega$ should be chosen carefully so that robot's maximum speed is bigger than the maximum target speed, denoted as $\max_{t\in\mathcal{R}} \dot{q}_{target}(t)$. The control strategy developed in equations (4) and (5) doesn't take into account the limited speed of the robot. That means that $F$ can arbitrary increase and the robot may not be able to follow it. By choosing $A, k$ and $\omega$, this speed limit will only impact in the time taken to converge to the target. On the other side, if $A, k$ and $\omega$ are arbitrary big, the robot path will be unstable trajectory tracking is not assured.

Furthermore, the Potential Fields strategy made good results, driving the robot smoothly to the goal point while maintaining a safe distance to the obstacles, but that is only valid when there is no local minimum in the path between the robot starting location and the goal.

The Tangent Bug algorithm proved a good strategy for following a target in a workspace with obstacles. Although, the high level algorithms has a lack of details that impacts the implementations. When developing such strategy in a real robot or in sophisticated simulators such as ROS one must develop additional strategies to deal with high level abstractions of tangent bug.

Finally, the Wave Front algorithm also got good results, and is resolution complete, meaning that it have no local minimum problems. But the necessity to process the entire map before starting the exploration needs to be taken in account in order to determine if it is a good approach to a problem.

## REFERENCES

[1] Howie M Choset, Seth Hutchinson, Kevin M Lynch, George Kantor, Wolfram Burgard, Lydia E Kavraki, and Sebastian Thrun. *Principles of robot motion: theory, algorithms, and implementation*. MIT press, 2005.

[2] 101 Computing. Lissajous Curve Tracing Algorithm. 101computing.net/python-turtle-lissajous-curve/. Accessed in November 1st, 2021.

[3] Geeks for Geeks. How to check if two given line segments intersect? geeksforgeeks.org/check-if-two-given-line-segments-intersect/. Accessed in November 8th, 2021.

[4] Hassan K Khalil. Nonlinear systems third edition. *Patience Hall*, 115, 2002.

[5] Adriano M. C. Rezende. A simple way to make a differential robot with a lidar follow a wall. github.com/adrianomcr/follow_wall, 2020.

[6] Mark W Spong, Seth Hutchinson, Mathukumalli Vidyasagar, et al. *Robot modeling and control*. Wiley, 2005.