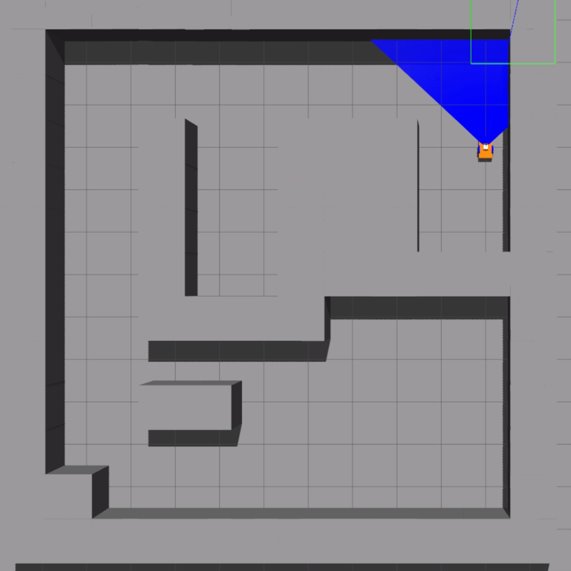


NAVIGATION AND OBSTACLES EVASION ALGORITHM

Advisor: Igal Alterman

Raeed Mundow and Wiaam Fares



**Motivation:** With the rapid advancement in robot technology, the need for autonomous navigation protocol is getting increasingly crucial. Giving the robot the ability to draw its own path and avoid obstacles using modern AI algorithms, removes a lot of pressure of the user (or owner) from having to navigate it manually or making sure it does not hit something and break.

**Project description:** Build a simulation for a robot that can navigate a maze that is filled with obstacles and dead ends using a navigation algorithm. The robot does not have a map layout, so it must learn the map while navigating.

**Initial plan:** First we start with the easier simulation to build, a known map and a robot that can navigate it. Then, we can use it as a basis to build another simulation where the map is not known, and the robot must figure out a way to navigate it. To start, we need a simulation environment. There were many candidates, and we chose to go with Gazebo ROS.

**Gazebo:** Gazebo is an excellent way to simulate a robot in a real life like simulation. In it, we can build a robot from scratch and configure it. We can adjust the wheels size, torque and how slippery it is. We can also build a map in many ways, we can add laser scanner, sun light, slopes, and many other things. But for this project we will stick with something simple, a simple grid map and toy car robot, both will be built from scratch.

**ROS:** short for robot operation system, is a set of software frameworks for development of robots. The framework we’ll use is called topics. We can communicate with the robot system using something called a topic. A topic can include a position and orientation, laser data or time stamps. Each topic can send and receive a message of a certain type (for example: Twist, responsible for movement, works with twist\_msgs). To establish communication, we declare a Node Handler. Through it we can subscribe or publish to a topic using the specified message type. One node is enough for multiple topics, we receive and send messages in parallel (software threads).

**Environment:** Now that we have established the tools we are going to use, we can start talking about the setup. We divided the environment into two sections, one for the physical environment, and one for controlling the robot. Here is how the file system looks like:

The two section we mentioned are called Simulation environment and Commander. The main launch file includes all the files necessary to build an environment, and the Cpp file will be responsible for controlling the robot. Here is how the include hierarchy looks like:

Main launch

Sub launch

World file

Robot file

Navigation Cpp file

Note that the Cpp file doesn’t include any other file. Before we talk about how to run these files, we should go over all the files and mention what their part in the overall build is.

**World file:** It is a file that ends in .world and it is responsible for creating the world. It is written in xml and contains all the obstacles we placed in the map.

**URDF file:** It is a file that describes the robot and the physics of it. It is written in xml. It contains the position and orientation of the robot, the visuals, the physics, and the laser mounted on top.

**The sub launch file:** It is a file that can launch the environment, we decided that it’s not the one that will be called, instead we made another launch file that will call this one when we launch the Gazebo. It is also written in xml. It contains the physical engine, light source the position of the robot, the slope of the map and the friction of it.

**The main launch file:** This file is contained in the Commander section; it calls the sub launch file and will be used to run the environment.

**The C++ file:** This file will control the robot using C plus plus. In it we will use ROS framework to subscribe and publish to topics.

**XML packages:** In each section we have an xml file that includes the necessary packages to run the environment.

**Makefile:** In each section we have a txt file that will be used to build the environment.

**Running the Gazebo:** Before moving the robot we first have to build the world and display it, to do that we run the following commands from the workspace terminal:

* source/opt/ros/melodic/setup.bash
* catkin build
* source/devel/setup.bash
* roslaunch commander <main launch file>

**Moving the robot:** To move the robot, we open another terminal from the workspace, and the run the following commands

* source/devel/setup.bash
* rosrun commander <Cpp file>

**Navigation algorithms:** Now that we went over the environment and how to run it, we can start talking about how the robot will navigate the maze. As we mentioned earlier, we will take two steps, one in a known map, and one where the map is not known.

**Navigation known map:** There are many algorithms that can be used to navigate a map:

- BFS

- Dijkstra

- Greedy

- A star

All these above are attractive options and would get the job done, but we’ll go for A star for the following reasons:

* It is optimal for a finite map.
* It works on a heuristic, which emulates how a human would think in these kinds of situations.
* It works best with a predetermined starting and ending point.
* It will be easy to extrapolate to an unknown map.

We should also mention that A star is not perfect and has some drawbacks, for example:

* It does not work for an unknown map; we have to modify it.
* The efficiency of the heuristic is dependent on the map layout, so one heuristic can be optimal for a certain map and sub optimal for other maps.
* It works on discrete states, so we’ll have to move the robot on a grid instead of continues x, y coordinates.

Despite these drawbacks A star is still very good for our project, and so we will stick with it.

**A star pseudo code:**

Open\_list = empty

Closed\_list = empty

Add start to Open\_list

While(Open\_list not empty)

        Current = Open\_list.pop

        Create 4 neighbor nodes

        For each node

                If node = finish

                        Return path

                If node not in Open\_list or Closed\_list

                        Add node to open and calculate heuristic

                Else

                        Check if path from start node to current node is better

                        If so, remove it from Closed\_list and add it to Open\_list

       Add Current to Closed\_list

**The map representation:** For A star to work, we must define node states. To do that, we created a 2D map that represents the map layout, we inserted 1 where there is an obstacle and 0 where we have an available tile we can move to. A state node is the position of the robot in this map.

**Overall algorithm for a known map:**

- Represent the map as a 2D matrix

- Define a starting point and an end point

- The position of the robot at any giving point is a state node

- Use A-star algorithm

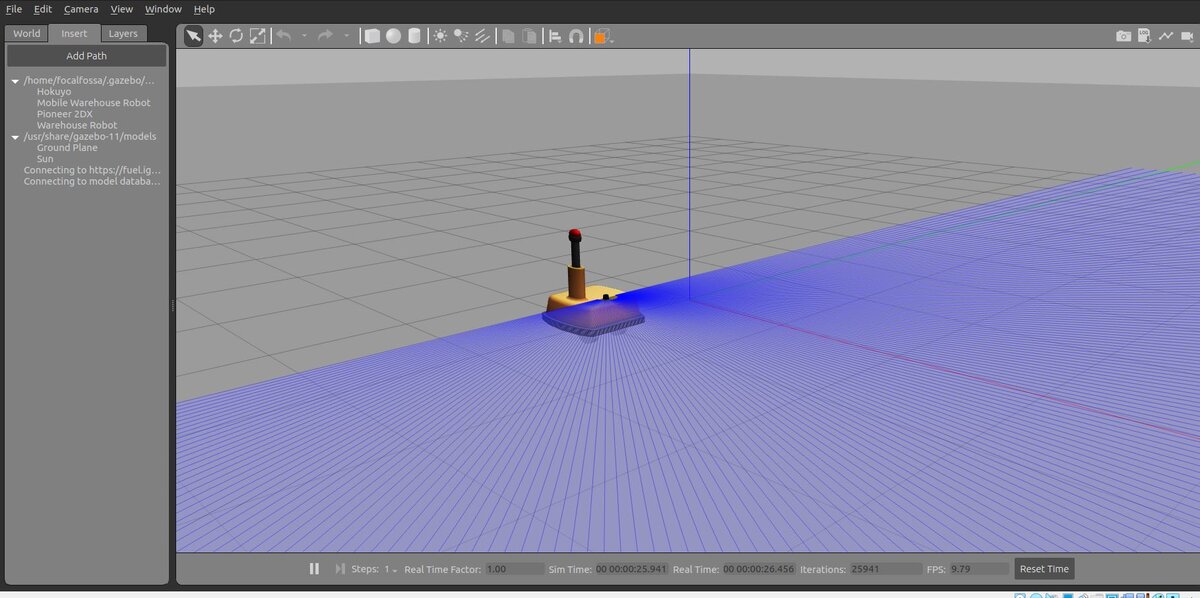
- Use Manhattan heuristic

- Use the path generated by A star to move the robot using a topic

**Solution absoluteness:** One thing to mention is that the solution is absolute, it is generated before the robot even moves, and it is only dependent on the starting and ending point and not on the relative position of the robot. Obviously, this is not the case for an unknown map.

**Navigating an unknown map:** Before we even start talking about the algorithms needed for this task, we should talk about the utilities the robot needs. We need a tool that would let the robot see what is in front of him, and some function that he can use to identify obstacles and free tiles. First, we’ll use a laser scanner to “see”. Second, we’ll use a 2D map to memorize obstacles and free tiles. Third, we’ll write some functions to identify the obstacles and free tiles.

**Hokuyo Laser:** For our laser scanner, we went with a pre-built plug-in laser called Hokuyo. It is included in the robot URDF xml file, there we can parametrize the laser, meaning we can control the width and depth of it, the sample amount, as well as its error std. It works by sending pulses, each in a defined angle going from minimum angle to a maximum angle, if the pulse hits something it writes in an array in an appropriate index the distance it traveled. This array can be called using the laser scan topic in our C++ file. Here’s an example of a laser going from -180 to 180 degrees:



**Analyzing the scan results:** Using the data in the laser scan array to identify the obstacles is not an easy task. To make this simple, we will only analyze the scan when the robot is standing still and is orientated in either North, East, West or South direction. This means, when the robot is not moving, it will scan what is in front of it, identify the obstacles, and according to his orientation, we rotate the result to fit his orientation and the map. For this task, we wrote a lot of helping functions, for example, one functions returns the orientation of the robot, one maps the point from the POV of the robot to the actual map coordinates….

**Identifying the obstacles and free tiles:** Using sinus and cosine on the angle we can find the coordinate of the obstacle. To find the available tiles we first calculate the angle between the robot and the tile itself and check the ray that goes in that angle if it reaches further than the tile.

**Updating the map:** In an unknown map, we start with a 2D map, represented as a matrix, filled completely with unknown areas, marked with 1’s. and each time we find a free tile or road, we update that in the map matrix with either 0 (free tile), or 2 (obstacle). As we go through the map, we slowly build up the environment. Along the way of the project, we also updated this part of the project to detect dead ends, we’ll touch on that later.

**Navigating the unknown map:**

* Scan
* Identify obstacles and available tiles
* Get rid of dead-ends
* Update the map
* Pick a neighboring point using Manhattan cost formula
* Move there
* Repeat the process until end point

**State machine:**

Start, Scan

Scan = true

Move = false

Search = false

scan

search

Move

End

Scan

Scan = true

Move = false

Search = false

Update map

Scan = false

Move = false

Search = true

Scan, if not end

End

Scan = false

Move = false

Search = false

Navigate

Scan = false

Move = true

Search = false

**Pseudo code:**

Current\_state = start; //Global variable

Scan()

UpdateMap()

Open = getNeighbors(start)

Sort(open)

While open!=empty :

next = open.popfront()

if next is available:

updateState(next)

Current\_state = next

open.empty()

return

Backtrack()

**Optimization:** To make our solution faster we added two optimization functions:

- Backtracking: In case we reached a point with no further optional point, we back track our steps until we reach an unvisited neighbor.

- Dead-end elimination: right after we identify obstacles and free points, we run a function that checks if the free points lead to a dead-end, if so, we update them on the map as obstacles, so we don’t have to visit them.

**Backtracking:** In case we reached a point with no ending point in sight, we’ll trace back our steps until we reach a point where one of its neighbors is still not visited.

**Dead-end elimination:** sometimes the robot would wander into a well where there is no possible exit and take a lot of time to get out, to solve this, we wrote a function that can recognize a dead end and tell the robot that there is no point in going there. To do this, we must change the way we update a map. We start by a matrix filled with 1’s like previously, when we see an obstacle we update the appropriate index with a 2, and before we update the free tiles with 0’s, we update them with 3’s then we run a function that goes over the 3’s in the map and check if they belong to a dead end, if not it will turn them into 0’s.

**Eliminating the threes of the map:** How do we know if we have a dead end? To answer this, we must ask the question: what separates a dead end from a normal free tile. A dead end is corned by three directions with obstacles and the last direction by a free tile, why? Because if so, then the robot can see the free tile and doesn’t have to go to the bottom of the dead end to reach the said zero, it can go to it directly (Pythagoras theorem indicates it’s shorter to move to it directly). So, we go over each 3 on the map, and check if it’s a corner like (has two adjacent neighbors which are obstacle) if so, move along the other direction and check if we hit another corner, if so, it means the path we took is a dead end and we can get rid of it.

**Environment variables:** In this kind of simulation a lot of variables can change the efficiency of the algorithm. For example, changing the speed of the robot can obviously speed up the solution. But we’ll talk about variables that can affect the algorithm not the physics. These variables are:

* Starting and ending point: Changing one of these has a huge impact on the execution time for obvious reason.
* Laser depth: The laser depth didn’t change the execution time that much; it mattered when the robot encountered a dead end. depending on the depth of the dead end it might not recognize it as such and then would enter it. Here are examples of a short depth and long one:

Chart

Description automatically generated

Chart

Description automatically generated

As you can see, in the second picture the laser can’t reach the end of the dead end.

* Laser width: The width of the laser can affect the execution time of the algorithm, but it also mostly mattered when hit with a wide dead end.

A screenshot of a computer

Description automatically generated with low confidence

If the laser isn’t wide enough it might not recognize the dead end.

And so, we decided to go with a laser with 90 degrees of width (laser rays go from -45 to 45 degrees), and a depth of 7 tiles. This way we don’t make the navigation too easy and cover up most of the obstacles and dead ends.

**The final product demo:** Here’s a link to a demo of the final product

**Possible improvements to implement:** Obviously this is a simple prototype that can be improved and built on to create a much more versatile navigation system. There are a lot of ways to improve the algorithm, for example:

* More differential and continues movement and positioning: we can increase the size of states to (more tile density). This will make the movement more fluid since the tiles are smaller and closer to each other. It can also make the scanning much better.
* Better movement function: we can improve our movement function to make the robot move more appropriately to its surroundings.
* Better laser analytics: One subject we had a hard time dealing with was the analytics of the laser data and how to use it to identify obstacles and free tiles. It can be improved to make it more dependable.
* Better heuristic: This is an open research question but maybe a different heuristic can make the execution time much lower, but it would also be map dependent.
* 3D extrapolation: We can extend this algorithm to three dimensions, and thus usable for drones. It should be easy to do this, but it would take a long time, since we must change the whole environment not just the movement.

**Applications:** This protocol can be implemented in any grounded robot that has to reach a certain point or navigate through obstacles. It can (potentially) be also implemented in arial moving robots (drones) if the 3D algorithm is done correctly.