A* path planning in ROS

Student: Kryštof Hes

Course: Robótica móvil, Universidad de Sevilla

Date: 11.2.2020

Project description

The aim of the project is to investigate the function of graph-based algorithms, especially the A* search algorithm. It should be implemented in Matlab and its functionality should be demonstrated on a few examples. In the end, the algorithm should be implemented in ROS - ideally with a simulation of autonomous robot navigation though an environment in Gazebo.

Introduction

Path planning is a crucial task for autonomous robots. In order to navigate through an environment from a start point to a given navigation goal (end point), they have to calculate an optimal path in terms of distance covered or other cost variables. Furthermore, they must avoid obstacles and constantly map their environment using sensors to localize themselves and watch out for dynamic obstacles or other unexpected events. During the years of development in the field of robotics, several approaches to the different parts have been established. In this project, the *path planning task* should be investigated.

Path planning algorithms

In general, the path planning algorithms are distinguished between graph-search algorithms using deterministic methods and randomized planners which can provide only probabilistic guarantees of having encountered the optimal path. The two approaches differ substantially. Whereas the graph - search methods can only deal with cells and thus with positions determined by them, the probabilistic approach is able to use arbitrary positions of poses, as well as arbitrary forms of obstacles. The major difference between the two methods is also the outcome. The cell-based planning can only calculate an optimal sequence of nodes (grid cells, waypoints) and does not take into account neither the vehicle model nor the input limits or vehicle constraints. In other words, the methods do not ensure that the robot can carry out the task due to its kinematics constraints such as maximal joint angles etc. Neither do they provide any information about in which time the robot reaches the goal or with which acceleration it has to go from one point to the other. However, all this is considered in the randomized planning so that the output is a true trajectory.

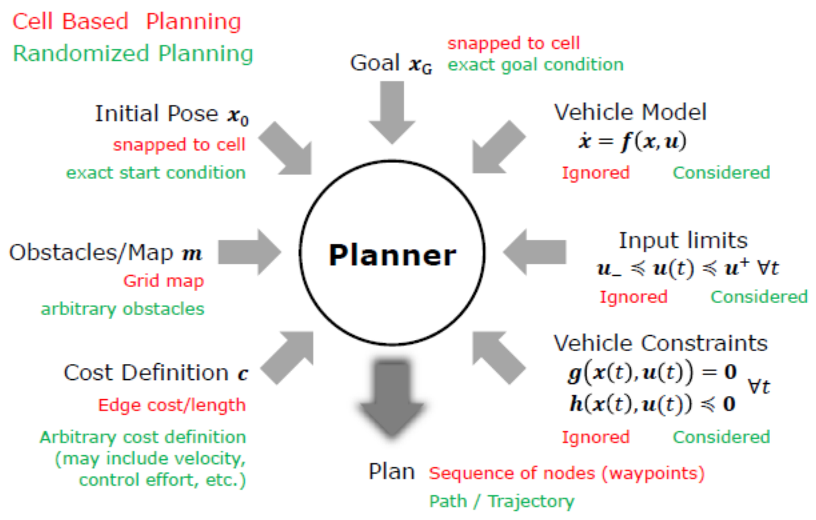


Figure 1 - Comparison of path planning approaches

The difference between a path and a trajectory is that the first only determines a sequence of points through which the robot should go whereas the later informs also about at what time should the robot be at which point. Therefore, the final time is undetermined when using a path and fixed at design time when dealing with a trajectory. If the robot moves itself on a path, the control needs a tracking system in order to know where it is located. Such systems can be expensive. The advantage of trajectory planning is that it's already known where the robot is, so the cost gets much lower. However, it's possible to obtain a trajectory from a path by parametrizing it. Starting from the path in parametric form $x(\tau)$ we use the parameter τ as a function of time to obtain the trajectory.

$$x(\tau) \;\rightarrow\; \tau(t) \rightarrow x(t) = x(\,\tau(t))$$

In this project, graph-search algorithms, especially the A*, should be described further in detail.

Graph-search algorithms

First step of this type of algorithm is the decomposition of the environment into a grid of equally sized cells. The environment is then defined as an occupancy grid, a binary matrix which indicates whether the corresponding cell is an obstacle (1) or free space (0) where the robot can move.

Secondly, the neighbourhood structure must be defined. In a 2D space, typically 4 or 8 cells are considered as neighbours as shown in the figure below. For the diagonal ones, the cost to achieve them is higher as the path there is a hypotenuse of a triangle. Commonly, the cost is approximated as being 10 for the "direct" neighbours and 14 for the diagonal ones. Similarly, in a 3D space one could define a 6, 18 or 26 cells neighbourhood.

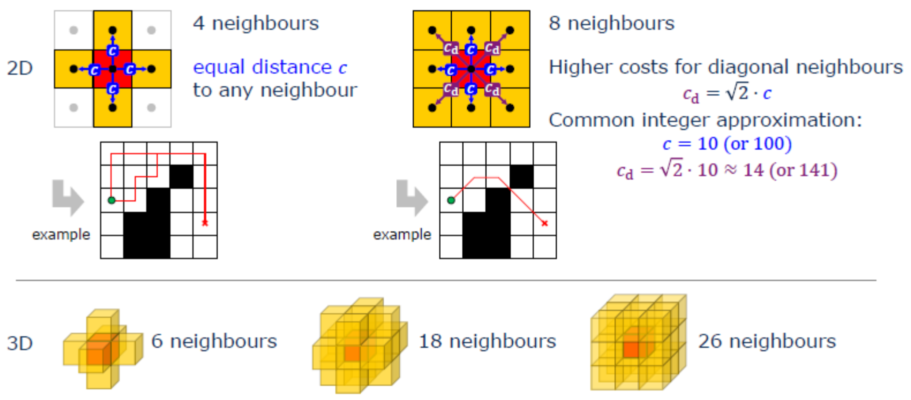


Figure 2 - Neighbourhood structures

When the neighbourhood is defined, the search algorithm is applied and finally a trajectory is implemented based on the calculated optimal path. This can be done in cooperation with a local planner which ensures that the constraints neglected by the graph-based search are taken into account (as discussed in the previous paragraph).

There are several graph-search algorithms, and these should be briefly introduced in the following part. The focus lies on the A* as it was implemented in the practical part of the project.

A*

The main idea is to advance through the cells of the grid in the correct direction. This is done by assigning costs to each cell. There are three types used:

$$\begin{aligned} &g(x)\text{: } \textit{cost to come to the cell from the start} \\ &h(x)\text{: } \textit{estimated cost from the cell to the end} \\ &f(x)\text{: } \textit{total cost}, f(x) = g(x) + h(x) \end{aligned}$$

The movement cost starting from the start point and going to the current cell is g(x). The prediction of the cost to finish, also called heuristics, is h(x). There are multiple possibilities of its calculation.

Manhattan distance:

This heuristic is used when only 4-directions movement is allowed (thus no diagonal movement)

$$h(n) = |n.x - goal\_cell.x| + |n.y - goal\_cell.y|$$

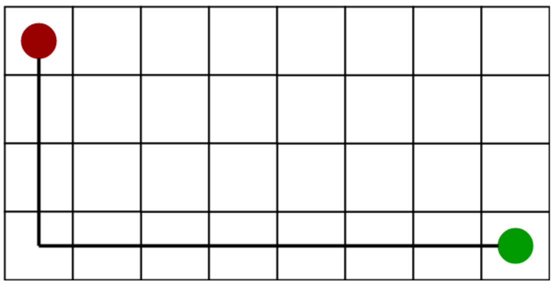


Figure 3- Manhattan distance heuristics

Diagonal distance, uniform cost

If the motion in the diagonal direction is considered to have the same cost as the direct one, this heuristic can be used. This calculation is only accurate if the diagonal and non-diagonal cost can be assumed to be the same.

$$h(n) = \; c \cdot \max\,(\,|n.x - goal_{cell}.x|, |n.y - goal\_cell.y|\,)$$

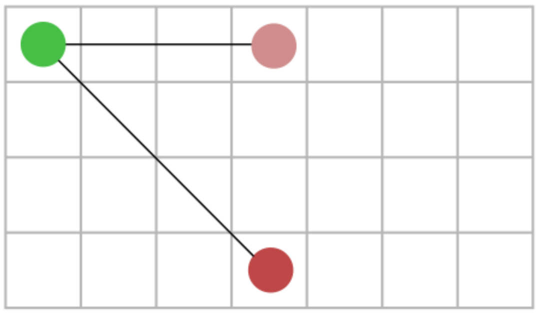


Figure 4 - Diagonal distance, uniform cost

Diagonal distance

In case of 8-directions movement and different diagonal and non-diagonal costs, the following calculation can be used. The robot has to travel in x and y direction. The direction with the lower number of cells is chosen to be travelled diagonally (e.g. with higher cost per cell). When the diagonal motion is complete, motion in either x or y is already finished and there are dmax – dmin cells in the other direction to be walked through in order to come to the end point. These are covered in the direct direction.

$$\text{direction\_max} = \max\,(\,|n.x - goal_{cell}.x|, |n.y - goal\_cell.y|\,)$$

$$\text{direction\_min} = \min\,(\,|n.x - goal_{cell}.x|, |n.y - goal\_cell.y|\,)$$

$$c_d = diagonal\ motion\ cost \qquad c_n = direct\ motion\ cost$$
$$cd = \sqrt{2} \cdot c_n \approx 1.414 \cdot c_n$$

$$h(n) = \; \text{direction\_min} \cdot c_d + c_n(\text{direction\_max} - \text{direction\_}min)$$

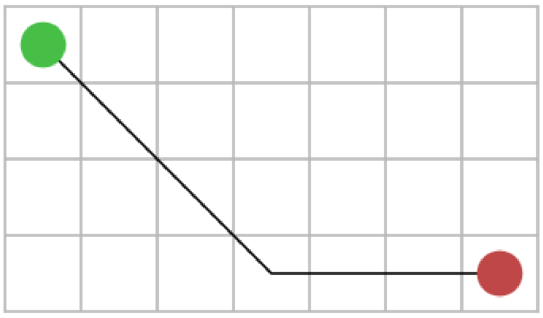


Figure 5 - Diagonal distance

Euclidean distance:

It's the distance as the crow flies between the current cell and the goal. It can be used when movements in all directions are allowed. Since taking a square root is a relatively time-demanding computing operation, it's a good practice to omit it. The results won't be changed by this.

$$h(n) = \sqrt{(n.x - goal.x)^2 - (n.y - goal.y)^2}$$

$$h(n) = (n.x - goal.x)^2 - (n.y - goal.y)^2$$

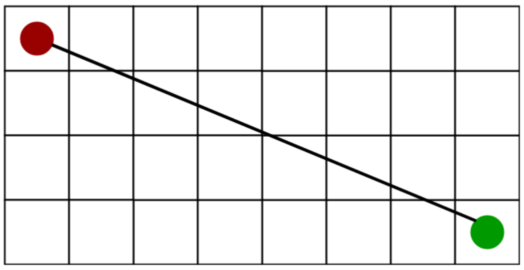


Figure 6 - Euclidean distance heuristics

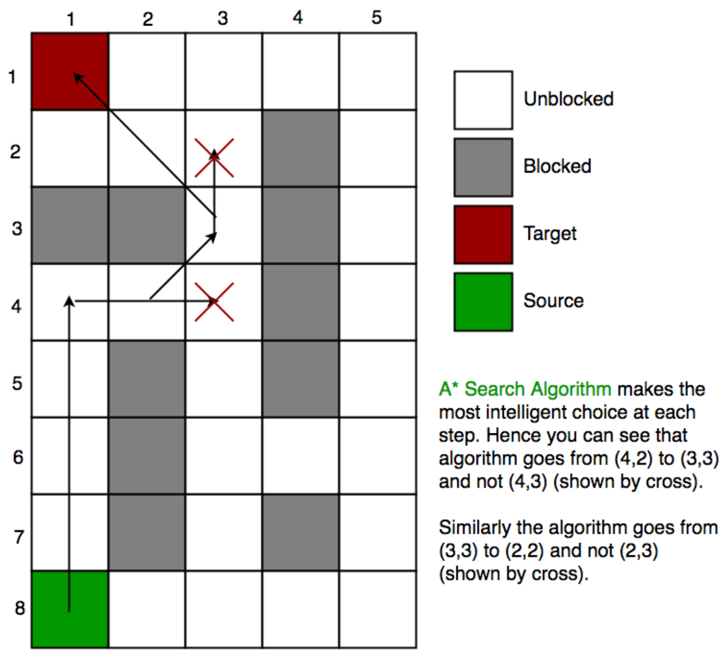
The algorithm

The cells in the grid are divided into two lists – opened and closed. The open list contains cells which are candidates for the path and in the closed list cells which have already been processed are put. The algorithm starts with putting the start node in the open list and proceeds step by step towards the goal. The algorithm is either terminated by reaching the goal node or by having open list empty and all the grid cells in the closed list. In this case, there is no possible path between the two points (start and end point are divided by an unpassable obstacle).

The process can be described as follows:

```
1.  Initialize the open list
2.  Initialize the closed list
    put the start node on the open list

3.  while the open list is not empty // e.g. if there are still cells to be processed
   a) find the node with the lowest cost f on the open list, call it f.ex. "current"
   b) pop "current" off the open list
   c) generate "current's" 4 or 8 successors (children) and set their parents to "current"
   d) for each child
      1) if child is on the closed list, skip it
      2) if child is the goal, stop the search
          if child still not the goal, calculate the variables
               child.g = current.g + distance between child and "current" (see topic
                                neighbourhood)
               child.h = distance from goal to child (using heuristics)
               child.f = child.g + child.h

      3) if a node with the same position as child is in the open list:
               - if the child has a lower g than the node in the list, update the values (of the node in
               the list) and the parent
               - continue to the next child (skip adding it to the open list)
      4) add  the node to the open list
    end (for loop)

   e) push "current" on the closed list
   end (while loop)
```

How the algorithm works in practice shows the following example:



1
2
3
4
5
1
2
3
4
5
6
7
8
Unblocked
Blocked
Target
Source
A* Search Algorithm makes the most intelligent choice at each step. Hence you can see that algorithm goes from (4,2) to (3,3) and not (4,3) (shown by cross).
Similarly the algorithm goes from (3,3) to (2,2) and not (2,3) (shown by cross).

Figure 7 - A search example*

Dijkstra

Dijsktra is a special case of A* algorithm where the heuristic function is 0 for all the neighbours. This means that the algorithm does not prefer to continue working towards the goal direction but instead searches through all the cells. Therefore, it's less efficient and more time-consuming.

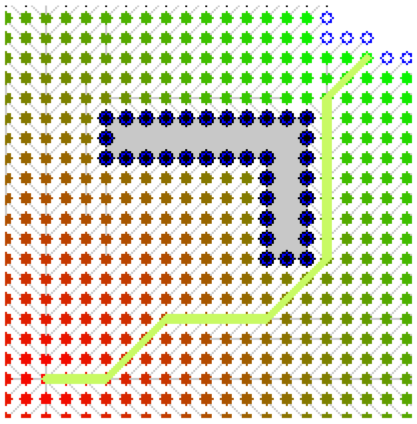


Figure 8 - Dijsktra algorithm in comparison

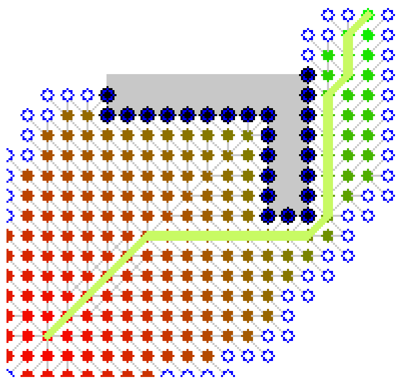


Figure 9- A algorithm in comparison*

D*

It's a variation of A* algorithm. Instead of planning the complete path from the start to the end, it updates the path continuously on the go based on the actual sensor readings.

Matlab implementation

The implementation in Matlab should be split into two parts: the GUI (user input and algorithm visual output) and the algorithm itself. First, the GUI is to be described.

GUI

The input is possible through loading an image and converting it into an occupancy grid.

Displaying the map

There are two cases of how the occupancy grid and the path can be viewed. When the image is small (a few pixels for tests), it needs to be enlarged in order to view the map, the start and end points, as well as the path. Therefore, a scale is specified to magnify it. For example, if the scale is 10, then 1 px in the bitmap will be converted into a square of 10 by 10 px for display. In the case of a large enough image, there is the problem that 1 px such as the start/end point or points on the path can't be seen well anymore. A scale is set up but this time it enlarges only the mentioned points and not the map itself. For example, if this scale is set up to 3, the single start point pixel is converted into 10x10 px region (3 px before and three after the point). In the first described case, the function `dispImageEnlarged` is called, in the latter `dispImage`.

User input

The start and end point can be chosen by clicking in the map. The selection is done by a left-click and ended by a right-click. This feature is possible due to the `getpts()` function of Matlab. After the points are selected, they are marked by calling the self-written `markPoint()` function .

Algorithm

The nodes are implemented as objects of the *Node.m* class. This has the properties:

```
properties
        position   % (i,j) position of the node
        parentNode % previous node on the path
        g          % cost to arrive at the current node
        h          % estimated cost to the end
        f          % total cost, f = g+h
end
```

The node is created by calling the constructor and passing the parameters position and parentNode

```
function obj = Node(position,parentNode)
```

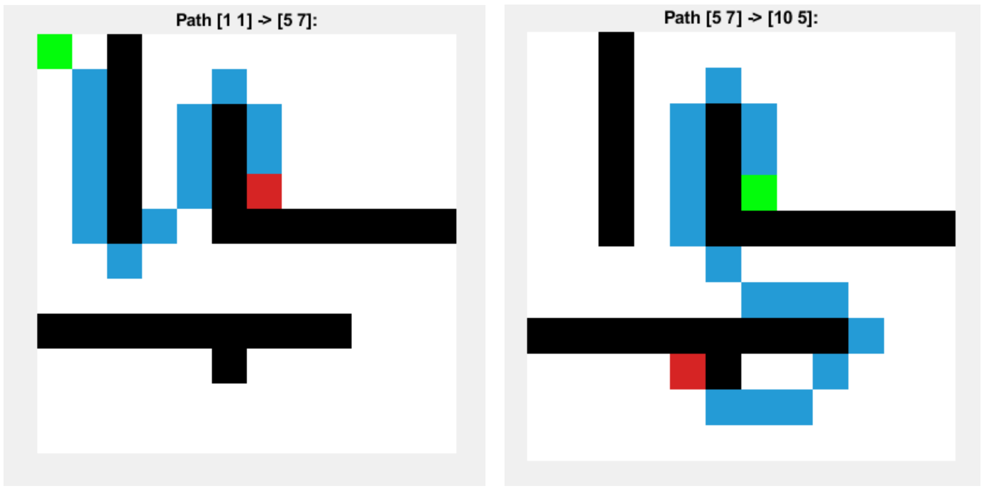
The code is written according to the already presented pseudocode. The two funcions which are being used are `is_in_list` and `get_node_idx`. The first finds out whether a given node object is in the list or not. The latter returns its index in the list.

After the algorithm reaches the end point and thus finishes the path calculation, the path can be returned by creating a new vector `path`, adding the current node (final point) position and then proceeding in a loop towards the start by setting current node to the parent, saving its position. At the end of the procedure, the array is flipped so that the start point becomes the first entry.

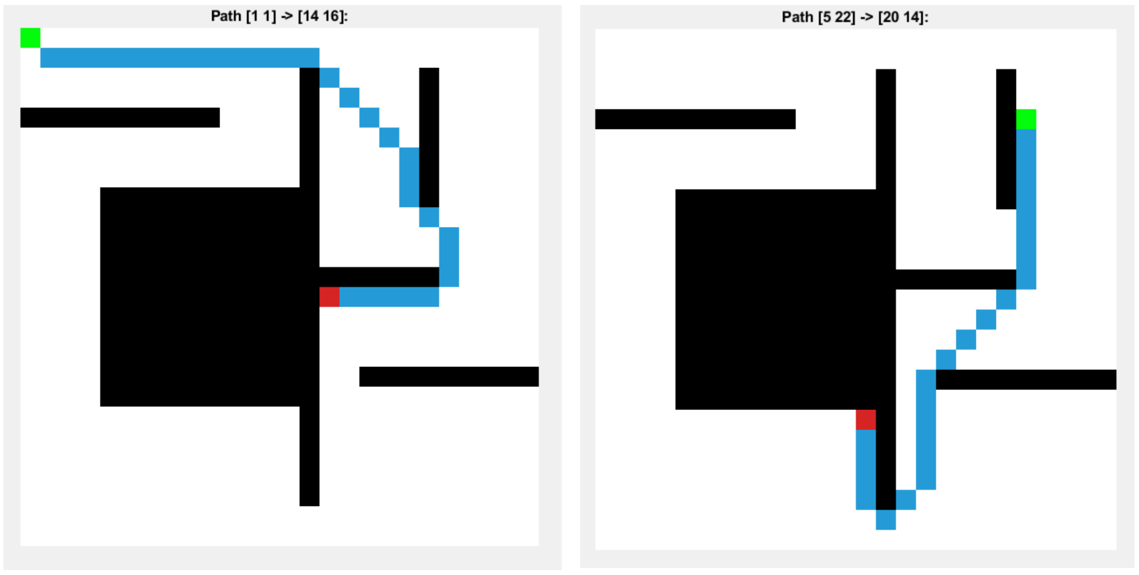
Examples

The original image used for the following examples is a following 12x12 grid:

The examples use a diagonal distance heuristics and 8-point neighbourhood.



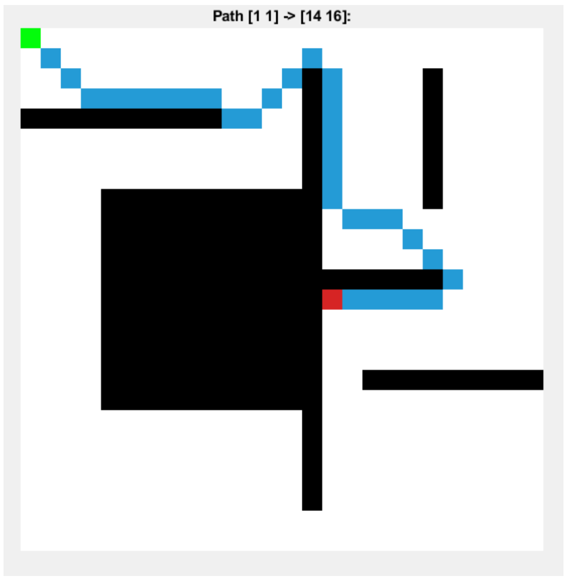
Two more path examples using diagonal distance heuristics. Clearly, the algorithm seems to work well.

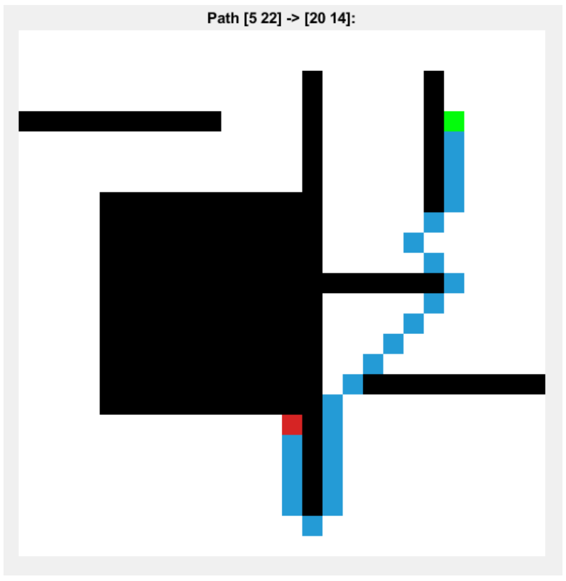


Now the same map and points are run with a different heuristics. Instead of using diagonal distance heuristics, now the approximation of the euclidean is used.

$$h(n) = 10 \cdot ((n.x - goal.x)^2 - (n.y - goal.y)^2)$$

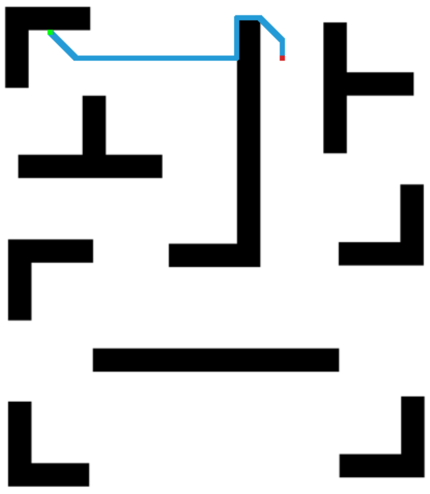
This procedure was suggested in one tutorial. It is true that by omitting the square root, the computing time reduces and it is true that if we use 14 as the approximation of $\sqrt{2}$ for the diagonal neigbor cell's distance, it seems necessary to multiply the number of cells by 10 to get the distance. However, the problem is that now the g and f values are not to scale. By omitting the square root, the h value is unproportionally high. This increases the significance of the heuristic when choosing with which cell from the open list to proceed. As this depends on f and f is the sum of g and h, if h is significantly higher then g becomes far less important. The effect is clear in the following example. The algorithm tries to proceed in the direction of the goal altough the path becomes less optimal.

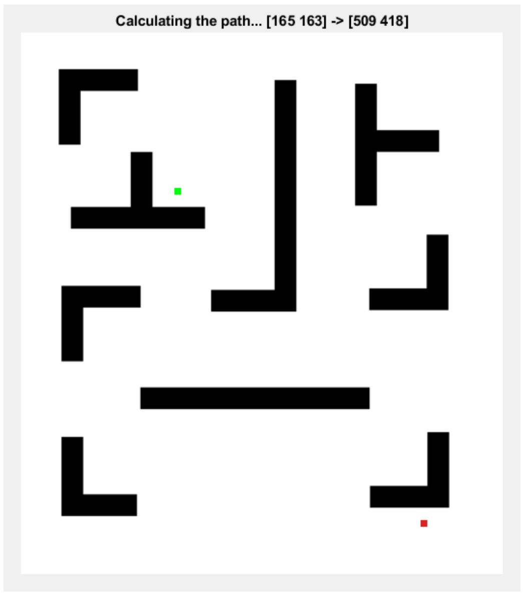


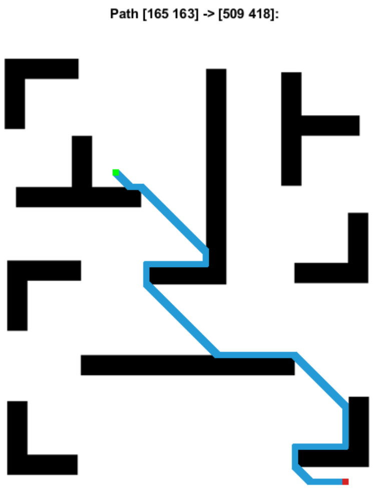


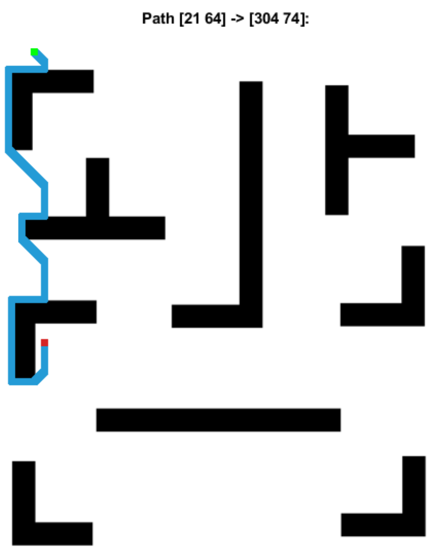
Although the algorithm seems to work perfectly for small-scale maps, when it's run on bigger scales, the computing time becomes unbearably long. The possible cause can be the searches in the open and closed lists. As the number of elements increases, the computing time rapidly grows. Therefore, it would be useful to implement the lists as binary heaps in order to make the search faster. However, due to missing time, it was not possible to accomplish this in the project.

When using the eclidean distance heuristics, the algorithm becomes fast enough in order to calculate even bigger maps. However, the path's are not optimal as shown in the following examples. The reason is the same as already discussed – the heuristic function h is not in scale to g.

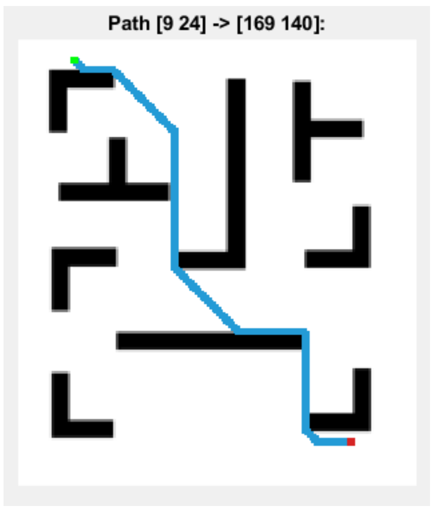


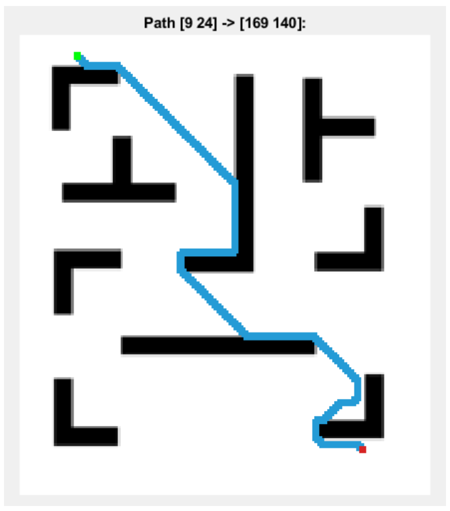

Calculating the path... [165 163] -> [509 418]


Path [165 163] -> [509 418]:



Two more examples have been calculated. The first one using the diagonal heuristics and the latter the euclidean without the square root. Whereas the first took minutes to calculate, the other was ready in a few seconds. Clearly, it is not optimal however.





Decreasing h means that the A* algorithms expands itself on more cells and needs more computing time. The extreme is when h is set to 0 and the A* algorithm turns into *Dijsktra*. By increasing h the algorithm does not expand so much and is faster but will probably not find the optimal solution. The extreme is with h so high that g loses importance and the algorithm becomes so-called *Greedy Best-First-Search*. The fine-tuning of A* lies in finding the balance between the accuracy and speed. Depending on its application, it can be often useful to obtain a non-optimal path but faster (for example in computer games).

ROS and Python implementation

Following the algorithm verification in Matlab, a simulation of robot navigation through a map should be carried out. The aim is to get familiar with the ROS environment and to get the knowledge of the different parts needed to accomplish the task – communication between nodes, client-server communication, robot models, modelling of environments, odometry data and velocity commands, custom message creation and more.

Introduction to ROS

ROS (Robot operating system) is a framework for writing robust software for robotic systems and is widely used in robotics today. It includes libraries and conventions for all aspects of robotic systems – navigation, communication, localization, mapping, visualization, physics modelling and simulation etc. It can be run on Ubuntu and the code can be implemented in C++ or Python. In this project, *Python* programming language together with *ROS Melodic* was chosen.

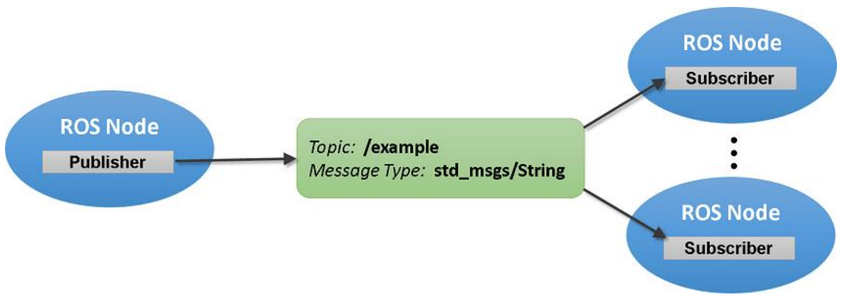
The projects in ROS are to be stored in one workspace, which is by convention named *catkin_ws.* It's a good practice to create this workspace in the home folder. The projects itself are organized into packages, which are created by the terminal command in the workspace directory

catkin_create_pkg <<package_name>> <<dependencies>>

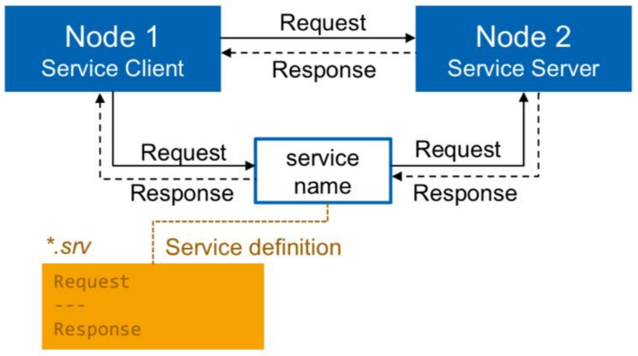
The workspace *catkin_ws* has three important directories which serve different purposes:

src	Source files of the projects we want to build. This is the folder where we work in.
devel	Development space. Here build targets are placed prior to being installed. In this project, the installation was not carried out as the final step.
build	Build space, here the intermediate files during the invoking of CMake for the building of the project are stored.

In robotic systems, the various components are represented as nodes in ROS. There are *publisher* nodes and *subscriber* nodes. The publisher can be for example a sensor sending data about robot position, acceleration, distance to obstacle etc... The data is then sent as a *topic* of a specific *message type* to the subscriber nodes. Multiple subscribers can subscribe to a single topic being published in the system. The message containing the information can be of standard or custom types. How the custom types can be configured is to be explained in the project description. From the default message types, it's important to mention the package *geometry_msgs* which contains types such as *Point*, *Pose*, *Twist* or *Quaternion* – all being extensively used for example for motion control. A second important package for messages is *std_msgs*, which contains all the well-known variable types such as int, bool, string, etc. but also for example time or duration (time difference). The following graphics shows a typical publisher – subscriber information exchange in ROS.



Another type of communication is between a client and a server. The difference is that whereas the publisher-subscriber is a one-way communication which typically runs periodically at a specific rate (specified by *rospy.Rate*), the client server is a two-way communication containing a request and a response. This means that it's a one-time information exchange which is event-driven (client sends a request, service receives it and responds) rather than a periodical one. The service message is specified in the .srv file and the whole communication is carried out according to the following diagram:



Important step when writing ROS code is the building of the code with the command *catkin_make.* The command takes into account all the dependencies listed in the *package.xml* and *CMakeLists* files. In order to use custom messages, it's necessary to list *message_generation* and *message_runtime* in these, as well as the names of the messages.

Project description

The project is named *robot_navigation_project* and contains the following folders and files whose function should be explained.

/doc

Used for basic information about the project in the readme file

/launch

Contains launch files of the project. These are XML files which enable to start multiple nodes with a single terminal execution (as opposed to opening multiple terminal windows and running every node in each of them). Furthermore, arguments can be passed so that for example in this specific case a specific map is launched in Gazebo with a specific robot model in a default and configurable start pose.

An exerpt from the launch file *robot_navigation.launch* which is used to start the configured Gazebo environment should be described:

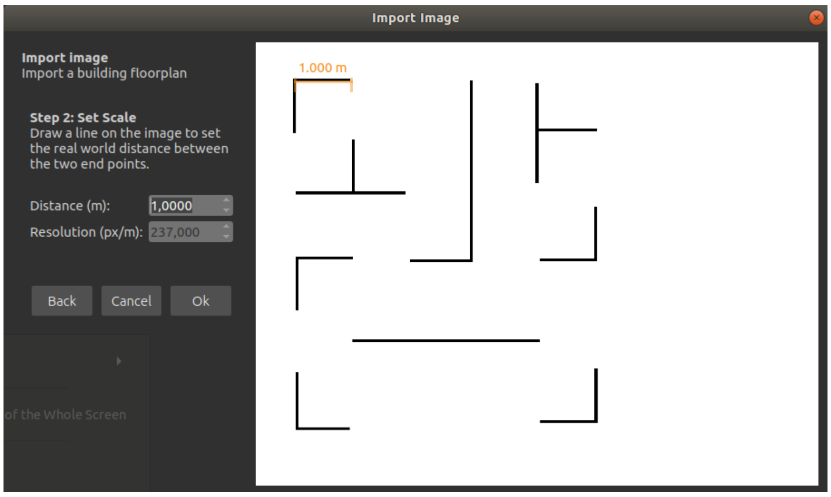
```
<launch>
  <arg name="model" default="$(env TURTLEBOT3_MODEL)"
doc="model type [burger, waffle, waffle_pi]"/>
  <arg name="x_pos" default="0.5"/>
  <arg name="y_pos" default="0.5"/>
  <arg name="z_pos" default="0.0"/>
...
```

The Gazebo environment comunicates with ROS through so-called environment variables. These are variables in UNIX systems that are set up when you log in. They are used for example for storing PATHs and can be permanent (in that case need to be configured in the startup files of the basher) or non-permanent which disappear after closing the terminal. In the case of this project, the environment variable `TURTLEBOT3_MODEL` is used to specify which robot model should be spawn into the Gazebo simulation. The Turtlebot3 platform offers three models which can be downloaded together with ROS. Before executing the launch file, the export of the model is needed with the following command (here for the burger type):

export TURTLEBOT3_MODEL=burger

/maps

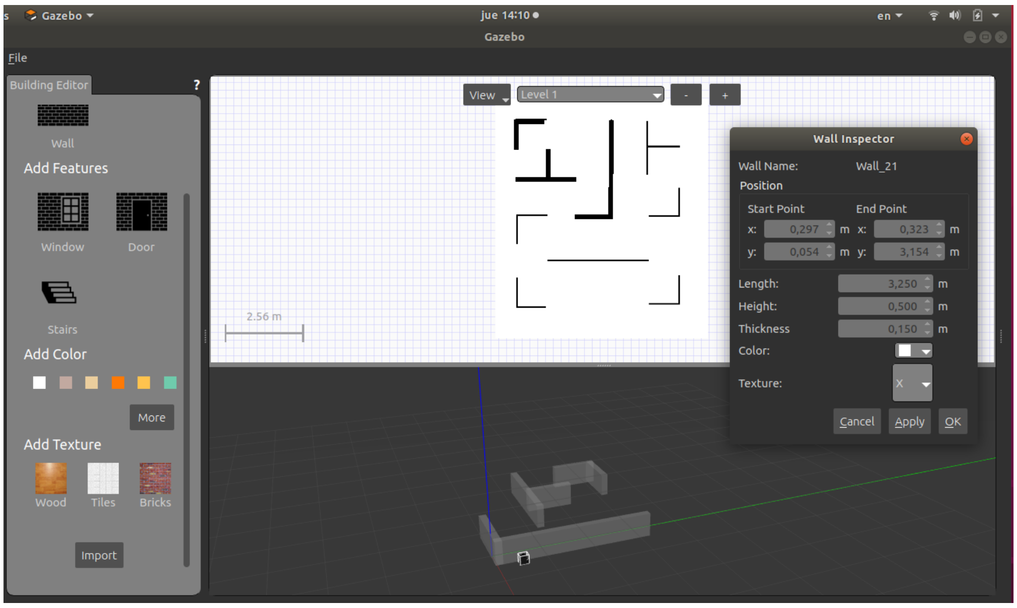
The maps folder includes the png files for the map which was tested in Matlab and was now turned into a Gazebo map. The worlds in Gazebo consist in two files - *.config* and *.sdf* These contain information about the objects in the environment and are created automatically by the *Building editor* integrated in Gazebo. The editor enables to load an image and then draw the walls in the corresponding location. The parameters such as its dimensions, texture or color can be specified. Important is to note that the path-searching works with a grid of pixels and Gazebo uses on the contrary units of m. As a consequence, when creating a map, it's important to note the px/m resolution which is then crucial for the robots navigation.


Import Image

Import image
Import a building floorplan

Step 2: Set Scale
Draw a line on the image to set the real world distance between the two end points.

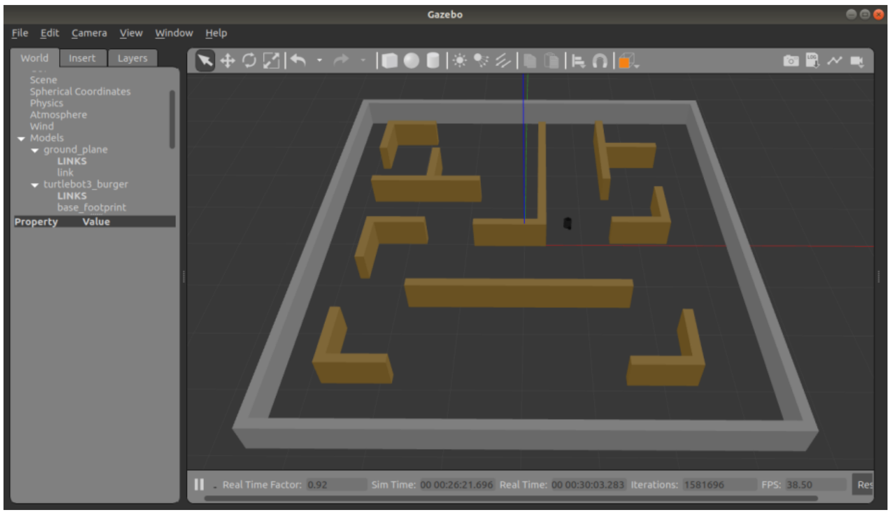
Distance (m): 1,0000
Resolution (px/m): 237,000

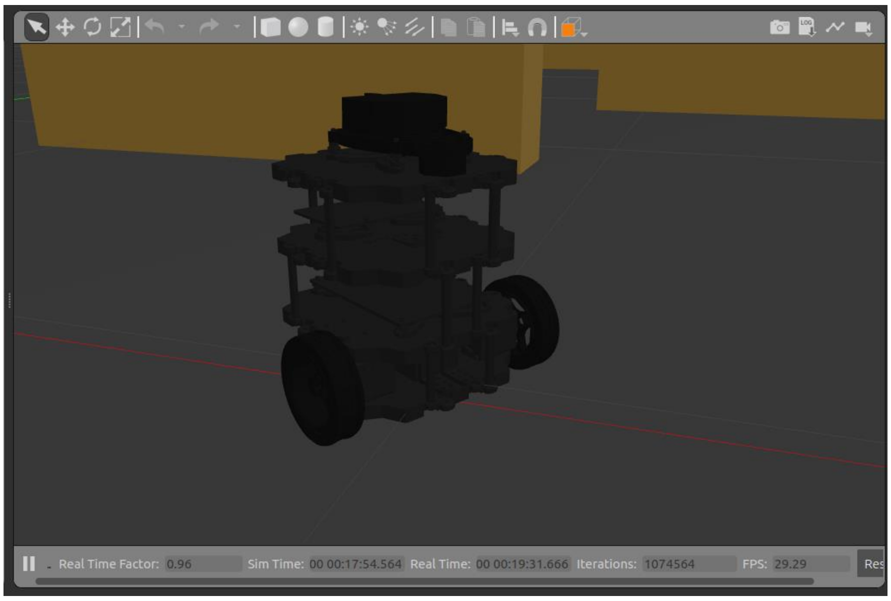


After the map is created, it can't be further edited in the Building editor. The map files are created, but have to be **moved in the Gazebo directory** in order to be able to use them. Its location depends on the ROS installation and probably this directory is only accessible for root, so a sudo-command such as this needs to be used:

sudo cp -r ~/catkin_ws/src/robot_navigation_project/maps/map1 /usr/share/gazebo-9/models

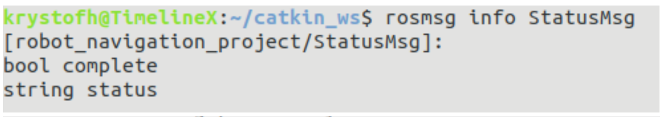
How the complete environment with the robot model spawn looks like show the following figures:



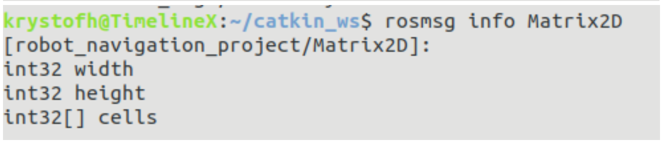


/msg

This folder contains the custom messages used in the project. As described in the introduction, messages are data structures used to exchange information between nodes. For the implementation of the simulation, two custom messages were created, the *StatusMsg.msg* and *Matrix2D.msg*:



```
krystofh@TimelineX:~/catkin_ws$ rosmsg info StatusMsg
[robot_navigation_project/StatusMsg]:
bool complete
string status
```



```
krystofh@TimelineX:~/catkin_ws$ rosmsg info Matrix2D
[robot_navigation_project/Matrix2D]:
int32 width
int32 height
int32[] cells
```

The aim of *StatusMsg* is to send information about task execution between two nodes. The considered application is that a robot controller processes one point of the A* path after each other and the other node (PID controller), which takes care of moving the robot to the point sends back status messages about the completion of the task. Only if the task is completed, the robot controller can proceed with the next point.

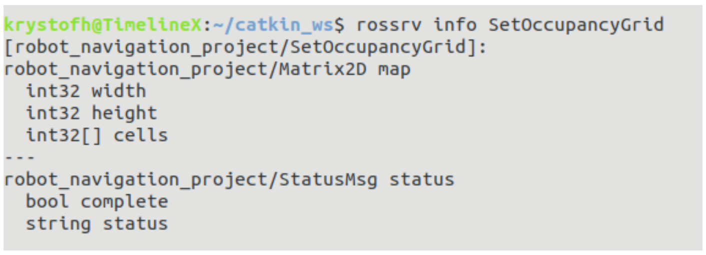
The *Matrix2D* is a message type used for setting the map of the server *path_planner_server.py.* As the map (occupancy grid) is a 2D matrix of integers, there is a need for a structure of this type. ROS offers a message type called *MultiArray*. However, its use is a bit complicated and it's handier to use a custom message instead. It includes the 1D array of the grid cells, as well as the information about the original width and length of the map. This means that for sending, it needs to be reshaped into this form by the client and afterwards the 2D matrix retrieved by the server. However complicated it may seem, it turned out to be possible to implement quickly and to work well.

/srv

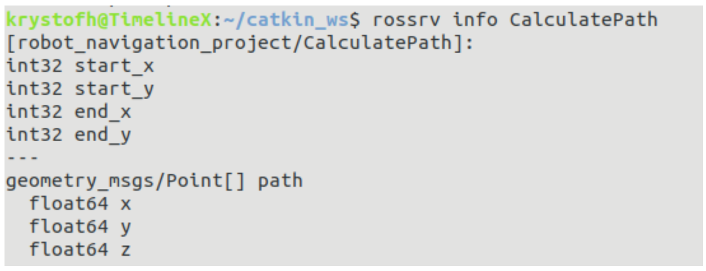
Here the service messages are stored. As already discussed, service message exchange information between a server and a client and have two parts – request and response. Two service messages are used in the project – *SetOccupancyGrid.srv* and *CalculatePath.srv*

The service SetOccupancyGrid is used to send a request to save a map in the server. The request part contains the map as a *Matrix2D* message type and the response is a *StatusMsg* message which informs the client that the saving was complete.

The other implemented service is *CalculatePath.* The client sends the start and end point coordinates to the server, which then calculates the optimal path according to the A* algorithm and its current map. The server converts the points on the path as Point objects and responds to the client with an array of them.



```
krystofh@TimelineX:~/catkin_ws$ rossrv info SetOccupancyGrid
[robot_navigation_project/SetOccupancyGrid]:
robot_navigation_project/Matrix2D map
  int32 width
  int32 height
  int32[] cells
---
robot_navigation_project/StatusMsg status
  bool complete
  string status
```



```
krystofh@TimelineX:~/catkin_ws$ rossrv info CalculatePath
[robot_navigation_project/CalculatePath]:
int32 start_x
int32 start_y
int32 end_x
int32 end_y
---
geometry_msgs/Point[] path
  float64 x
  float64 y
  float64 z
```

/worlds

This folder contains the *empty_world.world* file used to launch the Gazebo simulation with the ground plane, light source and the map.

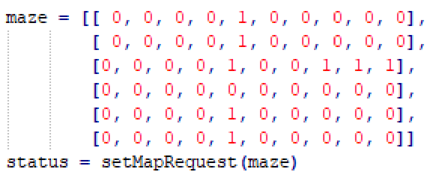
/src

Src is the source folder for the python scripts. These include the path planner server and client, as well as the A* algorithm. Besides, in the subfolder */movement_functions*, some basic scripts for the robot control in Gazebo are introduced (as described later). The client and the server have already been described. For the A* algorithm implementation in Python, example code from an internet tutorial [1]was used with some additional changes. This included a correction of the condition to skip the cell in the open list and the closed list and proceed with the next sucessor cell. Because the algorithm was already tested in the self-written Matlab version and the aim of the ROS part was to get to know its environment, my focus during the work lied in obtaining this knowledge instead of writing the Python code from zero.

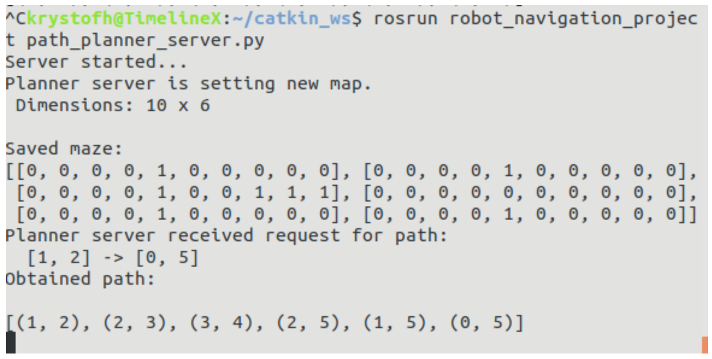
Path client – server communication example

Here is a small example of a map-saving and path-calculation request to the server. The server *path_planner.py* is started and waits for the request. If it arrives, server prints the obtained map and saves it as a global variable. If a path request is sent, the server prints the start and end points and calls the A* algorithm. Then it returns the path as an array of Points to the client. The small map for the demonstration looks as follows.

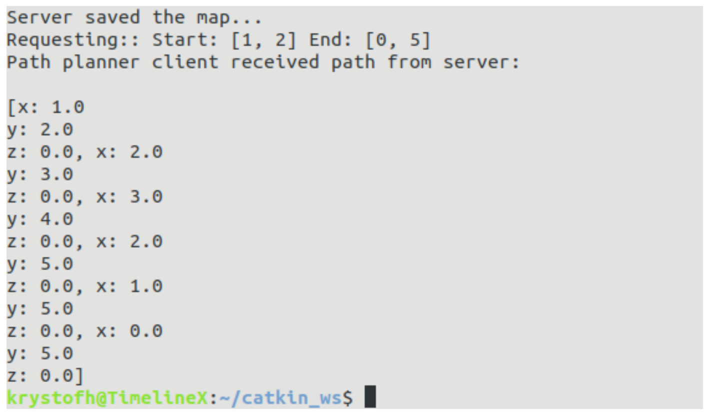
[1] https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2 viewed lastly on 10.2.2020



```
maze = [[ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [ 0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 1, 1, 1],
        [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
        [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]]
status = setMapRequest(maze)
```



```
^Ckrystofh@TimelineX:~/catkin_ws$ rosrun robot_navigation_projec
t path_planner_server.py
Server started...
Planner server is setting new map.
 Dimensions: 10 x 6

Saved maze:
[[0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 1, 1, 1], [0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 0, 0, 0, 0, 0], [0, 0, 0, 0, 1, 0, 0, 0, 0, 0]]
Planner server received request for path:
  [1, 2] -> [0, 5]
Obtained path:

[(1, 2), (2, 3), (3, 4), (2, 5), (1, 5), (0, 5)]
```



```
Server saved the map...
Requesting:: Start: [1, 2] End: [0, 5]
Path planner client received path from server:

[x: 1.0
y: 2.0
z: 0.0, x: 2.0
y: 3.0
z: 0.0, x: 3.0
y: 4.0
z: 0.0, x: 2.0
y: 5.0
z: 0.0, x: 1.0
y: 5.0
z: 0.0, x: 0.0
y: 5.0
z: 0.0]
krystofh@TimelineX:~/catkin_ws$
```

Robot movement control & simulation

Basic idea

The initial idea was to accomplish a full simulation of the robot's movement through the map after the path is obtained. Due to time reasons, this was not possible. However, the idea should be described, as well as the individual movement functions which have been implemented.

The idea of the robot movement control was following. Once the path is obtained from the algorithm, the server should also include the conversion of the px coordinates into m. Here the resolution known from the map building process (see above) is necessary. The path in units of meters should then be processed by a robot controller (client). It should go point by point through the path and send requests to the PID controller (server) to move the robot to the specific point. This node would constantly overwatch the odometry of the robot and once the given point is achieved (with a finite distance error), it would send back a status message to the robot controller client about the success of the operation. The controller would then proceed with the following point and so on until the end is reached.

Challenges and problems

There are several challenges with this approach. First obvious problem is the size of the robot. As the robot has probably a bigger size than 1 px, the obstacles in the map would need to be made significantly bigger in the occupancy grid in order to avoid them. Also, there is no notion of the time in which the robot finishes the movement (see difference between trajectory and path above).

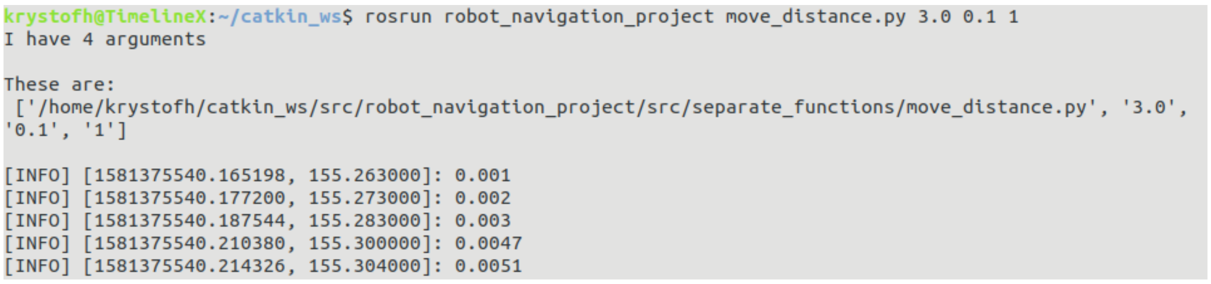
Besides, the controller controls only the position and not the orientation of the robot. The orientation is only controlled implicitly which leads to unknown trajectories between the points. If the distance between the points is too big (for example 1 px cell bigger than the robot model), the robot could crash into obstacles while turning and moving from one point to the other. Another difficulty is fine-tuning the PID controller. As there are almost infinite possibilities for the combination of the parameters, it would be necessary to spend a lot of time until achieving the optimal results. So all in all, it's clear that calculating an optimal path between two points of a map is one thing and successfully implementing the robot movement is a different one. In fact, the path planning would have to be accomplished by other parts of the system such as a local planner for collision avoidance, trajectory generator for the particular robot (path to trajectory conversion taking the constraints into account) etc.

Basic movements demonstration

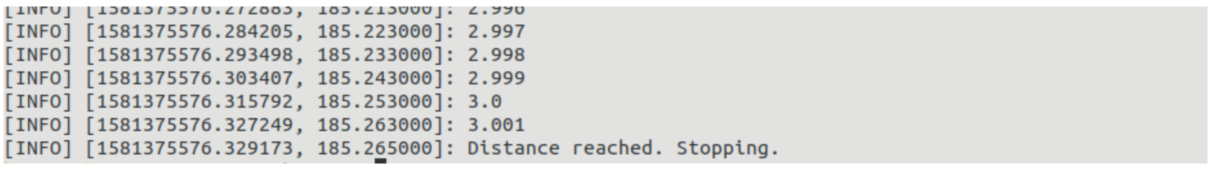
As a preparation for the full robot navigation and control, some basic move functions were implemented. This includes functions for moving forward a given distance (and at a given speed), rotating an angle or moving to a given point on the map. The individual functions and their parameters should be briefly described. Their tests can be seen in the videos folder.

move_distance.py

Moves the robot by a given distance (m), at a given speed (m/s) and forward (1) or backward (0). It subscribes to the */odom* topic containing the odometry and publishes a *Twist()* message with *velocity_message.linear.x* = *abs(speed)* at a rate of 100 Hz specified by *rospy.Rate().*



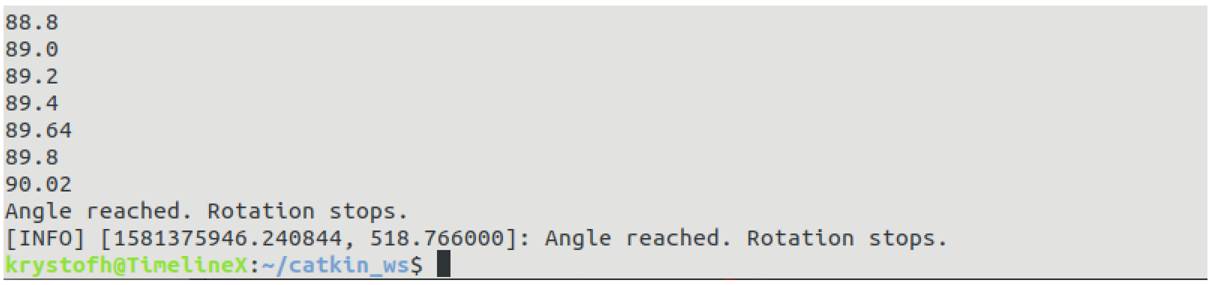
```
krystofh@TimelineX:~/catkin_ws$ rosrun robot_navigation_project move_distance.py 3.0 0.1 1
I have 4 arguments

These are:
 ['/home/krystofh/catkin_ws/src/robot_navigation_project/src/separate_functions/move_distance.py', '3.0',
'0.1', '1']

[INFO] [1581375540.165198, 155.263000]: 0.001
[INFO] [1581375540.177200, 155.273000]: 0.002
[INFO] [1581375540.187544, 155.283000]: 0.003
[INFO] [1581375540.210380, 155.300000]: 0.0047
[INFO] [1581375540.214326, 155.304000]: 0.0051
```



```
[INFO] [1581375576.284205, 185.223000]: 2.997
[INFO] [1581375576.293498, 185.233000]: 2.998
[INFO] [1581375576.303407, 185.243000]: 2.999
[INFO] [1581375576.315792, 185.253000]: 3.0
[INFO] [1581375576.327249, 185.263000]: 3.001
[INFO] [1581375576.329173, 185.265000]: Distance reached. Stopping.
```

rotate.py

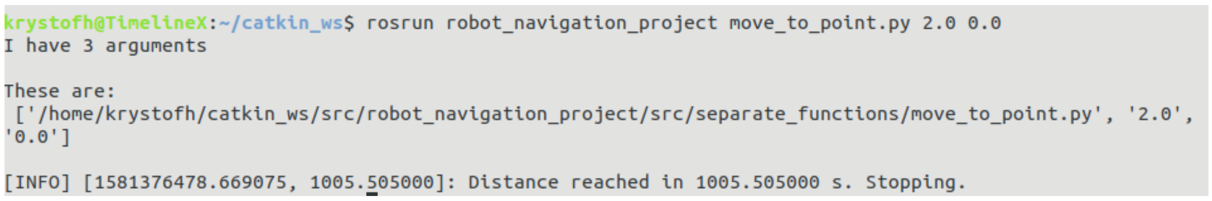
This script rotates the robot and takes the following arguments: angle (deg), angular speed (deg/s) and rotation direction (1=clockwise, 0=counterclockwise). The outcome of the rotation operation is shown below or in the videos.



```
88.8
89.0
89.2
89.4
89.64
89.8
90.02
Angle reached. Rotation stops.
[INFO] [1581375946.240844, 518.766000]: Angle reached. Rotation stops.
krystofh@TimelineX:~/catkin_ws$
```

move_to_point.py

Moves the robot to a given point. It takes its x and y coordinates (floats) as arguments. The script contains two PIDs (currently only with the proportional part) – one for the distance and the other for the angle control. In order to control the angle, the odometry callback has to know the yaw. The orientation in the */odom* message is defined in quaternions so in order to work with radians, it's necessary to do a conversion using the *euler_from_quaternion* function of *tf.transformations* package. The robot first uses the rotation control to rotate until a given precision (here 10 deg) is achieved. Then it starts also the forward motion whilst keeping the angle error minimal. Important is also to saturate the output of the controller so that the robot does not drive the robot with speeds extending the capabilities of the motors.



```
krystofh@TimelineX:~/catkin_ws$ rosrun robot_navigation_project move_to_point.py 2.0 0.0
I have 3 arguments

These are:
 ['/home/krystofh/catkin_ws/src/robot_navigation_project/src/separate_functions/move_to_point.py', '2.0',
'0.0']

[INFO] [1581376478.669075, 1005.505000]: Distance reached in 1005.505000 s. Stopping.
```

Further scripts:

Besides the described functions, a script for moving forward at a given speed and for stopping the robot was written.

Conclusion

Although a full simulation of the robot's navigation through the map was not achieved, it was possible to show the functionality of the A* algorithm as well as some basic concepts of ROS – the communication between nodes, server requests as well as Gazebo map building, launching worlds, moving the robot inside the environment etc. All these are building blocks for future projects to come and I am glad that I could learn how to use them now.

The work on the project turned out to be very demanding. The ROS ecosystem offers infinite number of possibilities and for a beginner it´s very difficult to choose in which direction and with which tools to proceed. The time needed to get the knowledge of the principles of communication between nodes, simulations in Gazebo, an overview of the message types etc. is considerable. Besides, for someone not particularly fit in Python programming (although with good knowledge of Matlab), the language represents another difficulty to overcome. Although in the end, it wasn´t possible to bring the project as far as I wanted, I´ve learned a lot about the principles of robot navigation, gained an insight into ROS and am well prepared for the next project in the field which will be my thesis next semester.

The ROS Part of the project is made freely available on Github:

https://github.com/krystofh/robot_navigation_project

Sources

- http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html
- https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2
- https://www.growingwiththeweb.com/2012/06/a-pathfinding-algorithm.html
- https://www.geeksforgeeks.org/a-search-algorithm/
- TU Dresden, Course „Mobile Robot Systems“, Fakultät Elektrotechnik und Informationstechnik · Institut für Automatisierungstechnik · Prof. K. Janschek
- https://wiki.ros.org/catkin/workspaces
- https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2