POLITECNICO DI TORINO

DAUIN - DEPARTMENT OF CONTROL AND COMPUTER ENGINEERING

Master of Science in Mechatronic Engineering

Master's degree thesis

Model-based design of a path planning algorithm for a motorized wheelchair

University Supervisor:
Prof. Alessandro Rizzo

Company Supervisor:
Eng. Luca Russotti

Candidate:
Giuseppe Mansoori Fard

October 2018

Ad maiora

Abstract

Motorized wheelchairs help people with disabilities in everyday life and often they are not able to control the wheelchair itself.
This project is placed in the autonomous guide of a wheelchair in an indoor environment, developed at Teoresi S.p.A, where the planning of the path represents the first step to move autonomously an item, considered as a mobile robot.
Several pathfinding algorithms exist, but the most appropriate is the A* algorithm, given its low computational cost and its high efficiency.
Using the innovative method, called "Model-Based Design", the code is no longer written "by hand", but is modelled, tested and auto-generated. This allows to reduce developing and testing times, saving half or more time needed to achieve the main goal.
Nevertheless, the handwritten code is not abandoned, since the system requires to interface with other modules to move the wheelchair independently.
Thanks to this kind of implementation, we are able to plan the path of a wheelchair safely.
The developed code can run independently of the final application, as long as it is considered as a differential mobile robot.

Acknowledgements

I would like to express my deep gratitude to Prof. Alessandro Rizzo and Eng. Luca Russotti, respectively my university supervisor at Politecnico di Torino and my company supervisor at Teoresi S.p.A., for their patient guidance, enthusiastic encouragement and useful critiques of this project work.

I would also like to thank Teoresi S.p.A. and Eng. Giuseppe Lo Giudice, for giving me this incredible opportunity to work in a fast growing, professional and friendly company.

Thanks to my colleagues and friends at Politecnico di Torino. In particular thanks to Antonio, for having being there and having tolerated me throughout these university years and more.

Thanks to Eleonora, for always supporting me without limiting my ambitions and being able to make mature me, giving that touch of kindness, unpredictability and instinct that was lacking in my everyday life. A unique, unrepeatable, special and rare girl.

Thanks to all my friends, from long-standing to new friendships. In particular thanks to my acquired brothers and sisters Alberto, Alessandro, Chiara, Dimitri, Federica B., Federica D., Francesca C., Francesca F., Gianluca, Isabella, Lorenzo, Luisa, Manuel, Marco, Mattia, Sara, Stefano and Virginia. All my friends, even those I have not mentioned, have had a crucial weight in achieving this result. Thank you for sharing with me the most important experiences throughout these years.

Thanks to my family, where degree of kinship is not important, for the support and love demonstrated over the years.

Thanks to my brother Bijean, for having been my model since I was a kid. Try to be more like you prompted me to do my best.

Finally, my biggest thank goes to my parents for giving me everything, bringing you all sorts of problem without feeling me the weight, betting on me, where nobody would ever bet. I am proud to be your son.

Ringraziamenti

Desidero innanzitutto ringraziare il Prof. Alessandro Rizzo e l'Ing. Luca Russotti, rispettivamente il mio relatore al Politecnico di Torino e il mio supervisore aziendale alla Teoresi S.p.A., per la loro pazienza, i loro incoraggiamenti e le loro critiche costruttive che hanno portato a terminare questo progetto.

Vorrei ringraziare la Teoresi S.p.A. e l'Ing. Giuseppe Lo Giudice, per avermi dato questa incredibile opportunit*á* di lavoro in un'azienda in crescita, professionale e cordiale.

Grazie ai miei colleghi e amici del Politecnico Di Torino. In particolare grazie ad Antonio per essermi stato vicino e avermi sopportato in questi anni universitari e non solo.

Grazie ad Eleonora per avermi sempre sostenuto senza limitare le mie ambizioni ed essere stata in grado di farmi crescere come persona, dandomi quel tocco di amorevolezza, imprevidibilit*á* ed istintivit*á* che mancava nella mia quotidianit*á*. Una ragazza unica, irripetibile, speciale e rara.

Un grazie a tutti i miei amici, dalle amicizie di vecchia data a quelle nuove. In particolare un grazie ai miei fratelli e alle mie sorelle acquisiti: Alberto, Alessandro, Chiara, Dimitri, Federica B, Federica D., Francesca C., Francesca F., Gianluca, Isabella, Lorenzo, Luisa, Manuel, Marco, Mattia, Sara, Stefano e Virginia. Tutti i miei amici, anche quelli non citati, hanno avuto un peso determinante nel conseguimento di questo risultato. Grazie per aver condiviso con me in questi anni le esperienze pi*ú* importanti.

Un grazie alla mia famiglia, dove il grado di parentela non *é* importante, e a chi ci ha lasciato per il sostegno e l'affetto dimostratomi negli anni.

Un grazie a mio fratello Bijean per essere stato il mio modello fin da quando ero bambino. Cercare di assomigliargli mi ha spinto a dare sempre di pi*ú*.

Infine, il mio pi*ú* grosso ringraziamento va ai miei genitori per non avermi mai fatto mancare niente, portandosi appresso ogni tipo di problema senza mai farmene sentire il peso, scommettendo su di me, laddove nessuno avrebbe mai scommesso. Sono orgoglioso di essere vostro figlio.

Contents

Chapter 1

Introduction

This work refers to the development of a light transport system for people with disabilities or with mobility difficulties, in collaboration with Teoresi S.p.A, an international company head-quartered in Turin (Italy) that is actively taking part in engineering services, particularly focused on analytical modelling, simulation, controlling and software development.
Wheelchairs are a fact of life for many people with disabilities. They enable those with disabilities to enjoy life in a way that might not be possible without access to these mobility assistants.
Unfortunately, people may have some disabilities that limit the controllability of a traditional electric wheelchair and therefore preclude any possibility of an autonomous movement.
The project investigates and integrates the best mobility technologies to create an advanced motorized wheelchair that helps people to walk around an indoor environment.
The Robotic Wheel-Chair supports autonomous driving with obstacle avoidance algorithm and it ensures the stability of the system in case of emergency situations through algorithms of tip-over preventions.
The aim of this thesis project is to investigate on the best algorithm for path planning in an indoor environment.
Guidelines to develop and design this project are:

- low cost;
- high portability;
- easily set up;
- low computational effort and time;
- high testability.

By using planimetry building, the path planning could be performed with A* (star) 2D - Algorithm: it is a computer algorithm that is widely used in pathfinding and graph traversal and it is the process of plotting an efficiently directed path between multiple points, called "nodes". It enjoys widespread use due to its performance and accuracy.
When writing a code, it is easy to stumble on multiple errors. Manually coded errors could be reduced by using modelling tools and methods, saving time to fix them.
Modelling is essential in complex systems and it is a new way to develop embedded code.
In order to apply the path planning, the algorithm code has been written using the Model-Based Design Approach (MBD): it is an innovative method, applied in designing embedded software, used in many motion control, industrial, automation, aerospace and automotive applications.
Concerning the Model-Based Design approach, the algorithm has to be modelled and it is possible to automatically generate code for embedded deployment and create test benches for system verification, avoiding the introduction of manually coded errors. Once the code is automatically generated and tested, it can be loaded and integrated on a "single-board computer"(such as the Raspberry Pi 2 Model B for the project), locally connected to the wheelchair.
The Model-Based Design approach with MATLAB[1] and Simulink improves product quality and reduce development time by 50% or more[2].
The Trajectory Planning module is prone to errors when connected to encoders due to the so-called "Dead Reckoning" and it needs some modules to fix the problem. In navigation, dead reckoning is a common method used to predict the position of a mobile robot by internal sensors, generally inertial sensors, and control variables, such as the encoder. The position estimation obtained by dead reckoning has an acceptable accuracy over the short term, but it has unbounded errors over the long term [3].
The Trajectory planning module contains *Pathfinding* module and needs links to other blocks, such as "Obstacle avoidance" module and "Localization" module, to work correctly in the environment, reducing the errors over the long term.
Master's thesis structure is organized in:

- **Chapter 2**: importance of mobile robots world, briefly explaining their characteristics, and existing motorized wheelchairs are presented, showing some features of past applications;

[1] https://it.mathworks.com/

[2] data on https://www.mathworks.com/solutions/model-based-design.html

- **Chapter 3**: mapping and path planning algorithms for mobile robots, especially in indoor environment, are shown, explaining their advantages and drawbacks for each approach, method or algorithm;
- **Chapter 4**: Model-Based Design world is introduced, explaining step by step the design procedures and its advantages;
- **Chapter 5**: heart of the thesis project, where the map is adapted, the model implemented, the code auto-generated and then implemented on board. Some results are shown;
- **Chapter 6**: final chapter that reports some comments and how it is possible to improve the wheelchair features;

Chapter 2

Mobile robots and motorized wheelchair

2.1 Hints of mobile robots

Definition of **Robot***: A reprogrammable, multifunctional manipulator designed to move material, parts, tools, or specialized devices through various programmed functions for the performance of a variety of tasks.*
According to "Robotic Institute of America",1979

Robotics is the study and design of robots. The essential component of a robot is the mechanical system, composed by a locomotion apparatus (wheels, crawlers,...) and a manipulation apparatus (end-effectors,artificial hands,...)[30][5].
The term *robot* can be used in different context for different applications[30]:

- industrial robots;
- service robots;
- humanoid robots;
- ...

The capability to exert an action is provided by the actuation system, while the capability for perception is estrused to the sensory system, which can be composed of proprioceptive sensors (to acquire data of the internal status of the mechanical system) or exteroceptive sensors (to acquire data of the external status of the environment[28].
Due to their mechanical structure, robots can be classified as:

- robot manipulator, with a fixed base;

- mobile robots, with a mobile base.

A *mobile robot* is considered as an autonomous robot, capable to move and act independently from a human supervisor, able to work in indoor or outdoor environment[28][4]. The environment can be assumed to be:

- structured, partially structured or unstructured[1];
- totally known, partially known or unknown.

A crucial prerequisite for a mobile robot is to perform tasks in the environment is the capability to autonomously navigate[25].
Navigation requires three main capabilities:

- Localization: the robot is able to determine its pose with respect to a given reference frame;
- Mapping: it has to build a consistent and meaningful representation of the environment;
- Path planning: the robot has to plan the motion strategy to reach a given target pose.

From a mechanical point of view, a mobile robot consists of one or more rigid bodies equipped with a locomotion system. This description [10] includes the following two main classes of mobile robots:

- Wheeled mobile robots: typically consist of a rigid body (base or chassis) and a system of wheels which provide motion with respect to the ground;
- Legged mobile robots: they are made of multiple rigid bodies, interconnected by prismatic joints or revolute joints.

Only wheeled vehicles are considered from now on, as they represent mobile robots actually used in applications.
The basic mechanical element of a mobile robot is the wheel. Three types of mobile robot's wheels exist[10]:

- The fixed wheel: it can rotate about an axis that goes through the center of the wheel, orthogonally to the wheel plane. The wheel is rigidly attached to the chassis, whose orientation with respect to the wheel is therefore constant;

[1] a structured environment is when it is known the type and the geometric characteristic of the environment

- The steerable wheel: it has two axes of rotation. The first is the same as a fixed wheel, while the second is vertical and goes through the center of the wheel, in order to allow the wheel to change its orientation with respect to the chassis;
- The caster wheel: it has two axes of rotation, but the vertical axis does not pass through the center of the wheel, shifted by a constant offset. This type of wheel is used to provide a supporting point for static balance without affecting the mobility of the base.

One of the most popular mobile robot is the *"differential-drive"*(DD) mobile robot [Figure2.1], where there are two fixed wheels with a common axis of rotation, and one or more caster wheels (typically smaller) whose function is to keep the robot statically balanced[10][27].

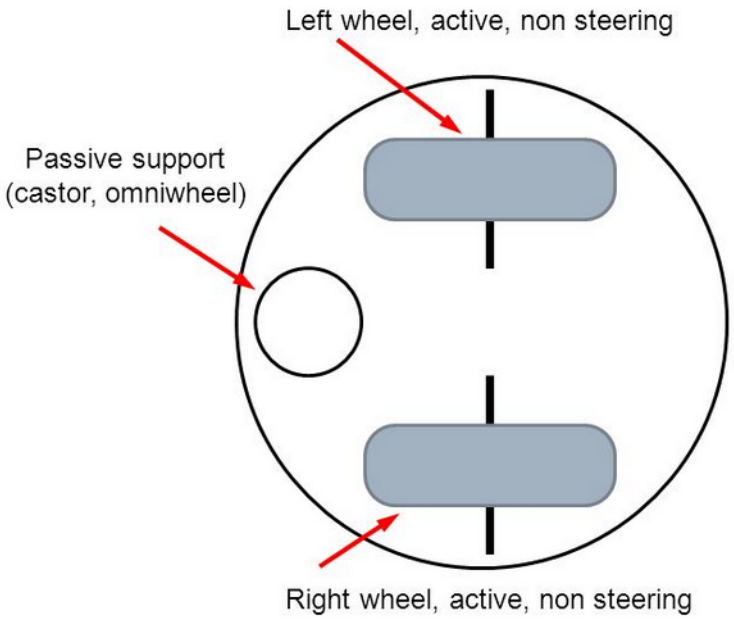


Figure 2.1: Differential drive rover[27]

Wheel axes meet in a common point called Instantaneous Curvature Centre (ICC). If an ICC does not exist, wheel motion occurs with slippage. The two fixed wheels are separately controlled, in that different values of angular velocity may be arbitrarily imposed, while the caster wheel is passive. These two wheels have velocities[27]:

$$V_l = \omega(\rho - \frac{l}{2}) \tag{2.1}$$

$$V_r = \omega(\rho + \frac{l}{2}) \tag{2.2}$$

where l is the baseline, ρ is the curvature radius, Vl and Vr the left and the right longitudinal velocities and ω the angular velocities.
Knowing Vl and Vr, it is possible to obtain the linear and the angular velocity of the robot as follows:

$$v = \frac{V_r + V_l}{2} \tag{2.3}$$

$$\omega = \frac{V_r - V_l}{2} \tag{2.4}$$

Since $R = \rho$ and $L = l$, if $\omega \neq 0$, wheel velocities and the curvatur radius are defined as:

$$V_l = \omega(R - \frac{L}{2}) \tag{2.5}$$

$$V_r = \omega(R + \frac{L}{2}) \tag{2.6}$$

$$R = \frac{L}{2} \cdot \frac{V_r + V_l}{V_r - V_l} \tag{2.7}$$

In a 2D environment, position and orientation are merged in a single vector called pose as follows:

$$P = \begin{pmatrix} x \\ y \\ \theta \end{pmatrix}$$

where x and y are the Cartesian coordinates and θ is the orientation of the rover.
Thanks to these equations, after some mathematical steps, direct kinematics equations are obtained as:

$$x(t) = \frac{l(V_r + V_l)}{2(V_r - V_l)} \sin \frac{V_r - V_l}{l} t \tag{2.8}$$

$$y(t) = \frac{l(V_r + V_l)}{2(V_r - V_l)} \cos \frac{V_r - V_l}{l} t + \frac{l(V_r + V_l)}{2(V_r - V_l)} \tag{2.9}$$

$$\theta(t) = \frac{V_r - V_l}{l} t \tag{2.10}$$

Instead the inverse kinematic problem is under-constrained and it has infinitely solutions, therefore is necessary to apply additional constraints, usually provided by path planning algorithms[27].
Resolvers, potentiometer and encoders may be used to measure angular displacements as position transducers, due to their robustness, reliability and accurancy[10].
Encoder types and characteristics are summarized in the following since the encoder has been used for the project.
There are two types of encoder[4]:

- Absolute encoder: consists of an optical-glass disk on which concentric circle tracks are disposed; it is able to obtain the absolute position of a axes in movement, thus includes the initial position of the axis at power up. Absoute encoders are useful for application in which accurancy is crucial.

- Incremental encoder: it has a wider use than absolute encoders, since they are simpler and cheaper. As the absolute one, the incremental encoder consists of an optical disk on which two tracks are disposed. Incremental encoders are also called "relative" encoders since they are abe to obtain a differential displacement without knowing the initial absolute position.

Based on their technology, encoders may be also classified in[4]:

- Rotary;

- Magnetic;

- Capacitive;

- Inductive.

- Optical [Figure2.2];

Dwelling on optical encoder, it has its own signal processing electronics inside the case, which provides direct digital position measurements to be interfaced with the control computer. When an external circuitry is employed, velocity measurements can be reconstructed from position calculations[10]. Thus, if a pulse is generated at each transition, a velocity measurement can be obtained through three possible procedures:

- by using a voltage-to-frequency converter (analog output);

- by digitally measuring the frequency of the pulse train;

- by digitally measuring the sampling time of the pulse train.

Mobile robots are equipped with incremental encoders that measure the rotation of the wheels, hence indirectly the pose of the vehicle with respect to a fixed frame, therefore localization procedures are used to estimate in real time the robot configuration[10].
Two possible ways are used to obtain a mobile robot's pose:

- GPS(Global Positioning System): it is able to determine the vehicle's pose but it is subject to a huge estimation error, therefore a bad ratio costs over accuracy. It is used for outdoor applications;

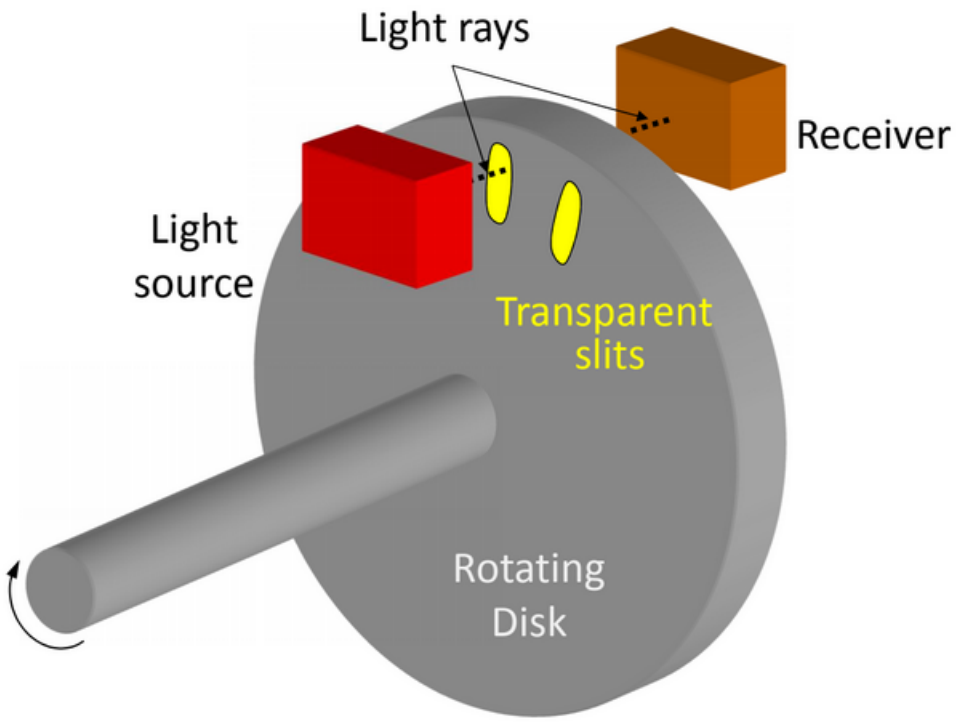


Figure 2.2: Optical encoder[26]

- Odometry: it is cheaper than GPS solution. It is used to estimate the wheeled robot's position relative to a starting location, based on the wheel's revolution measurements. Odometry is based on the assumption that wheels turn linearly with respect to the ground. Unfortunately, it is sensitive to errors due to the integration of velocity measurements over time to give position estimations, caused by different disturbances. In general it is possible to distinguish two different categories[4]
  - systematic errors: imperfect knowledge of the wheels geometry (for instance, wheels diameter can vary in time depending on weather conditions and usage), wheels misalignment, sensors with limit sampling frequency and resolution;
  - not-systematic errors: not perfectly flat floors, slippery ground, too high accelerations, too fast curves, caster wheels,...

It is important to distinguish systematic and not-systematic errors in order to minimize their effect on the odometry calculation. For instance, systematic errors are particulary damaging and have serious effects on the calculation because they tend to accumulate themselves time by time, leading to an increasing error during the operation. In fact, taking into account the indoor environment case, where

the floor is usually flat, systematic errors represent the main error. Nevertheless systematic errors can be compansated while not-systematic errors could have some difficulties due to their randomness[10].
A general algorithm to solve this problem is the Bayes Filter. This approach is based on the Markov assumption such that future states are independent on past ones, given the current state.
In general the prediction phase of the Bayes filter cannot be computed in closed form because the filter needs to be iterated at each time step, thus the effective implementation of Bayes filter requires further assumptions on the representation of the probability densities involved in the estimation[25]:

- using a Kalman Filter: the probability densities are supposed to be multivariate Gaussians;
- using Particle Filters: the probability densities are approximated with a set of weighted particles.

Focusing on the Kalman Filter Localization, it is possible to demonstrate that if the dynamic system describing the robot motion is linear, then:

$$\begin{cases} p_t = A_t\, p_{t-1} + B_t\, u_t + \omega_t \longrightarrow \texttt{PROCESS MODEL} \\ z_t = C_t\, p_t + v_t \longrightarrow \texttt{MEASUREMENT MODEL} \end{cases}$$

where p_t is the position of the robot in the 2D environment at time t, u_t is control vector at time t, the noises ω_t and v_t are zero mean Gaussian noise, with covariance matrices R_t and Q_t respectively, the initial state is $p_0 \sim N(\mu_0, \sum_0)$ then the state at time t will be distributed according to $p_t \sim N(\mu_t, \sum_t)$ for all the future time step. If we use only odometry for localization, the localization error accumulates over time, thus, in order to refine the pose estimate, we have to take into account not only $\hat{p_t^-}$, $\sum_t^-$ e $\bar{u_t}$ (command and a-priori estimate) but also the measurements acquired from the sensors $\bar{z_t}$. For each time instance we have:

PREDICTION[2]:

$$\begin{cases} \hat{p_t^-} = A_t \hat{p_{t-1}^-} + B_t\, \bar{u_t} \\ \sum_t^- = A_t\, \sum_{t-1}\, A_t^T + Q_t \end{cases}$$

UPDATE:[3]

$$\begin{cases} K_t = \sum_t^- C_t^T (C_t \sum_t^- C_t^T + Q_t)^{-1} \\ \hat{p_t} = \hat{p_t^-} + K_t(\bar{z_t} - C_t \hat{p_t^-}) \\ \sum_t = (I - K_t C_t) \sum_t^- \end{cases}$$

Actually in most of the real world problems, both the process and measurement

[2]A priori estimate

[3]A posteriori estimate

models are expressed by non-linear equations, therefore Extended Kalman Filter (EKF) is introduced. It is similar to Kalman Filter but, for filter prediction and update, the models are linearised and the approach becomes:

PREDICTION:

$$\begin{cases} A_t = \frac{\partial f}{\partial p_t}|_{p_t=\hat{p_t}^-} \\ \hat{p_t}^- = f(\hat{p_{t-1}}^-, \bar{u_t}) \\ \sum_t{}^- = A_t \ \sum_{t-1} \ A_t^T + Q_t \end{cases}$$

UPDATE:

$$\begin{cases} C_t = \frac{\partial h}{\partial p_t}|_{p_t=\hat{p_t}^-} \\ K_t = \sum_t{}^- C_t^T (C_t \sum_t{}^- C_t^T + Q_t)^{-1} \\ \hat{p_t} = \hat{p_t}^- + K_t(\bar{z_t} - h(\hat{p_t}^-)) \\ \sum_t = (I - K_t C_t) \sum_t{}^- \end{cases}$$

Extended Kalman Filter is an effective solution for mobile robot localization, but linearization can cause filter divergence if the original problem is hightly non-linear and it can be hard to precisely model the available informations[25]. Concerning the usage of Particle filters for the estimation of mobile robot pose (also called Monte Carlo Localization), the Bayes Filter can be reformulated as:

- Prediction: generate a new particle set given motion model and controls applied
- Update: assign to each particle an importance weight according to sensor measurements
- Re-sampling: re-sample particle based on weights

It can be stated that determining the pose of the robot in a given reference frame using odometry and sensor measurements allows to localizate the mobile robot[2]. Regarding to the Mapping and Path Planning capabilities, they will be discussed in "Chapter 3", explaining how it has been developed the Path Planning algorithm.

2.2 Motorized wheelchairs

"My disabilities have not been a significant handicap in my field, which is theoretical physics. Indeed, they have helped me in a way by shielding me from lecturing and administrative work that I would otherwise have been involved in."

Stephen Hawking

A lot of motorized wheelchairs have been developed and designed over the years, starting from a mobile robot scheme. Changes have been made in terms of comfort, security, performance and manufacturing [9][4][12]. In other words, a motorized wheelchair is simply a mobile robot on which it has been put a seat and a control device (such as a joystick most frequently).
Electric wheelchairs are items to overcome limits that can occur during life. From a scientific point of view, electric wheelchairs have the following characteristics[4]:

- it is able to move in a indoor or outdoor environment;
- it can run at different speeds, without damaging the user;
- it must have a robust structure on which sensors and devices are mounted;
- it must have a well-dimensioned power supply in order to use in a better way all the features that the electric wheelchair can provide as long as possible.

Sometimes, people may have some disabilities that limit the controllability of a traditional motorized wheelchair and therefore preclude any possibility of an autonomous control. For this reason, it is possible to distinguish electric wheelchair in two main categories: *semi-autonomous* and *autonomous wheelchairs*.
Semi-autonomous wheelchair main goal is to support the user while driving the wheelchair, such as implementing functions to avoid obstacles and approaching objects. The choice of the path to be followed is left to the discretion of the user. Different devices, like voice control or joystick, are useful to that purpose[4].
Nevertheless, autonomous wheelchairs have functionalities and features very similar to those usually implemented in the autonomous mobile robots. The user specifies a destination target and the control system cares to plan the path and run it. To carry out this type of activity, it is important to represent the map of the place where the wheelchair should work or have a recognition system of the places reachable by the wheelchair.
Integration between obstacle avoidance system and mapping is complicated and sometimes the wheelchair can offer the possibility to re-schedule the path planning, after an unexpected obstacle appearance[9][4].

2.2.1 Existing projects

First prototypes have been developed since '80. Thanks to the innovation technology, progress constantly adds features and comfort on assistive mobile robots.
Several smart wheelchairs exist, as shown in the following list (a more complete lists can be found in [9][4]):

- **SMART ALEC** (Stanford, USA, 1980): it was the first semi-autonomous wheelchair. It was a advanced wheelchair with ultrasonic distance sensors and wheel encoders;

- **Madarasz wheelchair** (Arizona state university, USA, 1986)[13]: it was basically the first autonomous wheelchair for disabled people. It was equipped with sonars and was able to navigate in corridor and hall environments, using a proper language. Odometry was based on optical encoders, but it was not used for robot's localization;

- **NavChair** (University of Michigan, USA, 1994)[31]: it was developed for people that suffered from different sorts of disabilities, such as blind people or people with a limited vision. It shared control decisions with the driver, reducing the motor and cognitive requirements of the user. The obstacle avoidance module tends to maintain the given path, taking decision in a safe mode when approaching obstacles;

- **Wheelesley** (MIT, USA, 1995)[22]: it was designed for people that are unable to drive a wheelchair with a normal joystick. It was equipped with infra-red proximity sensors and ultrasonic range sensors (such as Hall effect sensors). It was possible to steer the wheelchair with an eye tracking interface;

- **Rhombus** (MIT, USA, 1997)[19]: Rhombus means "Reconfigurable Holonomic Omnidirectional Mobile Bed with Unified Seating". It was a powered wheelchair with omni-directional drive;

- **Tin Man I and Tin Man II** (KIPR[4] - KISS Institute for Practical Robotics) [16]: TinMans were built from already existing commercial power wheelchairs. Tin Man I provides human guided with obstacle avoidance and it is able to move to a desired position. The Tin Man II was an evolution of Tim Man I, adding capabilities like backtracking, wall following at a constant distance, doorway passing, docking to objects, backup. Encoders were mounted on both wheelchairs to improve the accuracy;

[4]http://en.wikipedia.org/wiki/KISS Principle, Oct, 2006.

- **The Smart Wheelchair** (University of Edinburgh, Scotland, 1995)[23]: it had bump sensors to sense obstacles and it implemented a line following algorithm, for driving through doors and between rooms, therefore it was programmed to stop itself, back-off and turn to change direction;

- **CCPWNS** (University of Notre Dame, USA, 1994-2000)[14] it stands to "Computer Controlled Power Wheelchair Navigation System". It allowed to reproduce paths previously investigated by the system, but a obstacle avoidance module was not implemented;

- **VAHM** (University of Metz, France, 1992-2004)[21]: it is the acronym for "auto-Véhicule Autonome pour Handicapé Moteurnomous". It was built based on a mobile robot integrated with a seat. The vehicle had two semi-autonomous behaviours, such as wall following and obstacle avoidance, Thanks to a grid based method for path planning in a indoor environment, the vehicle is able to wall following at a constant distance and avoid obstacles. It was built a second prototype, with same functionalities as the previous one, based on a commercial wheelchair on which odometric and ultrasonic sensor were integrated and mounted;

- **HaWCos** (University of Siegen, Germany, 2002)[15]: it is the acronym for "Hands-free" Wheelchair Control System, that allows a wheelchair to be controlled without using the hands. It is suitable for people with very severe disabilities about manual dexterity. HaWCos is able to use muscle contractions and transform them as input signals;

- **SWCS** (University of Pittsburgh, CMU, USA, 2004)[18]: it stands for "Smart Wheelchair Component System". SWCS is a modular based system for commercial wheelchairs. It relies on sonar sensors, bump sensors, infra-red sensors and the so-called "Drop-off" sensor, which is able to detects obstacles like holes, stairs and pits. Navigation is possible thanks to the navigation system software, which runs on a computer interfaced through analog switches and joysticks;

- **Victoria** (Aachen University, Germany, 2004)[17]: it is based on the Storm wheelchair, made by "Invacare GmbH". It is equipped with two computers to support manual control, supervised control and autonomous control. A touch screen, where a processed stereo image can be viewed through cameras, is used for selecting object that a manipulator can grasp. Several functions are accessible through the touch screen, such as direct control, comfort and home automation functions;

- **The Walking Wheelchair** (University of Castilla-La Mancha, Spain, 2006)[20] it is equipped with four wheels and four legs. The wheels are mounted at the

end of each leg. The legs goal is to make possible climbing stairs for the vehicle. The project focuses on the mechanical components and design construction;

- **Permobil C350**[5]: it is a commercial wheelchair used by Stephen Hawking. Its main features are a tilt, recline, elevating power seat, adjustable suspension firmness and rear wheel drive. According to Hawking's website, a computer and speech synthesizer were mounter on his wheelchair and, using a software called "Assistive Context Aware Toolkit" (ACAT[6]), it was able to communicate by moving his cheek [Figure2.3].

(a) Permobile C350 basic (b) Stephen Hawking's advanced wheelchair

Figure 2.3: Permobile C350

[5]http://www.permobil.com/en-GB/English/C/Products/C350-Corpus/

[6]https://newsroom.intel.com/news-releases/new-intel-created-system-offers-professor-stephen-hawking-ability-to-better-communicate-with-the-world/

2.2.2 Wheelchair project and added devices

Through this paragraph, it is possible to know the basic structure used for the project, on which several devices and components have been added in order to create the autonomous motorized wheelchair. The wheelchair developed by Teoresi S.p.A will have the following main functionalities:

- **autonomous driving (collision avoidance)**;
- **map tracking with Dead Reckoning**;
- **path definition with voice recognition**;
- **tip-over prevention**;
- **IoT (lift activation service elevator)**;

This thesis project is the first step in order to achieve those features, using the *Pathfinding* (PF) module.
Used mechanical and electronic components are reported as follows:

- **787 BARIATRIC Solid bariatric armchair - SURACE wheelchairs**
  It is the basic mechanical structure on which sensors, encoders and electronic components were added. Main characteristics [7] of this wheelchair [Figure2.4] are:
  - **Design specifications**:
    - Diameter big wheels: 60cm each;
    - Diameter small wheels: 20cm each;
    - Height: 123 cm (total);
    - Width: 85 cm;
    - Weight capacity: 150 kg;
    - Adjustable width between armrest;
    - Padded chair;
    - Adjustable and removable footrests;
    - Adjustable with gas springs backrest;
    - All the parts of the wheelchair are specifically designed without sharp edges or corners which can create an hazard when moving the patient.

[7] http://www.surace.it

Figure 2.4: SURACE wheelchair

- **For safety and reliability**:
  - Self-extinguishing upholstery, in accordance with FIAT and ISO standards;
  - Spring, rack or gas piston adjustable leg rests;
  - Shock-proof specially rounded and contoured footrests
  - Double ball bearings on front and rear wheels;
  - Nylon container guides and frame protections against the damage caused by urine;
  - Non-toxic, physiologically safe, polyester-based polyurethane foam padding;
  - Double layer chrome plating (up to 50 micron thick) against wear.

- **For comfort**:
  - High-density, squash-proof (40-60 g/litre), printed and cast padding, in accordance with Surace specifications;
  - Adjustable seat width;
  - Integral, anatomical, side turning leg rest cushions with ABS counter frame;
  - Continuous position reclining backrests with a special original Surace quick tipping system;
  - Original Surace designed nylon integral side clothes guards;
  - Unique pattern fabrics;
  - Perfect trimming with nylon and ABS plastic components (backrest panel, armrest panels, leg rest cushion panels).
- **Brusched DC Motor - Proton Wira Power Window Compatible**[8]

Figure 2.5: Friction wheels and DC motor
Proton Wira Power Window Compatible

It is the component that transform a wheelchair to a motorized wheelchair, through friction wheels [Figure2.5]. Main features are:

- Voltage Rating: 12VDC;

[8]https://www.robotshop.com/en/power-window-motor-with-coupling-right.html#Specifications

- No load Speed: (85±15)RPM;
- Rated Speed: (60±15)RPM;
- Current (No Load): <5A;
- Rated Current (Load): <15A;
- Stall Current (Locked): <28A at 12V;
- Rated Torque: 30Kgcm (2.9Nm);
- Stall Torque (Locked): (100±15)Kgcm (∼10 Nm);
- Perfect fit with 5 inches robot wheel;
- Weight: 696g.

- **VNH5019 Motor Driver Carrier**[9]

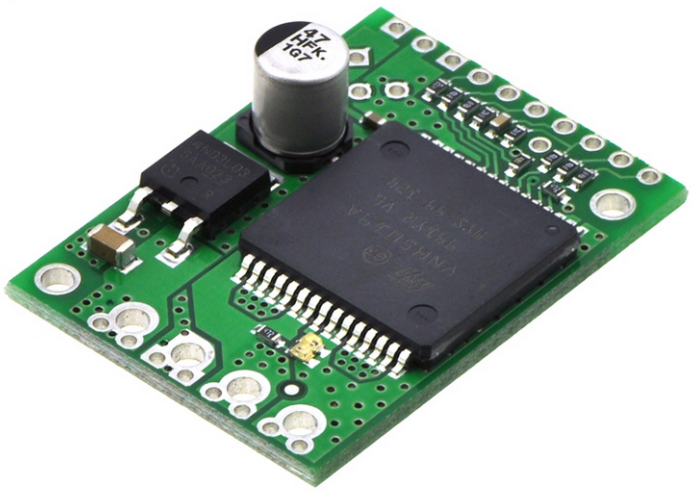


Figure 2.6: ST VNH5019

It is a general-purpose motor driver [Figure2.6]. This carrier board motor driver IC operates from 5.5 to 24 V and it can deliver a continuous 12A (30A peak). It works with 2.5V to 5V logic levels, supports ultrasonic PWM and features current sense feedback[10]. It has protection against reverse-voltage, over-voltage, under-voltage, over-temperature, and over-current.

- **Long distance ranging Time-of-Flight sensors**

[9]https://www.pololu.com/product/1451

[10]an analog voltage proportional to the motor current

Figure 2.7: ST VL53L1X Time-of-Flight sensor

It has been used for the obstacle avoidance module six time-of-flight (ToF) sensors in order to recognize the presence of fixed obstacles (such as tables or walls) and negative slops, and to avoid tip-over situations. ToFs are considered proximity sensors. They measure the time-of-flight, i.e. the time from the emission to the return of the signal. In constrast to Lidars, the measurement is performed for each point of the image[24]. The simplest version of a ToF camera uses light pulses and the distance resolution is larger than the Lidar's one. Advantages of that solution are[24]:

- The whole system is very compact;
- It is very easy to extract the distance information out of the output signals of the ToF sensor;
- They are able to measure the distances within a complete scene with one shot.

Drawbacks can be summarized as[24]:

- Presence of a background light that cause a limitation on the dynamic range of the pixels;
- If several ToF are working at the same time, interference between cameras can occur.

Regarding the project, ST VL53L1X ToF sensors [Figure2.7] are used due to their characteristics, suitable for this project application specifications.

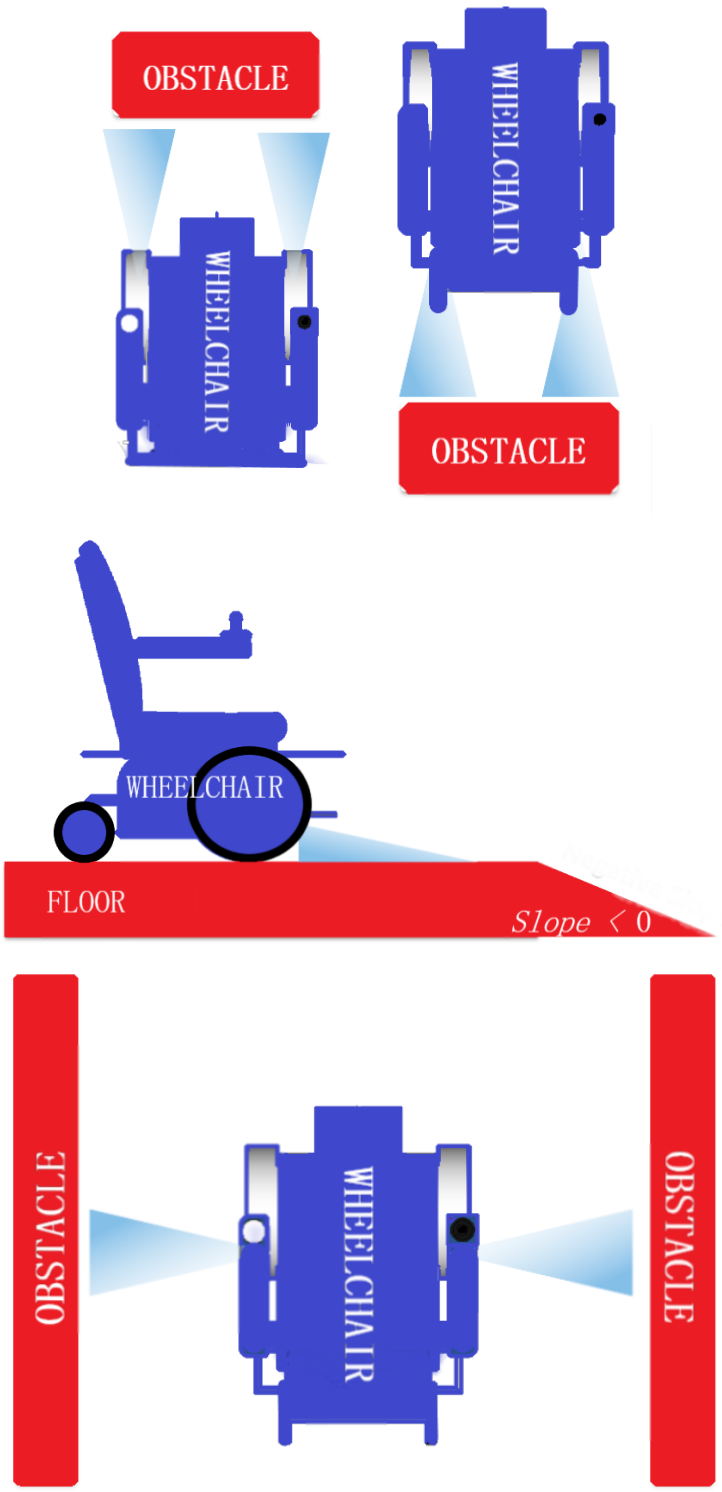


Figure 2.8: Front, rear, lateral ToF positions

- **Encoder AVAGO AEDB-9140**[11]
  It is a three channel optical incremental encoder modules with code wheel.

Figure 2.9: AEDB-9140 Series encoder

This project needs two of them (one per each wheel). Characteritics of this encoder are [Figure2.9]:

  - Two channel quadrature output with index pulse;
  - Resolution from 100CPR to 500CPR (Counts Per Revolution);
  - Low cost;
  - Easy to mount;
  - No signal adjustment required;
  - Small size;
  - Operating temperature: -10°C to 85°C;
  - TTL Compatible;
  - Single 5V supply.

- **Bds Battery AgmLong Life 12V 40Ah T9**[12]

[11] https://it.rs-online.com/web/p/encoder-rotativi/7967806/

[12] https://www.selcoitalia.it/12-volt/129-bds-battery-agm-long-life-12v-40ah-t9.html?gclid=Cj0KCQiA5t7UBRDaARIsAOreQthWlOXH59zpyXr5l4-Nn1hbPgtLJRksFUylV6ME-_Feu0JfH4-snCQaAvAyEALw_wcB

Figure 2.10: Bds Battery AgmLong Life 12v 40ah T9

It has been chosen a long life battery autonomy. Characteristics of this battery are [Figure2.10]:

- Dimensions: 197 x 165 x 170 x 170;
- Weight: 13,3 kg;
- Autonomy: 2h.

- **STM32F746 HIGH-PERFORMANCE MCU**

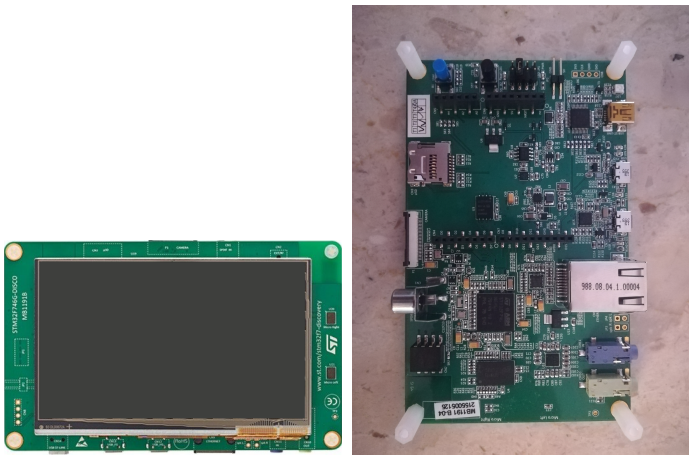


Figure 2.11: Display and pinout of STM32F746 MCU

It is the MCU board used to perform different functionalities for the motorized wheelchair, in particular it runs the "*Obstacle Avoidance*" module, which communicate through serial communication with the *Pathfinding* module developed for this master's thesis project. STM32F746 Discovery's datasheet is

available on website[13] and few characteristics and features are reported below:

- LCD parallel interface, 8080/6800 modes;
- General-purpose DMA: 16-stream DMA controller with FIFOs and burst support;
- Core: ARM 32-bit Cortex-M7 CPU with FPU, adaptive real-time accelerator (ART Accelerator) and L1-cache: 4KB data cache and 4KB instruction cache, allowing 0-wait state execution from embedded Flash memory and external memories, frequency up to 216 MHz, MPU, 462 DMIPS/2.14 DMIPS/MHz(Dhrystone 2.1), and DSP instructions;
- Up to 1MB of Flash memory:
  - 1024 bytes of OTP memory;
  - SRAM: 320KB (including 64KB of data TCM RAM for critical real-time data) + 16KB of instruction TCM RAM (for critical real-time routines) + 4KB of backup SRAM (available in the lowest power modes);
  - Flexible external memory controller with up to 32-bit data bus: SRAM, PSRAM, SDRAM/LPSDR SDRAM, NOR/NAND memories;
- clock, reset and supply management;
- 2×12-bit D/A converters;
- Up to 18 timers;
- debug mode;
- up to 168 I/O ports with interrupt capability;
- up to 25 communication interfaces:
  - up to 4× I2C interfaces (SMBus/PMBus);
  - up to 4 USARTs/4 UARTs (27 Mbit/s, ISO7816 interface, LIN, IrDA, modem control);
  - up to 6 SPIs (up to 50 Mbit/s), 3 with muxed simplex I2S for audio class accuracy via internal audio PLL or external clock;
  - 2 × SAIs (serial audio interface);
  - 2 × CANs (2.0B active) and SDMMC interface;
  - SPDIFRX interface;
  - HDMI-CEC;

[13] https://www.st.com/en/microcontrollers/stm32f746ng.html

  - Advanced connectivity:
    * USB 2.0 full-speed device/host/OTG controller with on-chip PHY;
    * USB 2.0 high-speed/full-speed device/host/OTG controller with dedicated DMA, on-chip full-speed PHY and ULPI;
    * 10/100 Ethernet MAC with dedicated DMA: supports IEEE 1588v2 hardware, MII/RMII.

- **Raspberry Pi Model B+ V1 2**

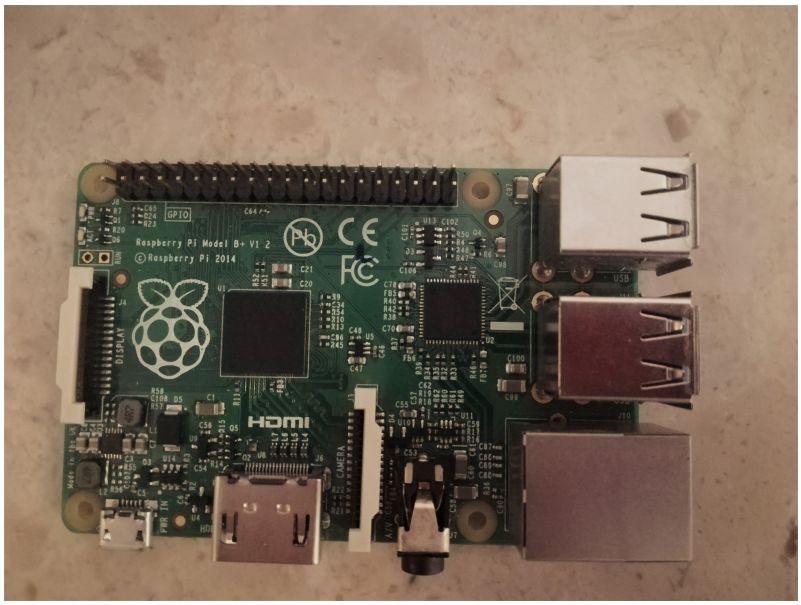


Figure 2.12: Raspberry Pi Model B+ V1 2

It is the main component used in this thesis project, where the *Pathfinding* (or called "*Path planning*"), that will become *Trajectory Planning*, module runs. Its main characteristics are reported below[14]

- 900MHz quad-core ARM Cortex-A7 CPU
- 1GB RAM;
- 100 Base Ethernet;
- 4 USB ports;
- 40 GPIO pins;
- Full HDMI port;

[14]https://www.raspberrypi.org/products/raspberry-pi-2-model-b/

- Camera interface (CSI);
- Display interface (DSI);
- Micro SD card slot;
- VideoCore IV 3D graphics core.

2.3 Chapter's salient and important points

In this chapter it has been briefly explained the mobile robot's world: the base of the thesis project. Several already existing projects were described to show the evolution of the electric wheelchair's environment, in order to introduce the project. The wheelchair in development at Teoresi S.p.A has several functionalities (such as obstacle avoidance, tip-over prevention, path following,exc.) and this thesis project deals with the Path Planning module that locally runs on the Raspberry Pi B+ V1 2 board, mounted on the wheelchair, interfaced with the STM32F746 board, where the Obstacle Avoidance and other modules are developed by other colleagues. In the following chapter, it is described the mapping adaptations and pathfinding algorithms, which can be implemented on board.

Chapter 3

Mapping and path planning

3.1 Mapping definition and representations

Definition of **Mapping***: is the task of building a consistent representation of the environment assuming that we know robot poses.*[25]

Mapping is one of the competencies of truly autonomous robots[7]. Depending on the scenario that has to be represented, different mapping representation exist:

- **Digital elevation map**: based on Digital Elevation Model (DEM), a mapping database that represents the relief of a surface between points of known elevation. A rectangular digital elevation model grid can be obtained by interpolating known elevation data from sources, such as ground surveys and photogrammetric data captured[1];
- **Point-cloud representation**: is a set of data points in space, generally produced by 3D scanners, able to measure a large number of points on the external surfaces of objects around them. It is used for metrology, animation, rendering, quality inspection and mass customization[2];
- **Landmark-based representation**: it is a stochastic map that contains a probabilistic description of the position of salient features of the scenario, such as doors, corners and objects. It enjoys widespread in the application of mobile robotics in a indoor environment. This representation does not provide information for navigation, such as obstacle avoidance and wall following and it needs data association to distinguish features, but it is a compact and efficient world representation, thus it does not occupy too much memory[25];

[1] https://www.caliper.com/glossary/what-is-a-digital-elevation-model-dem.htm

[2] https://en.wikipedia.org/wiki/Point_cloud

- **Occupancy grid map** [Figure3.1][3] is a grid in which each cell contains the probability that the cell can be occupied.
  Cells can be free, occupied and unknown cells, thus it is an intuitive model of the environment, suitable for navigation requirements. Several path planning algorithms (such as A*,B*,D*,etc.) deal with grid maps. As the *landmark-based* mapping representation, it is used for indoor scenarios for mobile robots, but it can cause a huge memory occupation[25];

Figure 3.1: Occupancy grid map

As it has already been discussed in "Chapter 1", localization is based on the knowledge of the environment. Since mapping requires the knowledge of the robot pose, the mobile robot has to localize itself during the acquisition of the map. It can be said that mapping and localization are not independent: this problem is called "*Simultaneous Localization and Mapping*" (SLAM).
The SLAM problem relates to building a map of the environment while simultaneously determining the robot's position relative to the map of the environment[29].
Concerning *occupancy grid* maps, since it is suitable for path planning algorithm (the goal of this thesis project), they address the problem of generating consistent maps from noisy data, under the assumption that the robot's pose is known.
The basic idea is to represent the map as a field of random variables, disposed in a uniformly spaced grid. Each random variable is binary, corresponding to a free or busy[4] location[29].
Occupancy grid mapping algorithms implement approximate posterior estimation for those random variables.
A lot of SLAM's techniques do not generate maps fit for path planning and navigation.
Occupancy grid maps are often used after solving the SLAM problem[29].

[3] https://slideplayer.com/slide/5036518/

[4] with an obstacle

3.2 Path planning and trajectory planning definitions

Planning trajectories and paths date back the Greek myths.
Ariadne's thread, named for the legend of Ariadne in Greek's mythology, is the solving of a problem with a physical maze, through an exhaustive application of logic to all available routes. Applying Ariadne's thread to a problem means to create and keep track's records and exhausted options at all times. The purpose of the record is to permit backtracking, thus reversing earlier decisions and trying alternatives. Until it is possible to get out of the maze, there is a chance to go back, mark the path as already visited and take another path. Once a solution is found, it is possible to reconstruct the path following the thread. In this way, Ariadne helped Theseus to overcome the Minotaur and save the potential sacrificial victims.

Path planning is the name given to algorithms and procedures in order to support services related to path construction and management, thus including optimal path generation, real-time obstacle avoidance, vision-based exploration, on the based of priori informations (using maps) and real-time informations (using distance, vision and touch sensors)[27].
A path planning algorithm, in its simplest form, requires the definition of an initial pose and the desired (target) final pose on a map.
It is significant to underline that path and trajectory have two different meanings[27] [Figure3.2]:

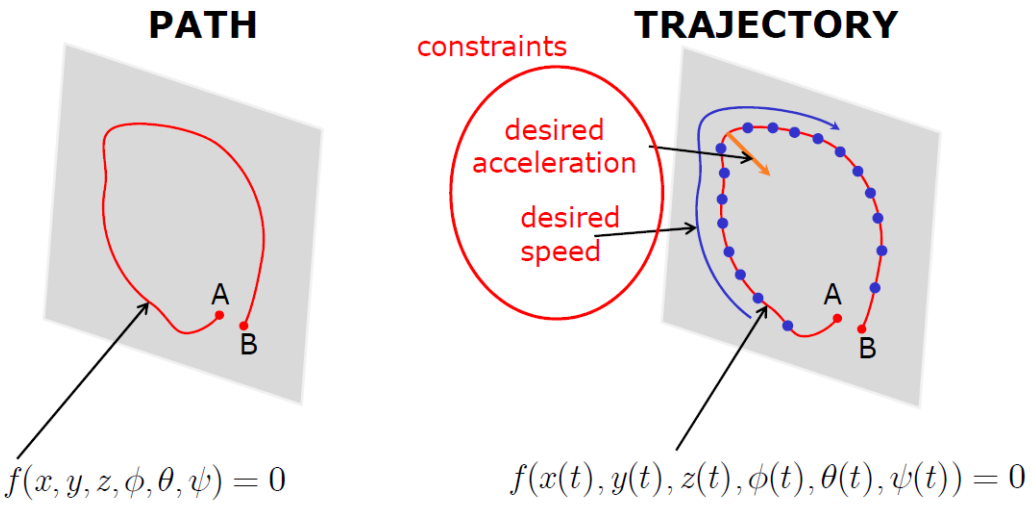


Figure 3.2: Difference between path and trajectory[27]

- *Path* is the geometrical description of the desired set of points, which represent poses that the robot has to follow to reach the final target pose;
- *Trajectory* is the path and the time law merged together required to follow the path from the starting point to the end point, applying kinematic and dynamic constraints (such as desired acceleration and speed) to the desired path.

Industrial robots often operate at the fastest possible speed because of the economic impact of high throughput on a factory line. The dynamics and kinematics of their motions are significant, further complicating path planning and execution. Conversely, some mobile robots operate at such low speeds that dynamics are rarely considered during path planning, further simplifying the mobile robot instantiation of the problem[11].
The task of planning is to find a collision-free path among a ensemble of static and dynamic obstacles.
Path planning can be distinguished in[29]:

- **Offline algorithms**: they are static and calculated a priori to execution;
- **Online algorithms**: they require algorithms that meet real-time constraints to enable path re-calculations and/or adaptations during the robot motions in order to react to and interact with dynamic environments. This means that a robot has to move along a path that has not necessarily been computed completely, because it may change during the movement.

Since trajectory includes velocities, accelerations, and/or jerks along a path, it can be considered as a path with kinematic and dynamic constraints.
A common method is computing trajectories for a priori specified paths, which satisfy determined criterions, such as, for instance, minimum executions time, maximum distance to workspace boundaries and energy consumption criterions.
It is possible to distinguish trajectory planning methods as well as path planning methods[29]:

- **Offline calculated trajectory**: it cannot be influenced during its execution;
- **Online trajectory planning methods**: they can re-calculate and/or adapt robot motions behaviour during the movement.

Re-calculation and/or adaptation for online methods are used to improve the accuracy and have a better utilization of currently available dynamics, in reaction and relation to a dynamic environment or events sensed by the sensors, because the robot acts in a (partly) unknown and dynamic environment.
Without reacting, the planning effort will not pay off because the robot will never physically reach its goal, while without planning, the reacting effort cannot guide

the overall robot behavior to reach a distant goal and, again, the robot will never reach its goal[11].
Furthermore, it can further distinguish between:

- **One-dimensional (1-D)** and **multi-dimensional (N-D)**[5] **trajectories**;
- **Single-point** and **multi-point trajectories**[6].

. Due to trajectory planning algorithms, several constraints may be applied. Constraints for trajectory planners can be manifold:

- **Kinematic**: maximum velocities, accelerations, jerks and workspace space limits;
- **Dynamic**: maximum torques and joint and actuator forces;
- **Geometric**: static and dynamic obstacle avoidance in the workspace;
- **Temporal**: reaching a state at a given time or time interval.

These and other constraints may have to be taken into account at the same time. Depending on the robot and the task, additional optimization criteria may be considered: time-optimality, minimum-jerk, maximum distance to workspace boundaries, minimum energy[29].
Once the trajectory planner has been developed, it can be used as reference signal generator to provide inputs to the control problem, which includes the controller and the actuators, feedbacked by the sensing informations. In the [Figure3.3], it can be seen how the various blocks and concepts discussed in the previous chapters (such as the usage of sensors for obstacle avoidance, localization and mapping blocks and path/trajectory planning) are reported and used to control a mobile robot[7]. In the next sub-chapter, it will be discussed some pathfinding algorithms, including the one used for this project.

[5] where N can be N=2,3,4,..

[6] multi-point trajectories typically relate to a path

[7] https://slideplayer.com/slide/1667514/7/images/7/General+Control+Scheme+for+Mobile+Robo+Systems.jpg

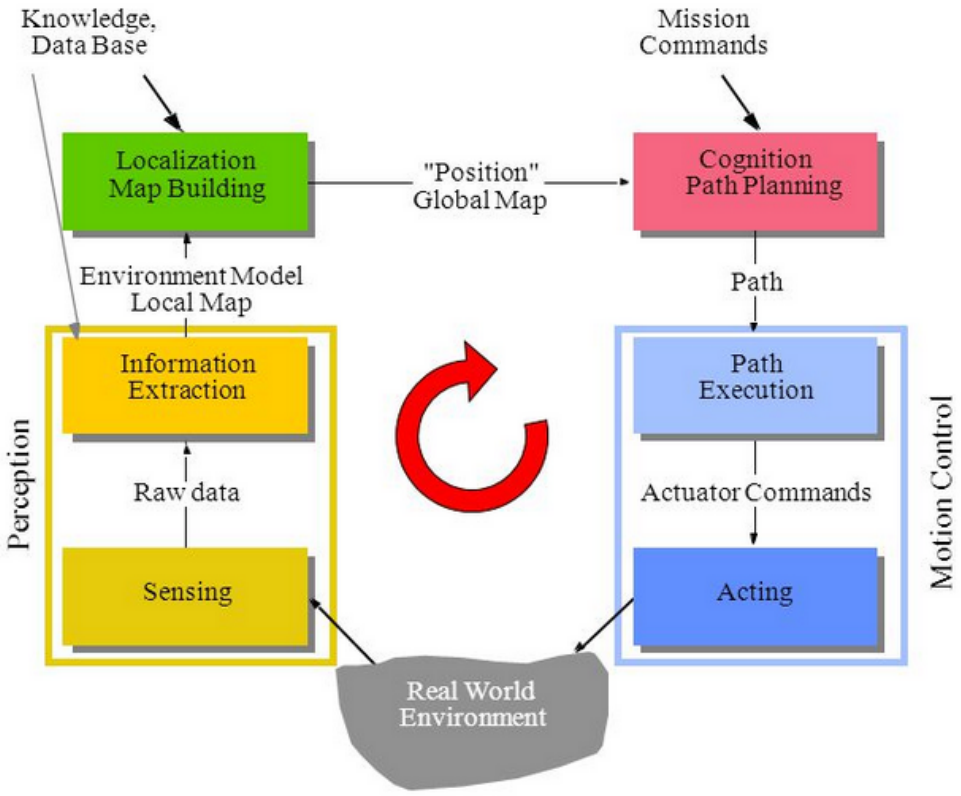


Figure 3.3: Control problem using trajectory planner

3.3 Pathfinding algorithms

Pathfinding is the plotting, by a computer application, of the shortest route between two points[8]. It is the way in which it is possible to solve mazes.
Pathfinding is closely related to the shortest path problem, within graph theory, to identify the path, following some criteria such as the shortest, cheapest and fastest one between two points.
A pathfinding method searches a graph by starting at one vertex (the starting node) and exploring adjacent nodes until the destination node (target node) is reached. Although graph searching methods, such as a breadth-first search, would find a path if enough time is given, they exist other methods which explore the graph that tend to reach the destination as soon as possible.
Main problems of pathfinding are:

- find a path between two nodes in a graph;
- find the optimal shortest path.

Basic algorithms, such as breadth-first and depth-first search, address the first problem by exhausting all possibilities.
Starting from the given node, they iterate over all potential paths until they reach the target node. Complexity of those algorithms is O(|V|+|E|) or linear time, where V is the number of nodes and E is the number of edges between nodes.
A more complicated problem is finding the optimal path. The exhaustive approach is known as the Bellman-Ford algorithm, which has complexity of O(|V||E|) or quadratic time.
Nevertheless, it is not necessary to examine all possible paths to find the optimal one. Algorithms such as A*,D* and Dijkstra's algorithm strategically eliminate paths.
By eliminating impossible paths, these algorithms can achieve time complexities of O(|E|log(|V|)). However, better time complexities can be attained by algorithms which can pre-process the graph to attain better performance[9]. Algorithms used in pathfinding are reported as follows.

3.3.1 Breadth-first search (BFS)

Breadth First Search explores equally in all directions. This algorithm is useful not only for path finding, but also for procedural map analysis.

[8] https://enen.wikipedia.org/wiki/Pathfinding

[9] data available on https://medium.com/omarelgabrys-blog/path-finding-algorithms-f65a8902eb40

The key idea is to keep track of an expanding ring called the frontier[37][40]. Until the frontier is empty, the following steps have to be performed iteratevely:

- Pick and remove a location from the frontier;
- Expand it by looking its neighbours.

It starts at the root node and explores all of its children in the next level (the neighbors) before moving to each children and then exploring the children of the root children and so on, until neighbors are finished and the frontier is empty[37][40]. It is important to underline that any neighbours not visited yet have to be added to the frontier and the visited set, and a queue is used to perform the BFS.
The pseudo-code is reported below[40]:

1. Add the root node to the queue and mark it as visited;
2. Loop on the queue until it is not empty;
   (a) Get and remove the node at the top of the queue (called "current" node);
   (b) for every child of the current node not visited yet , do:
      i. Mark it as visited;
      ii. Check if it is the destination (target) node. If it is the goal node, then return it;
      iii. Otherwise insert it into the queue;
3. If the queue is empty, the target node was not found.

This loop is the essence of the following graph search algorithms, including A*(A-star), but the loop does not actually re-construct the paths and it is not able to find the shortest path.
BFS only tells how to explore everything on the map; for this reason it can be used for map analysis.
BFS is the simplest pathfinding algorithm. It works not only on grids but on any sort of graph structure.

3.3.2 Deep-first search (DFS)

Similarily to the BFS, the DFS starts at the root but it explores one of its children's sub-tree. Once it has explored this sub-tree it moves to the next child's sub-tree and iteratevely repeats the steps[37][40]. Instead of queue, it uses stack or recursion to perform the DFS.
The pseudo-code is reported below[40]:

Recursive steps:

1. The current node is the root node;
2. Mark the current node as visited;
3. Check if the current node is the goal node. If yes, then return it;
4. Iterate over children nodes of the current node. Do:
   (a) Check if a child node has not visited yet;
   (b) If yes, mark it as visited;
   (c) Go recursively to its sub-tree until the goal node is found;
   (d) If the child node has the goal node in its sub-tree, then return it;
5. If the goal node is not found, the goal node is not in the tree.

Iterative steps:

1. Add the root node to the stack;
2. Loop on the stack until it is not empty;
   (a) Get the current node at the top of the stack and mark it as visited, then remove it;
   (b) For every not-visited child of the current node, do:
      i. Check if it is the goal node. If yes, return this child node;
      ii. Otherwise, insert the current node in the stack;
3. If the stack is empty, the target goal is not found.

3.3.3 Dijkstra's algorithm

It is also called Uniform Cost Search and it prioritizes which paths to explore. Instead of exploring all possible paths equally, like BFS and DFS, it is able to find lower cost paths. That is possible assigning lower costs to free spaces and higher costs to obstacles.
When movement costs vary, Dijkstra's algorithm is used instead of BFS and DFS[37][40].

Since tracking movement costs is needed, a new variable "g"(cost so far) is introduced, to keep track of the total movement cost from the start node. The queue is transformed into a priority queue, since it takes into account the movement costs. Since a node may be visited multiple times by different paths with different costs, instead of adding the node to the frontier if the location has never been visited, it will be added if the new path of this node is better than the previous path.
Dijkstra's algorithm tries to find the shortest path from the root node to every node, getting the shortest path from the starting node to the goal[37][40].
The pseudo-code is the following one[40]:

1. Assign distance[v] = INT_MAX $\forall$ nodes[10];
2. Assign distance[v] = 0, because it is the distance from the root node to itself;
3. Add all nodes to a priority queue;
4. Loop on the priority queue until it is not empty:
   (a) In every loop, choose the node with the minimum distance from the root node in the queue. The root node has to be selected first;
   (b) Remove the chosen current node from the queue: vis[current] = true;
   (c) If the chosen current node is the target node, then return it;
   (d) For every child node of the current node, do:
      i. If the child node is not already visited, thus it is not already in the queue, then skip this iteration;
      ii. Assign a temporal variable temp = distance[current] + distance from current to child node;
      iii. If temp < distance[child] then assign dist[child] = temp. A shorter path to child node has been found;
5. If the priority queue is empty, then target node was not found.

[10] *distance[v] is the distance between the root note and every other node*

3.3.4 Heuristic search - Greedy algorithm

Although with BFS and Dijkstra's Algorithm, the frontier expands itself in all directions to find a path to all locations or many locations, a common case is to find a path to only one location. That is possible using the Heuristic search[37][40].
A Greedy algorithm is the most simple case of an heuristic search.
A heuristic function tells how far the current node is to the goal.
In Dijkstra's Algorithm the actual distance from the start is used for the priority queue ordering. Instead, in Greedy algorithm, the estimated distance to the goal is used for the priority queue ordering[37][40].
Greedy is an algorithm which makes a choice based on guesses at each stage. The node with shortest heuristic distance from the goal node will be explored next and the location closest to the goal will be explored first. Those paths found by Greedy are not the shortest, but this algorithm runs faster when there are not a lot of obstacles. In order to fix this problem, instead selecting the node closest to the starting point[37][40], it has to be selected the node closest to the target.
Greedy does not guarantee to find a shortest path. However, it runs quicker than Dijkstra's Algorithm.
The pseudo-code is the following one[40]:

1. Assign distance[v] = INT_MAX ∀ nodes;
2. Assign distance[root] = heuristics(root, goal)[11];
3. Add all nodes to a priority queue;
4. Loop on the priority queue until it is not empty:
   (a) In every loop, choose the node with the minimum heuristic distance from the goal node in the queue. The root node has to be selected first;
   (b) Remove the chosen current node from the queue: vis[current] = true;
   (c) If the chosen current node is the target node, then return it;
   (d) For every child node of the current node, do:
      i. If child node is already visited, then skip this iteration;
      ii. Assign distance[current] = heuristics(current, goal);
      iii. Add child node to the queue;
5. If the priority queue is empty, then target node was not found.

[11] *distance from root node to goal*

3.3.5 A-star algorithm (A*)

Dijkstra's Algorithm works well to find the shortest path, but it wastes time exploring in directions that are not promising. Instead, Greedy algorithm explores in promising directions but it may not find the shortest path [37].
A* is a combination of Dijkstra and Greedy. It uses distance from the root node plus heuristics distance to the goal. The algorithm terminates when we find the goal node [35].
The A* search algorithm is an extension of Dijkstra's algorithm useful for finding the shortest path between two nodes of a graph.
The path may traverse any number of nodes connected by edges and each edge have an associated cost.
The algorithm uses a heuristic (such as Greedy) which associates an estimation of the lowest cost path from this node to the goal node, such that this estimation is never greater than the actual cost. It is important to underline that the algorithm does not assume that all edge costs are the same [39].
A* algorithm follows a path of the lowest known cost, keeping a sorted priority queue of alternate path segments along the way.
Whenever a segment of the path has an higher cost than another path segment, it is able to abandoned the higher-cost path and continue with the lower-cost path instead, and this process continues until the goal is reached[33].
A* uses a best-first search and finds a least-cost path from a given initial node to one goal node, out of one or more possible goals. It uses a distance-plus-cost heuristic function, denoted as $f(x)$, to determine the order in which the search visits its nodes in the tree [32] The distance-plus-cost heuristic is a sum of two functions:

$$f(x) = g(x) + h(x) \tag{3.1}$$

- $g(x)$: a path-cost function, which is the cost from the starting node to the current node;
- $h(x)$: an admissible[12] heuristic estimation of the distance to the goal.

The pseudo-code is reported below:

1. Assign distance[v] = INT_MAX ∀ nodes[13];
2. Assign distance[root] = 0 + heuristic(root, goal) [14]

[12] it must not overestimate the distance to the goal

[13] *distance[v] is the distance from root node + heuristics of every node*

[14] *0 because it is the distance from the root node to itself*

3. Add root node to priority queue;
4. Loop on the queue as long as it is not empty;
   (a) In every loop, choose the node with the minimum distance from the root node in the queue + heuristic distance. The root node is the one selected by first;
   (b) Remove the chosen current node from the queue: vis[current] = true;
   (c) If the chosen current node is the target node, then return it;
   (d) For every child node of the current node, do:
      i. Assign temp = distance(root, current) + distance(current, child) + heuristic(child, goal);
      ii. Assign distance[current] = heuristics(current, goal);
      iii. If temp < distance[child] then assign dist[child] = temp. This denotes a shorter path to child node has been found;
      iv. Add child node to the queue if not already in the queue (thus, it's now marked as not visited again);
5. If the priority queue is empty, then target node was not found.

3.3.6 Other algorithms

Other pathfinding algorithm exist and they can be more performant than A* algorithm in different applications.
Existing pathfinding algorithms are [32]:

- **D* algorithm (D-Star)**: is anyone of the following three related incremental search algorithms:
  - *Original D**: the name D* comes from the term "*Dynamic A**", because the algorithm behaves like A* except that the arc costs can change as the algorithm runs.
    Like Dijkstra's algorithm and A*, D* maintains a list of nodes to be evaluated, known as the "OPEN list". Nodes are marked as having one of the following states[32]:
    * *NEW*: node has never been placed in the OPEN list;
    * *OPEN*: node is currently in the OPEN list;
    * *CLOSE*: node is no longer in the OPEN list;

* *RAISE*: its node cost is higher than the last time it was on the OPEN list;
* *LOWER*: its node cost is lower than the last time it was on the OPEN list.

It works by iteratively selecting a node from the OPEN list and evaluating it and then propagates[15] the node's changes to all of the neighbouring nodes and places them on the OPEN list.
Every expanded node has a back-pointer that refers to the next node leading to the target, and each node knows the exact cost to the target. When the start node is the next node that has to be expanded, the path to the goal can be found by simply following the back-pointers.
When an obstacle is detected along the intended path, all the points that are affected are again placed on the OPEN list, this time marked as "RAISE".
Before beginning a RAISED node, the algorithm checks its neighbors and examines whether it can reduce the node's cost. If not, the RAISE state is propagated to all of the node's childrens, thus nodes which have backpointers to it. These nodes are then evaluated, and the RAISE state passed on, forming a wave. When a RAISED node can be reduced, its backpointer is updated, and passes the LOWER state to its neighbors. These waves of RAISE and LOWER states are the heart of D*[32].
The algorithm only worked on the points which are affected by change of cost. Differently to A*, which follows the path from start node to goal node, D* begins by searching backwards from the goal node[36].

– *Focused D**: is an extension of D* which uses a heuristic to focus the propagation of RAISE and LOWER toward the robot. Only the interested states are updated, in the same way that A* only computes costs for some of the nodes[32].

– *D* Lite*: it implements the same behavior as D* and Focused D*, but is simpler and can be developed in fewer lines of code, hence the name "D* Lite"[32]

- **Iterative deepening A* (IDA*)**: is a variant of the A* algorithm which uses a memory usage lower than in A*. The main difference is that it uses the $f(x)$ function-cost as the next limit and not just an iterated depth. In constrast to A*, it does not remember the current shortest path and costs for all visited nodes but it remembers one single path at a time[32].

[15] this process is called "*expansion*"

- **Simplified Memory-Bounded Algorithm (SMA*)**: is one of the "Memory Bounded" heuristic search that comes under "Informed Search Strategies". The main advantage of this search is that it only makes use of available memory to carry out the search. It is a variant of A* search in which the memory is bounded[32].

- **B* algorithm (B-star)**: is a best-first graph search algorithm that finds the least-cost path from a given initial node to any goal node, out of one or more possible goals.
  The algorithm stores intervals for nodes of the tree as opposed to single point-valued estimation and then leaf nodes of the tree are searched until one of the top level nodes has an interval which is the best one.
  Evaluations of leaf nodes in a B*-tree are intervals rather than single numbers and the interval is supposed to contain the actual value of that node. If all intervals attached to leaf nodes satisfy this property, then B* will identify an optimal path to the target node.
  In complex searches, it might not terminate within practical resource limits, thus the algorithm is normally boosted with artificial termination criteria, in terms of time or memory limits[32].

3.4 Chapter's salient and important points

In this chapter it has been briefly explained mapping and path planning method used in the mobile robot's world. Several designing implementation methods have been described to show their importance and their usage in the applications. Since this project works as a mobile robot in a indoor environment, an occupancy grid map has been used as mapping method and A* algorithm has been used as path planning method. The implementation of map and the path planning will be discussed in "Chapter 5".
It is important to underline that in this thesis work, the trajectory planning is out of the the thesis work. Trajectory planning is the next step of the pathfinding implementation, applying kinematic and dynamic constraints to the pathfinding code, such as simply adding the accelerations and speeds at which the encoders should run to perform a path, passing from point to another one. In the next chapter, the model based design approach (MBD) is discussed to implement the code algorithm and then load on board.

Chapter 4

Model-Based Design (MBD) approach

4.1 MBD introduction and motivations

Definition of **Model***: a simplified or partial representation of reality, defined in order to accomplish a task or to reach an agreement.* [41]

In traditional design process, the design information is usually handled in documents, which are difficult to understand and engineers develop embedded code manually, leading to time consuming and error-prone process[51][50].
Model-Based Design (MBD) approach is the solution to solve this problem.
Starting from the original system, "*Modelling*" is based on the abstraction of the system, generalizing specific features of real object, classifying the objects and then aggregating them into a more complex one that reflects the relevant section of the original system[41].
The *Model-Driven Software Engineering* (MDSE) is based on these principles: abstraction from specific realization technologies, requires modelling languages that do not hold specific concepts of realization technologies (such as Simulink, Stateflow, TargetLink) and permits an automated code generation from abstract models.
Model-Driven Software development offers a significantly more effective approach where models are abstract and formal at the same time.
Abstractness does not stand for vagueness here, but for compactness and a reduction to the essence.
MDSD models have the exact meaning of a program code in the sense that the final implementation can be generated from the model. In this case, models are no longer only documentation, but parts of the software, constituting a decisive factor in increasing both the speed and quality of software development[46].
The system functionality is described using a *platform-independent model* (PIM) and an appropriate *domain-specific language* (DSL), then the platform independent

model is translated automatically to a *platform-specific model* (PSM) that computers can run.
MDSE has the following advantages[46]:

- increase development speed;
- enhance software quality;
- avoid redundancy;
- manage complexity through abstraction;
- portability[1] and interoperability[2].

Model-Based Design (MBD) approach refers to MDSE to facilitate the effort reduction in terms of time, costs and resources. This method allows to integrate formal executable models of system modules that are not yet physically realized with available realizations of other modules.
Analysis of the integrated system by means of validation, verification, and testing is able to detect and prevent problems that could occur during real integration. This means that it reduces significantly efforts in the real integration and test phases, which allows time-efficient determination of the conformance of component realizations with respect to their requirements[47].
Model-Based Design approach is used to improve software quality and reduce design errors, that can be very expansive.
Thanks to MDB, it is possible to save up to 50% or more of development time[3][Figure4.1].

Principles of MBD are[41]:

- Abstraction from specific realization technologies: that requires modelling languages, that do not hold specific concept of realization technologies (e.g. Simulink), and improves portability of the software to new technologies;
- Automated code generation from abstract models: that requires expressive and precise models and increase productivity and efficiency;
- Separate development of application and infrastructure: that increases reusability.

It is important to underline that MBD approach can describe two types of model:

[1] platform-independence of software systems

[2] manufacturer-independence through standardization of software systems

[3] data on https://www.mathworks.com/solutions/model-based-design.html

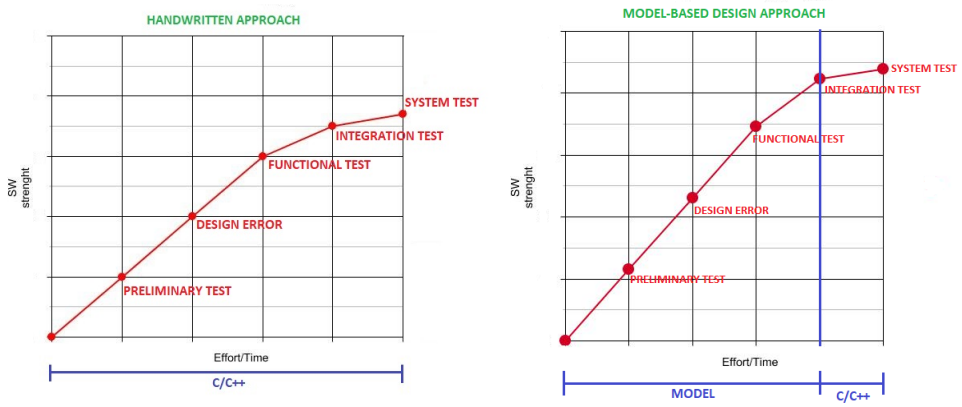


Figure 4.1: Difference between hand-coded approach (left) and MBD approach (right) in terms of effort/time and software strength

- **Static model**: focus on the static aspects of the system in terms of managed data and architecture of the system;
- **Dynamic model**: emphasize the dynamic behaviour of the system by showing the execution.

The Model Based Design is a prominent change in *embedded system* development. It provides a single design platform to optimize overall system design and helps embedded system developers to create simulation models and check whether algorithms will work before the embedded code is written[49]. Embedded software can be developed using MBD for system in aircraft avionics, digital motor controllers, medical devices and much more[51].

4.2 V-shape design flow

In software development, *V-shape* model represents the development process that has to be performed in order to create an embedded code.
The V-shape model is a chain of different steps that should be performed to develop the code. Each step (or phase) has its own associated phase of testing.
In terms of *design flow*, the following steps have to be performed [41] [Figure4.2]:

- **System Requirements**: item definition, defining which are the required functionalities and safety concepts to be implement;
- **System Design**: partitioning the functionalities into sub-modules that has to be designed and then implemented;
- **Software Design**: for each of the subsystem identified, define which are the functions that has to be used to implement a certain functionality;

- **Coding**: specify the single instructions for each function;
- **Software Integration**: integrate the different subsystems together;
- **HW/SW Integration**: put the software into the embedded hardware (execution platform for the application) and validate that everything works together;
- **Vehicle Integration and Calibration**: bring the item into the vehicle and validate if the item is doing what it is supposed to do in used vehicle. The same item may be used in different vehicles and the behaviour has to be adjusted according to certain parameters depending on the vehicle.

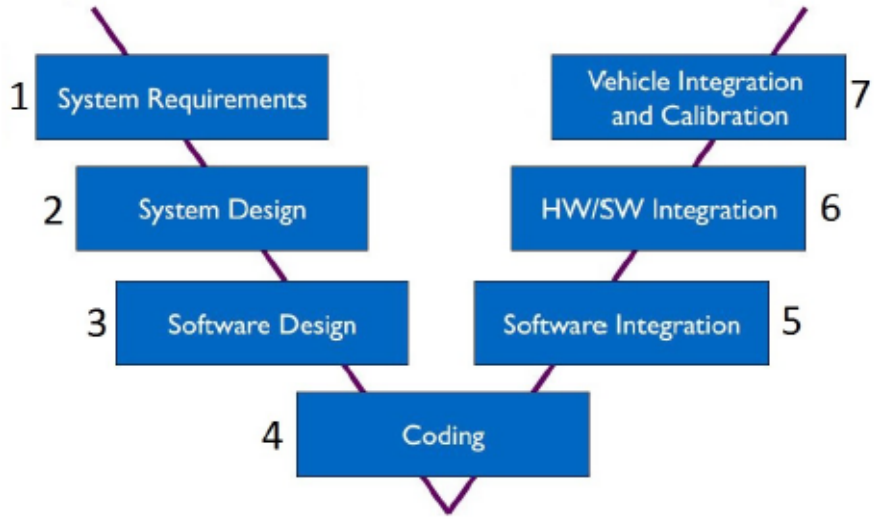


Figure 4.2: V-shape flow

Starting from the complete model of the controller and the plant, the design flow step permits different *types of analysis*[45] [Figure4.2]:

- **Model-in-the-loop test or MIL** (takes care of stages 1, 2 and 3): the model exists entirely in native simulation tool (such as Simulink), it is used for control algorithm development;
- **Optimization and code generation** (takes care of stages 3 and 4): there is the production of HW and/or SW;
- **Software-in-the-loop test or SIL** (takes care of stages 3, 4 and 5): the implementation is co-simulated with the plant model to test its correctness. It is still executed on a PC; part of the model exist in native simulation tool and part as an executable C-code. It is good for testing controller implementation in C-code;

- **Processor-in-the-loop test or PIL** (takes care of stages 5 and 6): the implementation is deployed on the target HW and co-simulated with the plant model to test its correctness. It is not in real time;
- **Hardware-in-the-loop test or HIL** (takes care of stages 6 and 7): part of the model runs in a real-time simulator and part may exists as physical driver (ECU). It is good for testing interactions with HW and real-time performance. The simulation is performed in real time and not in simulation time as before. This is the most expensive step.

After the HIL application, the code can be applied to the final application.
It may happen that sometimes it is useful to run part of the algorithm on another HW (often a *Rapid Prototyping* (RP) hardware) that allows to lighten the principal HW. This is possible with the so-called "*bypass*" technique.
The ECU Software is modified by adding hooks in the function to bypass: when the software execution in the ECU hardware reaches the hook, the bypass function in the RP hardware is invoked. While the bypass function executes in the RP hardware, the ECU hardware waits and when the RP hardware sends the results of the bypass function to the ECU hardware, execution of the ECU software restarts[41].
The V-shape flow is the base to use the MBD approach in a correct way.
Once the control has been modelled, it should be translated into code. This is done in two ways[41]:

- **Algorithm export**: take the controller and translate it into the control algorithm and then manually take the C code and interconnect it with the execution platform. The code is generated only for the platform independent model. In parallel the software platform is developed manually through handwritten code.
  The application must be manually linked with the software platform.
  Algorithm export is used for production code;
- **Full executable**: when the code is generated from the model, automatically the code is generated for the algorithm and for the basic software, merged them together and manual intervention is not needed.
  It may appears redundant instructions, more code and more operations then what it could have using algorithm export.
  Fully executable is used only for rapid prototyping.

4.3 Model-Based Software Design using Simulink and Stateflow

Simulink and *Stateflow* are two tools controlled and developed by MathWorks, used to develop algorithm graphically and generate automatically code.

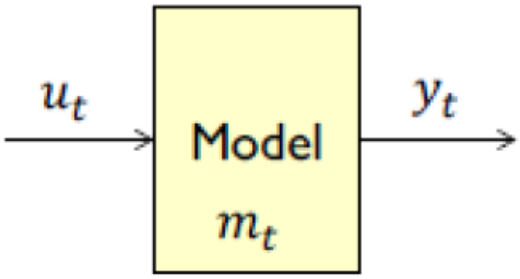


Figure 4.3: Model

Main *differences* between the two tools are:

- Stateflow is a tool to design sequential design logic graphically, using flow charts, graphs and truth tables, extremely used for *Finite-State Machines* (FSM), instead Simulink is a tool that uses blocks and modules;
- Simulink is used to respond to continuous changes in dynamic environment, while Stateflow is used to respond to instantaneous changes in dynamic environment;
- Simulink largely controls oriented solutions, optimally using math operations like sums, integrals, etc., but it lacks when conditional logic is needed (for instance using a IF condition), while Stateflow represents them in a clear and immediate way.

The model [Figure4.3] of a system, both for Simulink and Stateflow, is characterized by a set of element: input u_t, output y_t and the state of the model m_t.
It is defined by[42]:

$$y_t = \lambda(u_t, m_t) \tag{4.1}$$

$$s_{t+\Delta} = \delta(u_t, m_t) \tag{4.2}$$

where t is the current time, λ is an abstraction of the component that it is put on the Simulink model, while Δ is the integration step, that tells which is the next time when it will be repeated the evaluation process.
It is important to underline that the state keeps memory of what happened before the current time.
The integration step Δ depends on the adopted solver[4].
We have two categories of solvers [Figure4.4][42]:

[4]algorithm that Simulink adopts to treat the computation of the next time instant, of λ and Δ

- **Fixed-step solvers**: they have always the same value of Δ, defined once and kept fixed for all the simulation. Decreasing the integration step, it increases the accuracy of the results while increasing the time required to simulate the system. As Δ is selected a priori it can occur errors depending on the signal;
- **Variable-step solvers**: the integration step is chosen automatically by the solver itself. It changes depending on the dynamic of the system. The integration step is reduced to increase accuracy when a model's states are changing rapidly, while when the model's states are changing slowly is increased.

Solvers can be divided also in[42]:

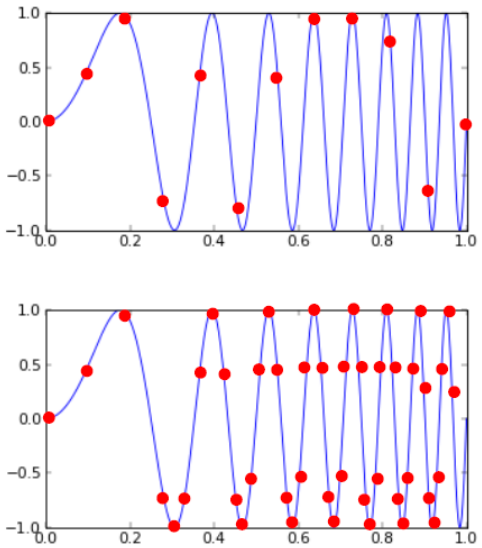


Figure 4.4: Fixed-step solver (*top*) vs variable-step solver (*bottom*)

- **Continuous solvers**: Δ is called "*Major step*" and it is divided in multiple sub-steps called "*Minor step*". Simulink evaluate the system for each minor time step and then it makes an integration to compute the value of the output at $t1$;
- **Discrete solvers**: it does not care about what is happening between one step and the next one.
  The model is evaluated only at major steps.

It is important to underline the difference between *Integration Time* and *Sampling Time*: the first one is the time when Simulink evaluates the behaviour of the model, the second one is the time when the control algorithm samples the plan and

computes the command that shall be actuated in order to reach the selected point [42].
Once it has been chosen the solver, the code generation solver is ready to generate the code.
Code generation files (in case of embedded coder "*ert.tlc*") are:

- **ert_main.c**: is there just for software in the loop purposes. It is possible to test the C code resulting from the Simulink model on the pc;
- **model.c**: contains the source code;
- **model.h**: contains the definition of the data structure;
- **rtwtypes.h**: define how the Simulink data types are translated into the hardware specific data types.

The last three files (*model.c, model.h, rtwtypes.h*) contain the source code of the model and they are the file that have to be included on hardware.
In those files, the model is always translated into three functions:

- **initialize()**: to prepare the execution and reset the model state;
- **step()**: to execute one integration step for the model with fixed-step or discrete time;
- **terminate()**: to clean up memory after the last execution of the model.

4.4 MAAB guidelines

Software design guidelines are not used for making models more efficient, but more readable. The *MathWorks Automotive Advisory Board* (MAAB) guidelines are important for project success.
Guidelines are used to achieve [48]:

- System integration without problems;
- Well-defined interfaces;
- Uniform appearance of models, code, and documentation;
- Reusable models;
- Readable models;
- Problem-free exchange of models;
- A simple, effective process;

- Professional documentation;
- Understandable presentations;
- Fast software changes;
- Cooperation with subcontractors;
- Successful transitions of research or pre-development projects to product development.

Both for *Simulink* and *Stateflow*, the reported design guidelines must be followed [42]:

- Develop layered models;
- Adopt modelling rules for modelling to maximize readability of models;
- Make restricted use of general purpose libraries;

MAAB guidelines focus on naming conventions (for instance, name without underscores), model architecture (from top level to bottom level), enumerated data types and son on[42].
Several MAAB guidelines exist for Simulink and Stateflow usage, as shown in the following list(a more complete lists can be found in [48]):

- **SIMULINK USAGE**
  - In Simulink Model, it must comply with the following rules:
    * Place Inport blocks on the left side of the diagram and move them to prevent signal crossing;
    * Place Outport blocks on the right side of the diagram and move them to prevent signal crossing;
    * "Goto" and "From" blocks can be used if the subsystems connected in a feed-forward or feedback loop have at least one signal line for each direction;
    * All blocks in a model must be sized such that the icon is completely visible and recognizable;
    * The name of the block is placed below the block;
    * Signal lines: should not cross each other (If possible), do not cross any blocks and should not split into more than two sublines at a single breaching point;
    * The signal flow in a model is from left to right;
    * Sequential blocks or subsystems are arranged from left to right;

    - Parallel blocks are arranged from top to bottom;

- **STATEFLOW USAGE**
  - In Stateflow Model, it must comply with the following rules:
    - Transitions do not cross each other, if possible;
    - Transitions do not cross any states, junctions or text fields;
    - Transition labels may be visually associated to the corresponding transition;
    - Transitions are allowed if it correspond to an internal state;
    - Transitions are drawn one upon the other;
    - A new line should start after the entry (*en*), during (*du*), and exit (*ex*) statement;
    - A new line should start after the completition of an assignment statement "*;*".

4.5 Chapter's salient and important points

In this chapter it has been briefly explained the Model-Based Design approach and its advantages in terms of developing time saving, efficiency, testing and validation. Following the V-shape design flow, the embedded code can be implemented on the final application without errors that can be appear using an hand-written approach. MBD uses different graphical tools, like Simulink and Stateflow, that permit to directly visual the algorithm and to generate automatically code.
Since it is a graphic method, some guidelines have to be followed in order to get a more readable model.
MBD approach, coupled with A*algorithm, has been used for the project and the implementation will be discussed in "Chapter 5".

Chapter 5

Algorithm implementation and results

All elements have been described in the previous chapter in order to implement the *path planning* algorithm on the motorized wheelchair using a MBD approach.
The algorithm implementation can be divided in different phases:

1. Mapping adaptation;
2. Model implementation and code generation;
3. On board adaptation.

5.1 Mapping adaptation

Since the motorized wheelchair is considered as a mobile robot that has to move in an indoor environment, such as a hospital, "*Occupancy grid mapping*" method has been applied as the most suitable approach.
Concerning occupancy grid maps, they are suitable for path planning algorithm like A* algorithm and they address the problem of generating consistent maps from noisy data, under the assumption that the robot pose is known.
The basic idea, as already discussed in "Chapter 3", is to represent the map as a field of random variables, disposed in an uniformly spaced grid where each random variable is binary, corresponding to a free space or obstacle location.
The map used to test the algorithm is the reported in [Figure5.1].
First of all, the given map, where the motorized wheelchair should run, has to be clean.
Locations on which the mobile robot can run have the binary value "0", while obstacles, such as walls, have value "1". Obstacles, such as stairs, that are extremely dangerous and can cause huge injuries, must be avoided.
In the cleaning phase, the map represents the space on which the motorized wheelchair

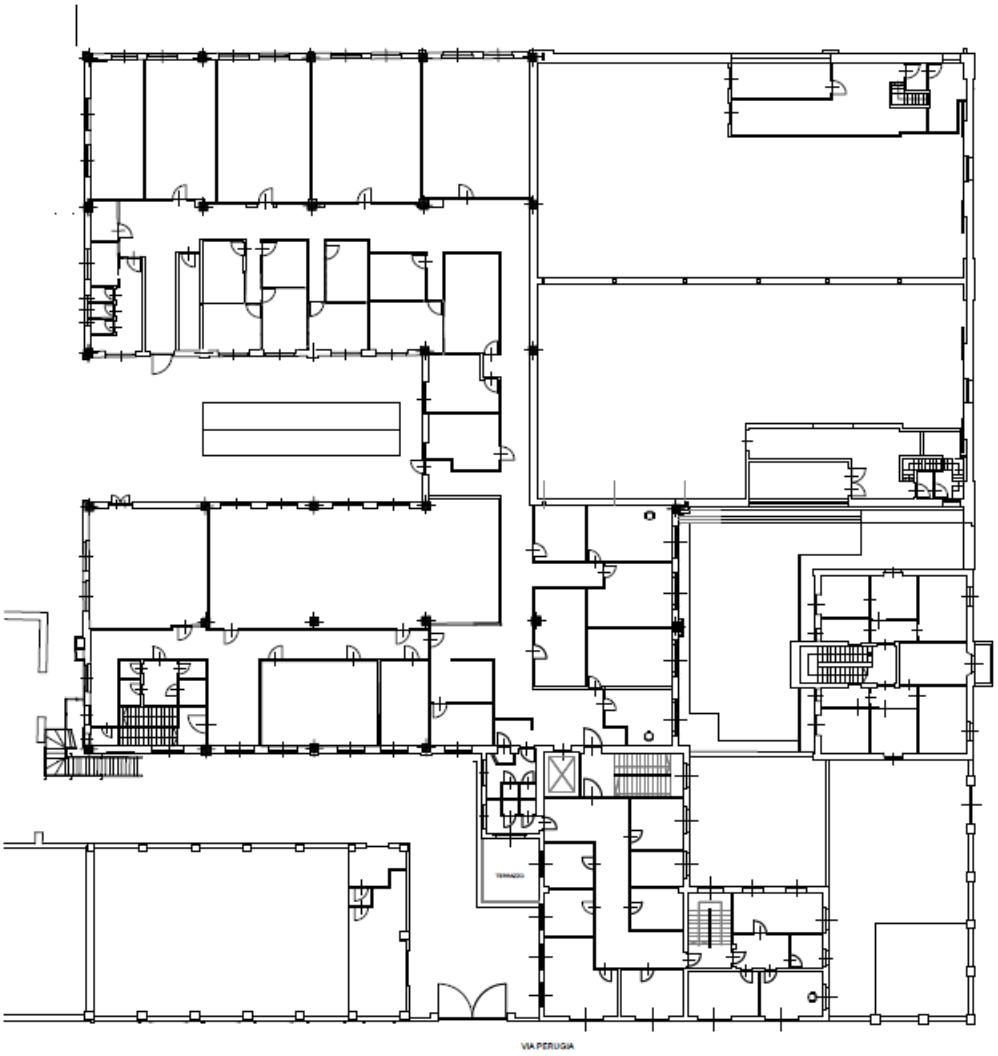


Figure 5.1: Map used to test the algorithm, before the cleaning phase and binary representation

can run, bounded by walls, where the stairs are forbidden and cancelled, to avoid the possibility that the item can reach the stairs.
Mobile obstacles, such as humans or stretchers, are not reported and they are taken into account in the "Obstacle Avoidance" module, out of this master's thesis project.
It is important to underline that since the algorithm should run on board which has a limited memory, each location of the map described by a binary value has an *ad hoc* data type.
In the project, a location in the map has to be considered as a element of a matrix, where each element can have value uint8(0) or uint8(1).
The [Figure5.1] has been described using a matrix 300x237 of "*uint8*" elements.

A* algorithm deals with this mapping adaptation and occupancy grid maps are often used after solving the SLAM problem. Since mapping requires the knowledge of the robot pose, the mobile robot has to localize itself during the acquisition of the map, but it is not the thesis purpose.

5.2 Model implementation and code generation

5.2.1 Model implementation

Once the map has been cleaned, it is possible to model the algorithm on Simulink, through *Simulink blocks*, *Stateflow* or a *Matlab function* [Figure5.2].

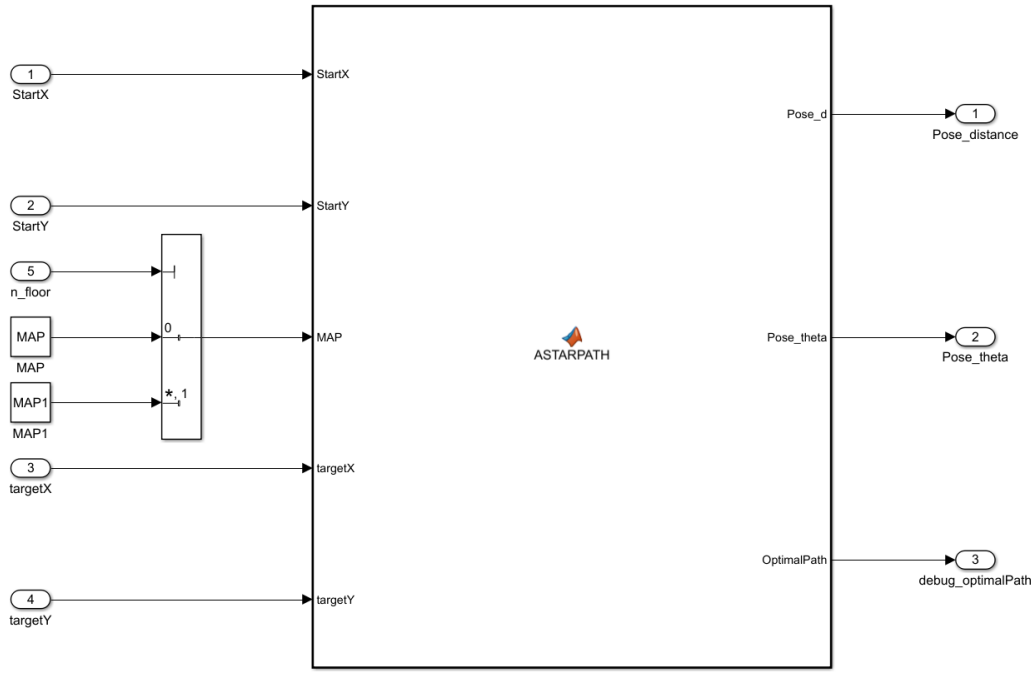


Figure 5.2: Simulink model of the algorithm using a Matlab Function. The usage of *n_floor*, *MAP* and *MAP1* will be discussed in the section "On board adaptation"

Based on A* algorithm, it is possible to adapt and implement an efficient A* search algorithm implementation for pathfinding in occupancy grids. Through the developed code, any heigh an width can be handled in occupancy grid mapping, it is fast and efficient, and it has the possibility to specify multiple goal nodes and connecting distance to other nodes.
The following *code* has been implemented:

```
function [Pose_d,    Pose_theta, OptimalPath] = ASTARPATH(
    StartX, StartY, MAP, targetX, targetY)

OptimalPath = ones([1000  2],'uint16');
Pose_d = zeros([1000  1],'single');
Pose_theta = zeros([1000  1],'single');

for u = uint16(1:size(OptimalPath,1))
    OptimalPath(u,1) = StartY;
    OptimalPath(u,2) = StartX;
end

CurrentY = int16(0);
CurrentX = int16(0);

[Height,Width] = size(MAP);
GScore = zeros(Height,Width,'single');
FScore = single(ones(Height,Width)) * realmax('single');
Hn = zeros(Height,Width,'single');
OpenMAT = int8(zeros(Height,Width));
ClosedMAT = int8(zeros(Height,Width));
ClosedMAT = MAP;
ParentX = int16(zeros(Height,Width));
ParentY = int16(zeros(Height,Width));
GoalRegister = int8(zeros(Height,Width));
GoalRegister(targetY,targetX) = 1;

Neighboors = [1  1 ; 2  1; 3  1; 1  2; 3  2; 1  3; 2  3; 3  3] - 2;
N_Neighboors = 8;

[col, row] = findValue(uint8(1),GoalRegister);
RegisteredGoals = [row col];
Nodesfound = size(RegisteredGoals,1);

for k = 1:size(GoalRegister,1)
    for j = 1:size(GoalRegister,2)
        if MAP(k,j) == 0
            Mat = RegisteredGoals-(repmat([uint16(j),uint16
                (k)],(Nodesfound),1));
            Hn(k,j) = min(sqrt(sum(single(abs(Mat).^2),2,'
                default')));
```

```
39            end
40        end
41    end
42
43    FScore(StartY,StartX) = Hn(StartY,StartX);
44    OpenMAT(StartY,StartX) = 1;
45
46    while 1==1
47        MINopenFSCORE = min(min(FScore));
48        if MINopenFSCORE == realmax('single');
49            RECONSTRUCTPATH=0;
50          break
51        end
52        [CurrentY,CurrentX] = findValue(single(MINopenFSCORE),
              FScore);
53        CurrentY=int16(CurrentY(1));
54        CurrentX=int16(CurrentX(1));
55
56        if GoalRegister(CurrentY,CurrentX)==1
57            RECONSTRUCTPATH=1;
58            break
59        end
60
61        OpenMAT(CurrentY,CurrentX)=0;
62        FScore(CurrentY,CurrentX)=realmax('single');
63        ClosedMAT(CurrentY,CurrentX)=1;
64        for p=1:N_Neighboors
65            i=Neighboors(p,1);
66            j=Neighboors(p,2);
67            if CurrentY+i<1||CurrentY+i>Height||CurrentX+j<1||
                  CurrentX+j>Width
68                continue
69            end
70            Flag=1;
71            if(ClosedMAT(CurrentY+i,CurrentX+j)==0)
72                if (abs(i)>1||abs(j)>1);
73                    JumpCells=2*max(abs(i),abs(j))-1;
74                    for K=1:JumpCells
75                        YPOS=round(K*i/JumpCells);
76                        XPOS=round(K*j/JumpCells);
77
```

```
78                      if (MAP(CurrentY+YPOS, CurrentX+XPOS)
                            ==1)
79                          Flag=0;
80                      end
81                  end
82              end
83
84              if Flag==1;
85                  tentative_gScore = single(GScore(CurrentY,
                        CurrentX) + sqrt(single(i^2)+single(j^2))
                        );
86                  if OpenMAT(CurrentY+i,CurrentX+j)==0
87                      OpenMAT(CurrentY+i,CurrentX+j)=1;
88                  elseif tentative_gScore >= GScore(CurrentY+
                        i,CurrentX+j)
89                      continue
90                  end
91                  ParentX(CurrentY+i,CurrentX+j)=CurrentX;
92                  ParentY(CurrentY+i,CurrentX+j)=CurrentY;
93                  GScore(CurrentY+i,CurrentX+j)=
                        tentative_gScore;
94                  FScore(CurrentY+i,CurrentX+j)=
                        tentative_gScore+Hn(CurrentY+i,CurrentX+j
                        );
95              end
96          end
97      end
98  end
99
100
101
102 k=uint16(2);
103 if RECONSTRUCTPATH
104     OptimalPath(1,1) = uint16(CurrentY);
105     OptimalPath(1,2) = uint16(CurrentX);
106         while RECONSTRUCTPATH
107             CurrentXDummy = int16(ParentX(CurrentY,CurrentX
                    ));
108             CurrentY= int16(ParentY(CurrentY,CurrentX));
109             CurrentX = int16(CurrentXDummy);
110             OptimalPath(k,1) = uint16(CurrentY);
111             OptimalPath(k,2) = uint16(CurrentX);
```

```
112
113             k = k+1;
114             if (((CurrentX== StartX)) &&(CurrentY==StartY))
115                 break
116             end
117         end
118
119         for i=1:size(OptimalPath,1)-1
120             y_curr_point = single(OptimalPath(i,1));
121             x_curr_point = single(OptimalPath(i,2));
122             y_prev_point = single(OptimalPath(i+1,1));
123             x_prev_point = single(OptimalPath(i+1,2));
124             Pose_d(i) = single( sqrt( (y_curr_point -
                    y_prev_point)^2 + (x_curr_point -
                    x_prev_point)^2 ));
125             if x_curr_point ~=x_prev_point
126                 Pose_theta(i) = single(atan(single(-(
                        y_curr_point - y_prev_point)/(
                        x_curr_point - x_prev_point)))*(180/pi));
127             end
128         end
129         Pose_d = flip(Pose_d);
130         Pose_theta = flip(Pose_theta);
131 end
132
133
134
135
136 function [Y,X] = findValue(value, matrix)
137 Y = uint16(0);
138 X = uint16(0);
139 for i = 1:size(matrix,1)
140     for j = 1:size(matrix,2)
141         if(matrix(i,j) == value)
142             Y = uint16(i);
143             X = uint16(j);
144             break;
145         end
146     end
147 end
148 \\
```

The implemented code, reported as a *Matlab function*, is based on the code available on Matlab website for a effiecient A* seach algorithm implementation for occupancy grip mapping [52][53].
Algorithm has simple *inputs*:

- *StartX* and *StartY*: starting point coordinates of the mobile robot (start node);
- *targetX* and *targetY*: target point coordinates of the mobile robot (goal register);
- *MAP*: occupancy grid map cleaned through mapping adaptation;

It is important to underline that, since the code will be imported on a board with limited memory and *double* data type are not supported by Raspberry Pi's boards, saving memory is crucial. Therefore, using a huge map, all variables have their own ad hoc data type. For instance, *StartX*, *StartY*, *targetX* and *targetY* are uint16.
Algorithm's *outputs* are:

- *Pose_d*: distance component of the pose with data type uint16;
- *Pose_theta*: orientation component of the pose with data type uint16;
- *OptimalPath*: matrix used in the algorithm to save the algorithm path, with data type uint16.

Other vectors and matrices that have been used are:

- *CurrentX* and *CurrentY* are two vectors that represents the current node considered in the algorithm. Since it is possible to move both in positive and negative direction, their data types are int16;
- *GScore* is a path-cost function, which is the cost from the starting node to the current node, with uint16 as data type;
- *HScore* is the heuristic estimation of the distance to the goal, with data type uint16;
- *FScore* is the sum of GScore and HScore;
- *OpenMAT* is the open list of the A* algorithm, that represents nodes that have to be visited and explored, with data type int8;
- *ClosedMAT* is the closed list of the A* algorithm, that represents node that have already visited and explored, with data type int8;
- *ParentX* and *ParentY* represent parent's coordinates, that will be used to reconstruct the optimal path. Their data types are int16;

- *GoalRegister* is a register useful to record time by time target points, with data type int8;

- *Neighboors* and *N_Neighboors*, that represent the concept of the "*connecting-distance*" to set up matrices representing neighboors to be investigated;

The concept of "*connecting- distance*" could be explained for navigation system problems.
In the standard A* algorithm search in occupancy grids, only the eight neighbouring tiles are investigated when expanding paths from a node, thus restricting the orientation of the planned path to eight directions, which leads to suboptimal paths[53]. A way to avoid the restriction is to allow each node to connect to nodes that are more than one tile away, as reported in [Figure5.3].

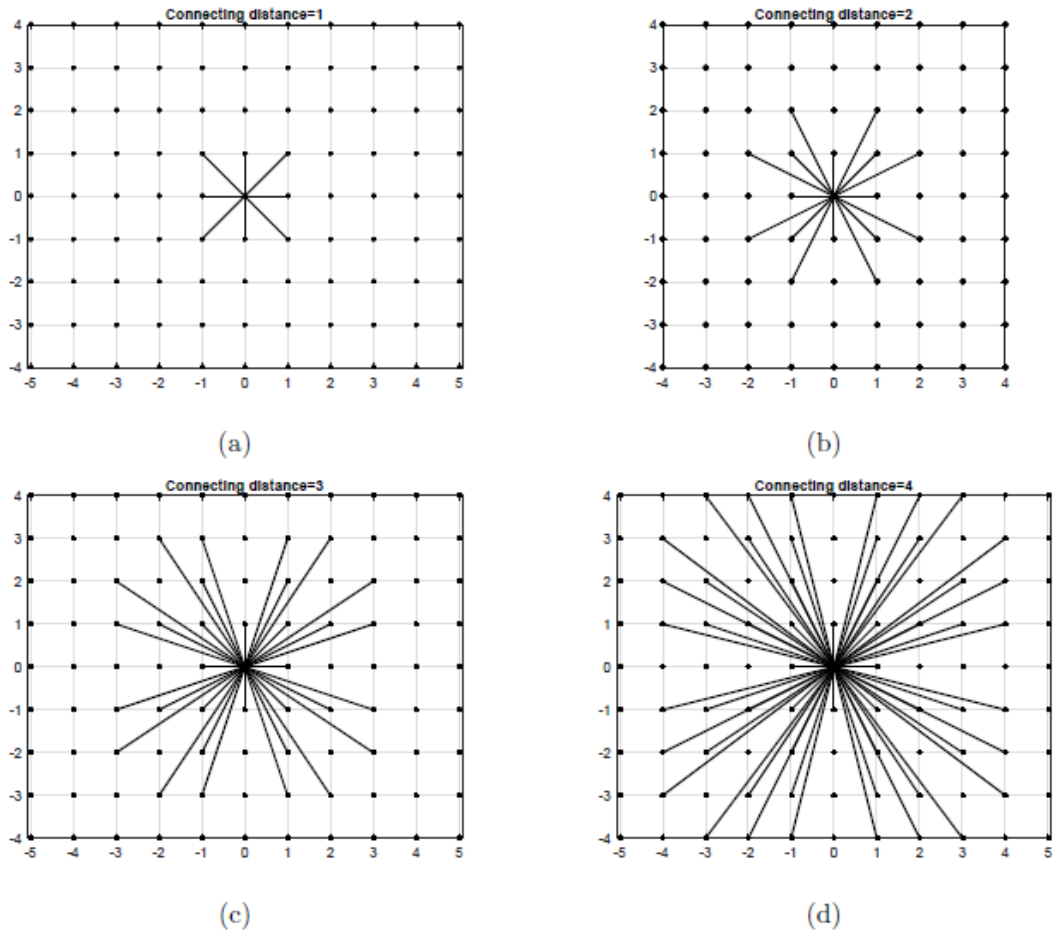


Figure 5.3: Node connections using different connecting-distances [53]

Increasing the connecting-distance quickly, it increases the number of possible directions (for instance, a "*connecting-distance*" of 2 yields 16 possible orientations, while 3 and 4 yields 32 and 54 possible orientations respectively and so on)[53].
It is been proven that the calculated path using a connecting-distance of 4 is both shorter and smoother than when the connecting-distance is 1, but it quickly increases the complexity of calculations and computation times[53].
In the implemented system, the connecting-distance is an integer parameter that has fixed to 1 (therefore, the number of neighboors is 8) to reduce complexity and completition times of the algorithm. In order to get an efficient algorithm, some temporal variables and flags, such as *RegisteredGoals*, *MINopenFSCORE*, *RECONSTRUCTPATH*, *JumpCells*, *Flag*, *tentative_gScore*, *CurrentXDummy*, *y_curr_point*, *x_curr_point*, *y_prev_point* and *x_prev_point* have been used and they will be discussed later on to explain the algorithm implementation.
First of all, it is important to initialize all vectors that will be used to reach the algorithm goal.
Using the size of the map, it is possible to adapt the initialization, sizing the vectors and the matrices:

- *Pose_d* and *Pose_theta* are two vectors of 1000 elements[1] initialized to zero, since they represent the distance and the orientation that the item should perform step by step to reach the goal, *OptimalPath* is initialized as ones and every element of the vector's components is initialized to *StartX* and *StartY* respectively;

- *GScore*, *Fscore*, *Hn*, *OpenMAT*, *ParentX*, *ParentY*, *GoalRegister* are set as zeros matrices. Since *ClosedMAT* represents the already visited nodes, first it is initialized as a zero matrix and then, adding the map to closed matrix, fixed obstacles (like walls) are updated;

Once the initialization is completed, the algorithm can start.
Since A* algorithm needs a heuristic distance, it has been developed an ad hoc function "*findValue(value, matrix)*", substituting the existing function "*find()*" in Matlab able to find the indices of indices of nonzero elements[2], and creating Heuristicmatrix based on distance to nearest goal node.
Once the goal node is updated, the heuristic distance is calculated as a simple distance between two points, thus two elements of the matrix.
At the starting node, the algorithm starts to explore nodes from itself, thus the *Fscore* takes only into account the heuristic distance because the *GScore* function

[1] 1000 is used to take into account a huge number of locations, thus the path planning results more accurate

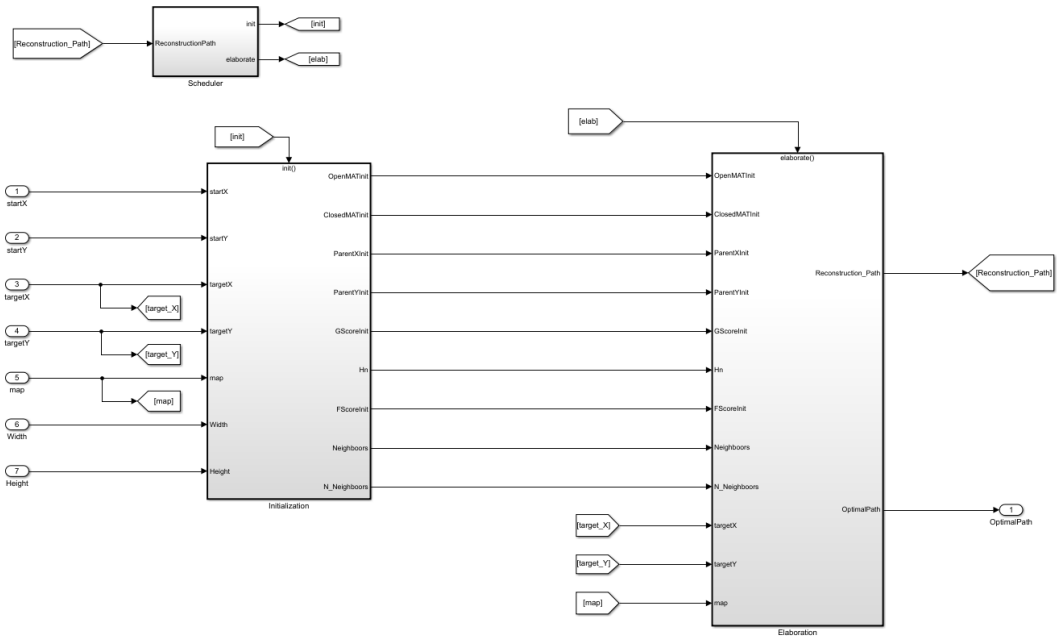
[2] but sometimes it is subjected to errors

represents the cost function from the starting node to the current node, therefore a null contribution to *Fscore*.
After finding *Fscore* value, the node is added to the *OpenMAT* list.
Inside an infinite loop, the code will break when the path is found or when no path exists.
It is important to underline that the algorithm gives distances and orientations from the target node to the start node and then, using a *flip* function, it has been obtain poses from the start node to the target node.
During the algorithm, If the current node is the node included in the goal register, a flag "*RECONSTRUCTPATH*" is set to 1.
Whenever the minimum value of *Fscore* is found, the node is set to 0 in the *OpenMAT*, removing it from the list, and then set to 1 in the *ClosedMAT* list, adding it in that list.
After considering the first node, the algorithm searches looking the node's neighboors: first of all, it check whether the current node considered is inside the available map and the path does not pass an object and then set a *Flag* to 1.
Using "*JumpCells*", it checks that the node, which has to be expanded, can be reached from the current node by checking that each node that is crossed is not a closed cell. The loop over K is supposed to ensure that all nodes, that need to be cleaned, have been checked.
After checking that the path does not pass an object, the algorithm takes into account every *tentative_gScore*, discarding the node with the higher cost function. Once the visited node has the minimum contribution to *FScore*[3], the variables *CurrentX* and *CurrentY* are updated as *ParentX* and *ParentY*.
From now on, the algorithm loops, updating variables such as *CurrentX*, *CurrentY* and *OptimalPath*, until the current node is the starting node.
Since this algorithm measure an absolute pose between a point and the final target point, where the first element is the pose to reach the target and the last one is referred to the pose between the start node and itself, the algorithm has been transformed in order to create a sequence of elements that represent each time the relative pose between the current node and the previous one, using *x_prev_point*, *y_prev_point*, *x_curr_point*, *y_curr_point* and the *flip()* function.
Relative *Pose_d* and *Pose_theta* are trasformed in meters and grades: for this reason, as expected, algorithm code has the following accepted values:

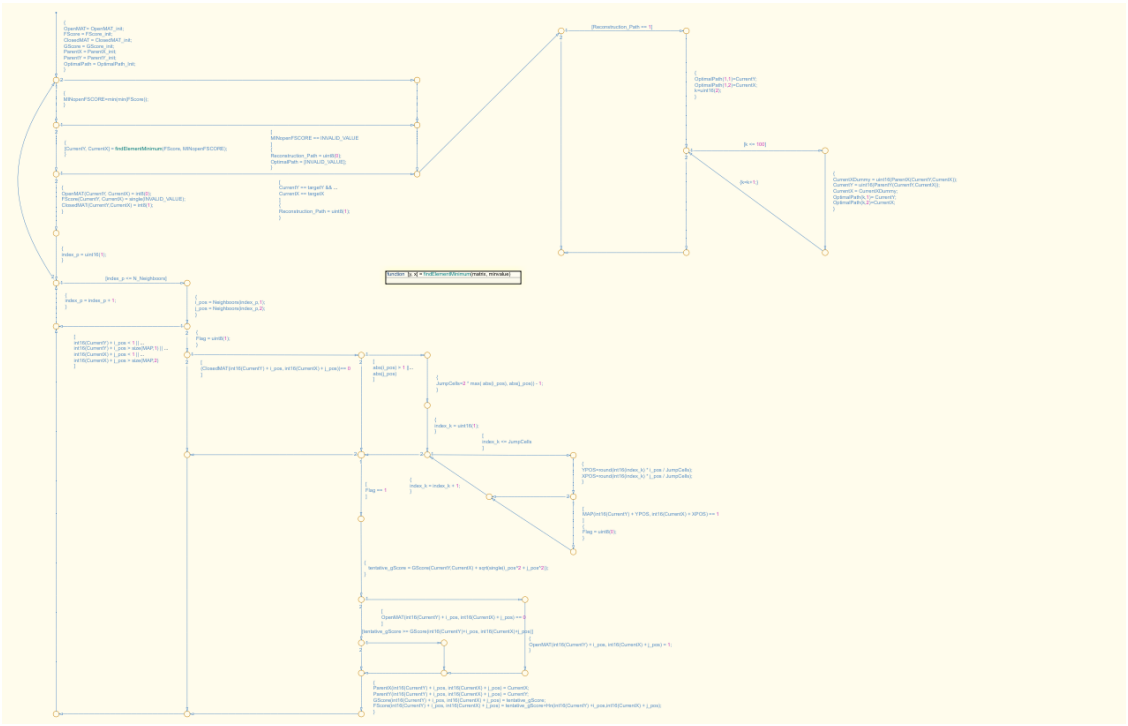
- for *Pose_d*: $[-1.41, -1{,}0, +1, +1.41]m$;
- for *Pose_theta*: $[-90, -45{,}0, +45, +90]^\circ$.

TThe reported Matlab code can be imported in a Matlab Function Block, as shown in [Figure5.2].

[3]simply finding the minimum GScore function



(a) Simulink model using Simulink blocks and Stateflow



(b) Entering in *elab()* subsystem: elaboration function developed using Stateflow

Figure 5.4: Usage of Simulink blocks and Stateflow instead of Matlab function

It is possible to model the same algorithm through Simulink and Stateflow, as reported in [Figure5.4].
It is important to underline that every variable has its own data type, chosen in the "*Model Workspace*[4]".
MBD approach needs to apply MIL, SIL, PIL and HIL tests.
Using model references and a Simulink tool, that permits SIL test, MIL and SIL tests certificate the behaviour of the developed Matlab code and model.
Since PIL needs particular instrumentations[5] and the algorithm does not depend on the hardware, both PIL and HIL tests are not provided.
Finally, the model is ready to be auto-generated.

5.2.2 Code generation

Once every test has been performed, it is important to set parameters in order to auto-generate the code using the code generation tool.
Since the resulting code will be implemented and loaded on board, the following setting parameters in *Configuration parameters* are used:

- Select *Fixed step solver - discrete*;
- Select the setting to use *single-precision* data type when Simulink cannot infer the data type of a signal during propagation;
- In diagnostic, an *algebraic loop* is considered as an *error*;
- In Hardware implementation, insert the *device details* of the board [6];
- In code generations, as system target file selects "*ert.tlc*" for embedded coder, to generate C-code;
- As toolchain, use *Microsoft Windows SDK v7.1*;
- As code generation objectives, add *MISRA C:2012* guidelines, *RAM efficiency* and *ROM efficiency*;
- Generate also *code generation report*, to visualize the results.

Those settings are important for embedded coder, especially it is important to add the code generation objectives. A list of available objectives is reported below:

[4]instead of Base Workspace, used in Matlab

[5]not available in the company

[6]in this case Raspberry Pi Model B+ V1 2

- *Execution efficiency*: configure code generation settings to achieve fast execution time;
- *ROM efficiency*: configure code generation settings to reduce ROM usage;
- *RAM efficiency*: configure code generation settings to reduce RAM usage;
- *Traceability*: configure code generation settings to provide mapping between model elements and code;
- *Safety precaution*: configure code generation settings to debug the code generation build process;
- *MISRA C:2012 guidelines*: configure code generation settings to increase compliance with MISRA C:2012 guidelines;
- *Polyspace*: configure code generation settings to prepare the code for Polyspace analysis.

It is important to select in a predefined order the code generation objectives, since the selection prioritizes the order of the objectives.
First of all MISRA C:2012 guidelines has been selected aiming to facilitate code safety, security, portability and reliability in the context of embedded systems, and then select RAM efficiency before ROM efficiency, since RAM memory is limited in embedded systems.
Before moving to the on board adaptation, it is possible to notice that this code could have problems , such as "*segmentation faults*", due to the usage of the memory. Since "*MAP*" is considered as an input that could change its values, it will be put on RAM memory. Changing the configuration of the code, it has been used the map as *a priori* data, stored in *flash* memory with constant values[7], avoiding the segmentation fault and saving RAM memory.
The resulting files were exported through code generation:

- "**AstarAlgorithm.c**";
- "**AstarAlgorithm.h**";
- "**AstarAlgorithm_private.h**";
- "**AstarAlgorithm_types.h**";
- "**AstarAlgorithm_data.c**";
- "**rtwtypes.h**".

[7] in "AstarAlgorithm_data.c"

5.3 On board implementation

Once code generation files has been generated, the *High level Software* (HLS) of the firmware is done and tested. As reported in [Figure5.5], a firmware has three main components:

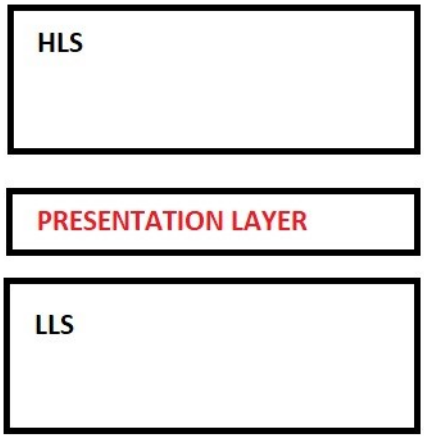


Figure 5.5: Firmware structure

- **HLS or High Level Software**: is the code generated using the Model-Based Design (MBD) approach, as reported in the "Code generation" subsection;
- **Presentation layer**: is a typedef, thus a data structure that interface HLS with LLS, because LLS is faster than HLS;
- **LLS or Low Level Software**: is the basic software that should be hand-written by the designer. Through basic software it is possible to interface with the external environment, using serial communication, CAN and other communication protocols;

Since the algorithm has to communicate with other modules, like the "*Obstacle avoidance*" module of the wheelchair, LLS describes the communication between modules.
In the developed C-code, called "*main.c*", values are sent and received by "*Obstacle avoidance*" (OA) module through serial communication.
It is important to underline that the developed code runs on Raspberry Pi Model B+ V1 2, that permits the serial communication through pins GPIO 14 and GPIO15, respectively for TXD and RXD [Figure5.6].
Other devices were not compatible with the Master's thesis project in the best way

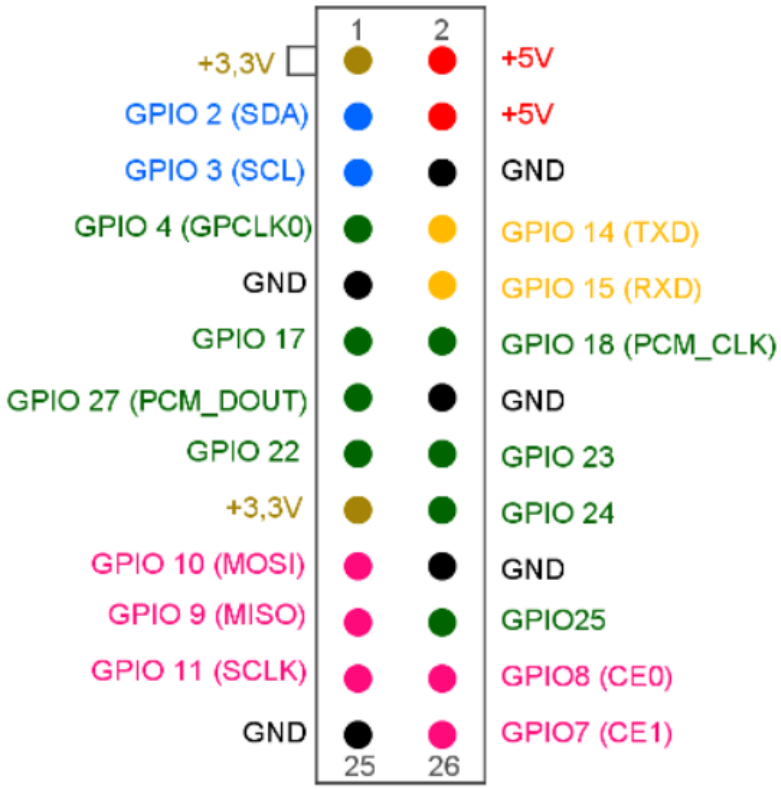


Figure 5.6: Raspberry's pinout

in terms of costs and efficiency: for instance Arduino needs an adaptive circuit between itself and the board on which the "*Obstacle avoidance*" block is implemented (STM32f746g board), or other microcontrollers have less amount of RAM memory. For the "*Obstacle avoidance*" module, the "*Pathfinding*" module has to send poses via serial communication, receiving an acknowledge signal by the OA module.
A sequence of *characters* and values has been chosen to send in the best way data from the pathfinding module to the OA module:

- **D**: start distance;
- **<distance value>** : distance value in *char*;
- **A**: end distance and star angle;
- **P** or **N**: positive or negative angle, depending on the angle value;
- **<angle value>**: angle value in *char*.

Figure 5.7: Raspberry's pinout

It is important to underline that the usage of a file "*Room.h*", where several *typedef struct{}* describe a priori the position that can be reached by the item, leaning on tags useful for OA module and depending on floor and room numbers, is crucial. Since OA module needs accuracy and precision location as target points, each path has different target points: for this reason each target point as a *tag* that indicates If the item is in the center of the room, in front of the door or behind the door. Using that solution, the algorithm between two location is repeated updating, each time the path planning between two points has completed, the target point as a new starting point:

- **5 times**: when the starting point is the "*deposit*" of the wheelchair (supposed to be on floor 0) and the target point is located on the *same floor* in a different room. Target points and path that has to be followed, in order to reach the desired target location and help the OA module, are:

  1. Location point inside the deposit, when exiting the deposit;
  2. Location point outside the deposit, in front of the deposit;
  3. Location point in front of the room (same floor as the deposit);
  4. Location point inside the room, closed to the door (same floor as the the deposit);
  5. Location point inside the room, in the center (same floor as the the deposit);

- **10 times**: when the starting point is the "*deposit*" of the wheelchair (supposed to be on floor 0) and the target point is located on a *different floor* in a different room. Target points and path that has to be followed, in order to reach the desired target location and help the OA module, are:

  1. Location point inside the deposit, when exiting the deposit;
  2. Location point outside the deposit, in front of the deposit;
  3. Location point in front of the elevator (same floor as the deposit);
  4. Location point inside the elevator, closed to the door (same floor as the the deposit);

5. Location point inside the elevator, in the center (same floor as the the deposit);
6. Location point inside the elevator, closed to the door (floor N);
7. Location point in front of the elevator (floor N);
8. Location point in front of the room (floor N);
9. Location point inside the room, closed to the door (floor N);
10. Location point inside the room, in the center (floor N);

It is fundamental, using this solution, that the wheelchair has to be positioned in the deposit at the beginning. In fact, using a *counter* and a *temporal variable*, until the temporal variable is not a predefined value (in the project case set to 1 when main target has reached), the algorithm is repeated and thanks to the counter, that increments itself each time target points are updated, the algorithm could be repeated as many times as request.
Similarly, If the starting point is no longer the deposit (because the item has already reached a new room, starting from the deposit), the algorithm between two location is repeated updating, each time the path planning between two points has completed, the target point as a new starting point:

- **5 times**: when the starting point is the *not* the *deposit room* (supposed to be on floor 0) and the target point is located on the *same floor* in a different room. Target points and path that has to be followed, in order to reach the desired target location and help the OA module, are:

  1. Location point inside the starting room, when exiting the room;
  2. Location point outside the starting room, in front of the the starting room;
  3. Location point in front of the new room (same floor as the starting room);
  4. Location point inside the new room, closed to the door (same floor as the starting room);
  5. Location point inside the new room, in the center (same floor as the starting room);

- **10 times**: when the starting point is *not* the *deposit* of the wheelchair (supposed to be on floor 0) and the target point is located on a *different floor* in a different room. Target points and path that has to be followed, in order to reach the desired target location and help the OA module, are:

  1. Location point inside the starting room, when exiting the room;

2. Location point outside the starting room, in front of the room;
3. Location point in front of the elevator (same floor as the room);
4. Location point inside the elevator, closed to the door (same floor as the the room);
5. Location point inside the elevator, in the center (same floor as the the room);
6. Location point inside the elevator, closed to the door (floor N);
7. Location point in front of the elevator (floor N);
8. Location point in front of the new room (floor N);
9. Location point inside the new room, closed to the door (floor N);
10. Location point inside the new room, in the center (floor N);

Using this solution, thanks to *AstarAlgorithm_step()* function, the algorithm runs and poses could be sent to the OA module.
Since a serial communication is needed, it has been included the "*WiringPi*" library, a PIN based GPIO access library written in C used in all Raspberry Pi versions.
Including "*wiringSerial.h*" and "*wiringPi.h*", serial communication functions can be used.
It is important to remember that since serial communication needs 8 bit communication, the pose calculated by the auto-generated code has to be compacted and transformed:

- Distance is transformed in *char*, divided by N[8], transformating pixel in meters based on the available map and multiplicated by 10, to get integer numbers;
- Orientation is transformed in *char* and it does not need any other transformation.

Other characters are sent via serial communication in order to synchronize OA module with pathfinding module, as it can be seen in the next section.
In the following section, simulation examples of the developing code are shown.

[8]decided a priori, depending on the map

5.4 Simulation results

Figure 5.8: Simulation starts when "R" character is received, the starting point is the "Deposit", located in position [39;24], and when the starting point is received it sends "F" as character. Choosing as a target point "Floor 0 - Room 3", it starts to send, according to the serial protocol, poses. It can be seen that the target point is not the final target point, but it is a target point needed for the OA module. It has been represented the first 1000 locations of 5000 (5 repetitions of the algorithm, updating every time the target point as the new starting point). Every time the "partial" target has reached, "C" character is sent.

Figure 5.9: Simulation continues to reach target point "Floor 0 - Room 3". Last 1000 poses of 5000 are shown and it can be seen that the final target has been reached. That target point will become the new starting point when inserting a new target.

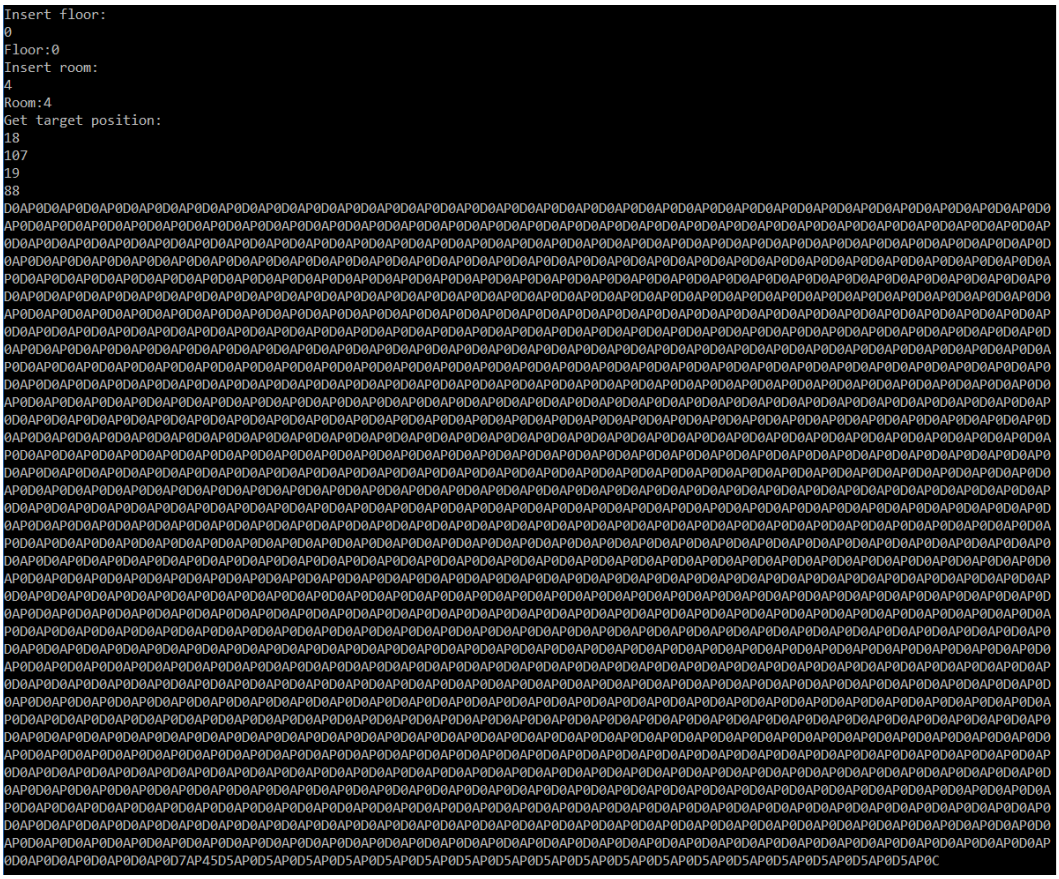


Figure 5.10: Now the starting point is "Floor 0 - Room 3". Inserting "Floor 0 - Room 4" as new final target point, the algorithm re-calculates poses and sends them serially, like the previous case but now the starting point is no longer the deposit. First 1000 poses are shown in the figure.

Figure 5.11: Simulation continues to reach target point "Floor 0 - Room 4". Last 1000 poses of 5000 are shown and it can be seen that the final target has been reached. That target point will become the new starting point when inserting a new target.

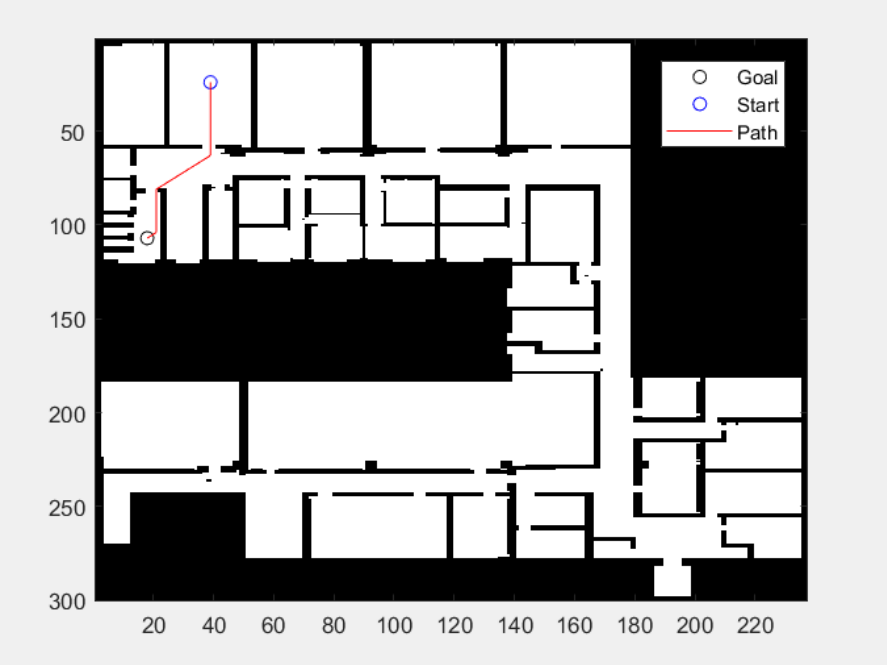


Figure 5.12: Matlab simulation: starting from the deposit to reach "Floor 0 - Room 3"

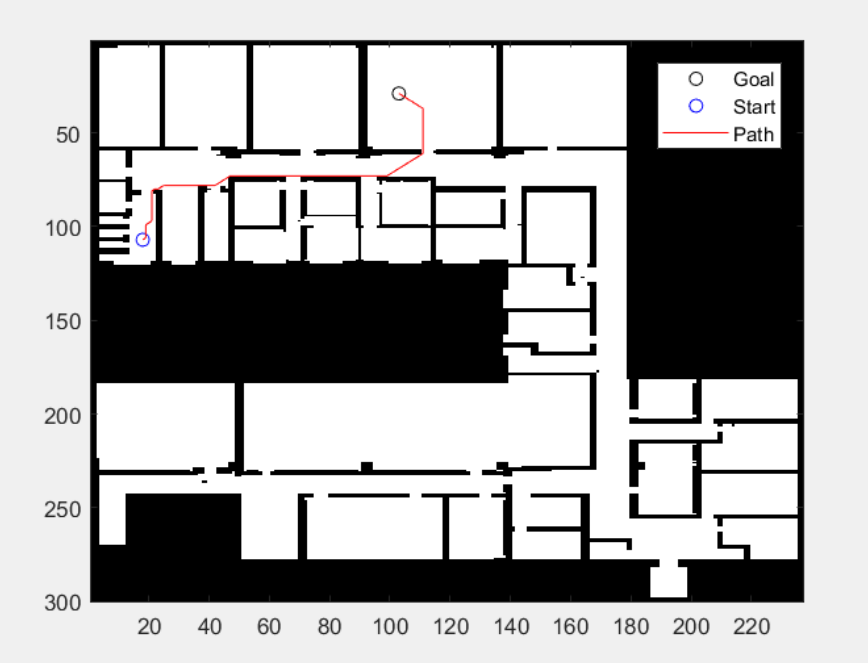


Figure 5.13: Matlab simulation: starting point is "Floor 0 - Room 3" to reach "Floor 0 - Room 4".

5.5 Chapter's salient and important points

In this chapter it has been briefly explained how it has been implemented the project.
Starting from the mapping adaptation, the algorithm has been modelled according to requested specification. Once code has been tested and simulated, the HLS part of the firmware is completed.
LLS part of the firmware has been developed, since it has to interface with other modules of the motorized wheelchair.
"*Pathfinding*" module interfaces with "*Obstacle Avoidance*" module via serial communication, using an ad hoc protocol.
Examples of simulation results, through terminal and Matlab, have been reported.

Chapter 6

Comments and future work

Pathfinding is the first step in order to create an autonomous mobile robot.
It is important to underline that this solution and implementation is *independent* from the final application point of view: implementation code can run on any mobile robot solution, not only on a motorized wheelchair.
Key point of the implementation is the usage of *Model-Based Design* (MBD) approach, an innovative graphical method able to save up to 50% or more of development time, including testing time.
Thanks to *A*(A-star) algorithm* adaptation, a low cost pathfinding solution in terms of computational time and effort is implemented on board. This solution is considered as a "*local*" solution, nevertheless it can be adapted on different motorized wheelchairs controlled by servers, but costs could be higher.
Different boards can be used instead of Raspberry Pi B+ V1 2, but algorithm should run locally on the motorized wheelchair and whole cost should be low, since it is a different low cost solution with respect to existing commercial motorized wheelchairs that could help people like in hospital.
It can be seen that the implemented pathfinding algorithm can run on different floors, depending on map sizes, and each map should be cleaned to avoid injuries obstacles such as stairs.
Other features that motorized wheelchair will have to avoid injuries can include module such as tip-over prevention.
Whole system project needs *trajectory planning* module, where pathfinding implementation represents the main part, and trajectory planning can be obtained simply including kinematic and dynamic constraints on pathfinding implementation.
Once trajectory planning module is ready, localization module is necessary to create a SLAM module and interface with the OA module.
Other modules can be added to the project, such as the *IoT* (Internet of Things) call of the elevator.
Project pathfinding solution includes location with *tags*, not only able to simplify OA module work, but it can be used by a IoT module that can recognize when the

item is in front of an elevator to call it.
Using tags such as:

- "**I**": inside the elevator, closed to sliding doors;
- "**O**": outside the elevator, closed to sliding doors;
- "**M**": inside the elevator, in the center.

characters can be sent serially to IoT module.
It is important to remember that the used serial communication needs *char* as data type, since it is fast and simple, and added devices send and receive *char*s, like between OA module and Pathfinding module.
An optimization can be done increasing the *connecting-distance* value, since the calculated path using a higher connecting-distance is shorter and smoother, but it quickly increases the complexity of calculations and computation times, thus the algorithm needs more power from the board and the usage of another board is suggested.
Another future optimization can be the reduction of *localization errors* thanks to light bulbs, positioned at a constant distance in the ceiling.
Taking into account light beams, localization errors that can be derived from wheel's slippage can be reduced.
Another module that will be added in the future is "*Voice recognition*" module: from now on, the item has its own planned path whenever the user (in particular the nurse who helps the patient) types and confirms the room and floor number, but in the future it will not be necessary anymore through voice recognition and command.
Thanks to MBD approach, modules can be modelled and added in the project, interfacing themselves, saving developing time and testing themselves.
A key point of the project is that MBD approach can be used not only in the automotive field, but it can be widely used in any technological field, like Robotics and Augmented Reality, expanding the range of opportunities to improve next-generation industries.
In conclusion, MBD is the *future of designing*, increasing technical communication efficiencies, improving product quality and unleashing the power of emerging technologies.

Bibliography

[1] B.S. Cho, W.S. Moon, W.J. Seo and K.R. Baek: *A dead reckoning localization system for mobile robots using inertial sensors and wheel revolution encoding*, Department of Electronic Engineering, Pusan National University, Kumjeong-Gu, Busan, Korea (Manuscript Received May 9, 2011; Revised June 27, 2011; Accepted July 13, 2011).

[2] E. Marcuzzi: *Localization methods and sensor fusion for mobile robots*, Universitá degli studi di Pavia.

[3] P. Salvagnini, F. Simmini, M. Stoppa: *Navigazione autonoma di robot in ambiente sconosciuto*. Universitá degli studi di Padova, Corso di Progettazione dei sistemi di Controllo a.a. 2008/09. Padova, 16 Febbraio 2009.

[4] M. Dalli: *Sviluppo di un sistema di controllo basato su odometria per una carrozzina robotica*. Politecnico di Milano. 2007-2008

[5] No author: Appunti di lezione "*Corso di Robotica Industriale* ". Universitá degli Studi di Catania.

[6] J.C. Latombe: *Robot Motion Planning*. Standford University. Originally published by Kluwer Academic Publishers, New York in 1991. Second printing, 1991.

[7] S. Thrun, W. Burgard, D. Fox: *Probabilistic robotics*. Standford University, University of Freiburg, University of Washington. 1999-2000.

[8] V. Ganapathy, S. Yun, T. Chien: *Enhanced D* Lite Algorithm for Autonomous Mobile Robot*. University of Malaya, School of Engineering - Monash University Sunway Campus, School of Engineering - Monash University. March 2011.

[9] S. Rönnbäck: *On Methods for Assistive Mobile Robots*. Luleä University of Technology. 2006.

[10] B. Siciliano, L. Sciavicco, L. Villani, G. Oriolo: *Robotics: Modelling, Planning and Control*. 2009 Springer-Verlag London Limited.

[11] R. Siegwart, I. R. Nourbakhsh: *Introduction to Autonomous Mobile Robots*. A Bradford Book. The MIT Press Cambridge, Massachusetts. London, England. 2004 Massachusetts Institute of Technology.

[12] R. C. Simpson: *Smart wheelchairs: A literature review*. Journal of Rehabilitation Research and Development (JRRD). July/August 2005.

[13] R. Madarasz, L. Heiny, R. Cromp, N. Mazur: *The Design of an Autonomous*

Vehicle for the Disabled. IEEE Journal of Robotics and Automation, 1986.

[14] G. Del Castilloa, S. Skaara, A. Cardenasb, L. Fehr: *A sonar approach to obstacle detection for a vision-based autonomouswheelchair*. Robotics and Autonomous Systems, 2006.

[15] T. Felzer, B. Freisleben:*Hawcos: The "hands-free wheelchair" control system*. Proceedings of the 5th International ACM SIGCAPH Conference on Assistive Technologies (ASSETS 02), 2002.

[16] Y. Kuno, N. Shimada, and Y. Shirai: *Look where you are going (robotic wheelchair)*. IEEE Robotics & Automation Magazine, 2003.

[17] F. Bley, M. Rous, U. Canzler, and K.F. Kraiss: *Supervised navigation and manipulation for impaired wheelchair users*. IEEE International Conference on Systems, Man and Cybernetics, 2004.

[18] R. Simpson, E. LoPresti, S. Hayashi, I. Nourbakhsh, D. Miller: *The smart wheelchair component system*. Journal of Rehabilitation Research & Development, May/June 2004.

[19] S. Mascaro, J. Spano, H. Asada: *A reconfigurable holonomic omnidirectional mobile bed with unified seating (rhombus) for bedridden patients*. In Proceedings of the IEEE International Conference on Robotics and Automation, volume 2, 1997.

[20] R. Morales, V. Feliu, A. Gonzalez, P. Pintado: *Kinematic model of a new staircase climbing wheelchair and its experimental validation*. International Journal of Robotic Research, 2006.

[21] A. Pruski, M. Ennaji, and Y. Morère: *Vahm : A user adapted intelligent wheelchair*. IEEE Conference on Control Application, Glasgow, Scotland, Sep 2002.

[22] H. A. Yanco, J. Gips: *Preliminary investigation of a semi-autonomous robotic wheelchair directed through electrodes*. In Proceedings of the Rehabilitation Engineering and Assistive Technology Society of North America Annual Conference (RESNA 1997), 1997.

[23] P.D. Nisbet, J. Craig, J.P. Odor, S. Aitken: *"smart" wheelchairs for mobility training. Technology and Disability*. 1995.

[24] B. Bona: *Mobile & Service Robotics. Sensors for Robotics - 4*. Slides: ROBOTICS 01PEQW. 2015/2016.

[25] B. Bona: *Mobile Robots Navigation. Overview, examples and ongoing research*. Slides: ROBOTICS 01PEQW. 2015/2016.

[26] B. Bona: *Mobile & Service Robotics. Sensors for Robotics - 2*. Slides: ROBOTICS 01PEQW. 2015/2016.

[27] B. Bona: *Mobile & Service Robotics. Kinematics*. Slides: ROBOTICS 01PEQW. 2015/2016.

[28] B. Bona: *Mobile & Service Robotics. introduction*. Slides: ROBOTICS 01PEQW. 2015/2016.

[29] B. Siciliano, O. Khatib: *Springer Handbook of Robotics*. Springer-Verlag Berlin Heidelberg, 2016.

[30] A. Rizzo: *Robotics*. Slides AUTOMATION AND PLANNING OF PRODUCTION SYSTEM 01PEDQW , 2018.

[31] S.P. Levine, D.A. Bell, L.A. Jaros, R.C. Simpson, Y. Koren, J. Borenstein: *The NavChair Assistive Wheelchair Navigation System*. IEEE Transactions on Rehabilitation Engineering, 1999.

[32] M. Nosrati, R. Karimi, H.A. Hasanvand: *Investigation of the *(Star) Search Algorithms: Characteristics, Methods and Approaches*. World Applied Programming, Vol (2), No (4), April 2012.

[33] H. Reddy: *PATHFINDING - Dijkstra's and A* Algorithm's*. December 13, 2013.

[34] P.K. Das, S.N. Patro, C.N. Panda, B. Balabantaray: *D* lite algorithm based path planning of mobile robot in static Environment*. International Journal of Computer & Communication Technology (IJCCT), Volume-2, Issue-VI, 2011.

[35] No author: *A* search algorithm*.
Web link: https://en.wikipedia.org/wiki/A*_search_algorithm

[36] No author: *D**. Web link: https://en.wikipedia.org/wiki/D*

[37] No author: *Introduction to A**.
Web link: http://theory.stanford.edu/∼amitp/GameProgramming/AStarComparison.html

[38] S.Heber: *A**. 2012.
Web link: https://github.com/BigZaphod/AStar

[39] No author: *A* search algorithm*.
Web link: https://rosettacode.org/wiki/A*_search_algorithm

[40] No author: *Path Finding Algorithms*.
Web link: https://medium.com/omarelgabrys-blog/path-finding-algorithms-f65a8902eb40

[41] M. Violante: *Introduction to model-based software design*. Politecnico di Torino, Dip. Automatica e Informatica. Slide notes, 2016.

[42] M. Violante: *Model-based software design with Simulink*. Politecnico di Torino, Dip. Automatica e Informatica. Slide notes, 2016.

[43] M. Violante: *Introduction to hardware in the loop testing*. Politecnico di Torino, Dip. Automatica e Informatica. Slide notes, 2016.

[44] M. Violante: *Software Testing*. Politecnico di Torino, Dip. Automatica e Informatica. Slide notes, 2016.

[45] M. Violante: *Introduction to code optimization*. Politecnico di Torino, Dip. Automatica e Informatica. Slide notes, 2016.

[46] T. Stahl, M. Völter, J. Bettin, A. Haase, S. Helsen:*Model-Driven Software Development*. British Library, 2006.

[47] N.C.W.M. Braspenning, J.M. van de Mortel-Fronczak, J.E. Rooda:*A Model-based Integration and Testing Method to Reduce System Development Effort*.

Department of Mechanical Engineering, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands, 2006.

[48] No author: *MathWorks Automotive Advisory Board: Control Algorithm Modeling Guidelines Using MATLAB, Simulink and Stateflow*. 2007-2015 by MathWorks Automotive Advisory Board.

[49] G. Gaviani, G. Gentile, G. Stara, L. Romagnoli, T. Thomsen, A.Ferrari: *From conception to implementation: a model based design approach*. IFAC Advances in Automotive Control, Salerno, Italy, 2004.

[50] No author: *Why adopt Model-Based Design for Embedded Control Software Development?*. The MathWorks, 2014.

[51] K. Chauhan: *Why is Model-Based Design important in embedded systems?*. 2018.
Web link: https//:www.einfochips.com/blog/why-is-model-based-design-important-in-embedded-systems

[52] E.S. Ueland: *A* (Astar / A Star) search algorithm. Easy to use*. 2017.
Web link: https://it.mathworks.com/matlabcentral/fileexchange/56877-a-astar-a-star-search-algorithm-easy-to-use?s_tid=mwa_osa_a

[53] E.S. Ueland, R. Skjetne, A.R. Dahl, H.M. Heyn, P. Norgren: *Marine Autonomous Exploration Using a Lidar*. Norwegian University of Science and Technology, Department of Marine Technology, June 2016.