POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering
A.y. 2021/2022

Master's Degree Thesis

New Path Planner Methods for Planetary Explorations

Supervisors:

Prof. Marcello Chiaberge
Ing. Andrea Merlo

Candidate:

Gabriel Palcau

October 2022

Abstract

In this thesis a new path planner method based on NURBS (Non Uniform Rational Basis-Splines) geometric curves is presented and applied in the context of planetary surface exporation. A review on robot navigation and path planner algorithms is done with particular focus on exploration rovers tecniques, used by *NASA* and *ESA*. After that, NURBS technicalities and integration in a new path planner algorithm are explained in detail. Last, a comparison between classical path planner algorithms, like A*, and NURBS path planner is discussed, focusing on the advatages and results of the latter.

The methodologies used are several, form NURBS mathematics, which is a kind of parametric curves with particular properties, to computer vision and C++ algorithms. The designed path planner is composed by three differents parts. The first is the obstacle detection algorithm, that takes a top-view surface image (such as moon or mars surface) of the surrounded robot environment, and detects the obstacles based of computer vision and perception algorithms. The second is the path planner algorithm based on NURBS matematics and SISL (Sintef Spline Library) library; the last is a grafical viewer based on OpenGL (Open Graphics Library) that renders obstacles, rover and paths. This last one is also used for a dynamic simulation of the rover along the path with a continously re-planning of the trajectory in the waypoints.

Finally, results, simulation and tests are provided and explained. Comparison with A* is carried out and advatages/disadvantages of NURBS path planner are discussed. Advangates can be found in the peculiar properties of NURBS geometric curves. It is possible to define curves (i.e. rover trajectories) having lower complexity and grater accuracy, which is indipendent from the number of obstacles and few points, called control points, can express and completely characterized the NURBS. These properties will lead to benefits in term performace, usage memory and smoother trajectories. This will hand up on having easier curve to manipuate since very low degree can be imposed independently, less memory to save the generated trajectory respect to mesh approximation, and smoother trajectories are achieved, that are optimal in the planetary exploration context.

Acknowledgements

First of all, I would like to thank Thales Alenia Space Italia for giving me the opportunity of doing the thesis with them, in such a interesting context. In particular, I would like to thank Andrea Merlo, my thesis advisor in Thales, and specially Marco Lapolla who followed me throughout the thesis.
Moreover, I would like to thank my thesis coordinator *Prof.* Marcello Chiaberge of DET at Politecnico di Torino and the Pic4Ser members of Politecnico di Torino.

Finally, I express my gratitude to my parents and brother for providing me with support and continuous encouragement throughout my years of study.
Thank you to all.

Table of Contents

List of Figures

Chapter 1

Introduction

In this chapter we will contextualize and introduce the problem of autonomous navigation related to rover planetary explorations. An overview on robot and rover applications on space is done.

1.1 Space Robotics

Space robotics is a fascinating field born in the context of space exploration. The use of this technology is essential to conduct space missions, where human has its limit that are related to human physical endurance. Long duration of flights and missions together with hostile environment in space limits human involvement [1]. Autonomous missions became necessary to successfully complete an exploration mission.

Space robotics had a great increase from the last decades of the 20-th century, and their continuous evolution has allowed humans to explore the solar system with probes or rovers. An emerging application has been the use of robotic arms for the International Space Station (ISS) or for Moon and Mars surface exploration.

There are three types of robots used in space: manipulators, mobile, and flying robots. Manipulators are robotic arms used in space operation, assembly, and servicing. The second group are the mobile robots such as rovers and autonomous group vehicles (AGV). The third category are flying robots or drones known as unmanned aerial vehicles (UAV) [2].

The rovers are designed to explore hostile environments in “search of life” or for study purpose. The primary task of rover in to navigate autonomously generating a path and localize itself using cameras. The other tasks that the rover has to

do are those for which it was designed, from those that have to search for life on an extra-terrestrial body, to others that have to acquire data's to have a greater knowledge of the birth of the solar system and the cosmos.
Depending of the tasks for which the rover was designed, different instruments are present to achieve the misison, such as drill or rope analysing soil samples on site and sending data's to the Earth.

Rovers which land on extra-terrestrial bodies, such as Mars, cannot be remotely controlled in real-time due to the large distances from Earth and consecutively to the big delays in real-time communication. For example, sending a signal from Mars to Earth takes up to 3 to 20 minutes. These rovers must be capable of operating autonomously using cameras and sensors.

1.2 Planetary Explorations

During years, several rover exploration programs have been launched to explore firstly the Moon and Mars planet next, and technologies have improved at every mission, in particular the level of autonomy of a rover, ranging from remote-controlled to autonomous navigation.

Robotic planetary explorations missions started in the last quarter of 20-th century with Lunokhod 1 deployed on November 1970 on the Moon as part of Luna 17 Soviet mission. Lunokhod 1 and Lunokhod 2, launched on January 1973, were the first remote-controlled rovers to freely move on an extra-terrestrial body. These rovers could take pictures and measure the physical and chemical properties of the lunar soil.

A more advanced rover was deployed on Mars in July 1997, with the Mars Pathfinder mission by NASA. The mission was composed of the lander, Pathfinder, and the rover called Sojourner. It was the first rover that was able to move in autonomy from its position to the arrival point. The rover communicated with Earth once per *sol* (a sol corresponds to a solar day on Mars $\sim$ 1.02 Earth day) receiving the coordinates of the arrival point with the command "Go to Waypoint". An appropriate strategy for overcoming obstacles autonomously was implemented. The path generation algorithm implemented on the on-board computer attempted to reach the arrival point in straight line, if an obstacle is detected with the laser emitter different strategies were present. Basically to avoid obstacles the rover turn left or right until the obstacle is no longer in sight and proceed in that direction from 30 centimetres and recalculates the path with a straight line [3]. The main tasks of the rover is analysing Mars rocks composition and taking images while moving.

In 2003 the NASA's Mars Exploration Rover (MER) mission started. It was a robotic mission involving two Mars rovers, Spirit and Opportunity. The movements of the rovers were planned based on the returned images; simulation and tests were performed before in a simulated Mars environment on Earth, with the aim of letting the rover conduct the commands autonomously in the real Mars environment. Opportunity has the primate of the longest distance travelled by any rover (45.16 km) and for the most days operated (14 years and 140 days).

A more advanced navigation technique was experimented with Curiosity rover in the mission of Mars Science Laboratory (MSL). The rover landed on Mars surface on 6 August 2012, and was design to explore Gale crater, since offers environment condition favorable for microbial life, and planetary habitability studies in preparation for human exploration.
The navigation system uses two pairs of black and white navigation cameras (navcams) used to acquire stereoscopic 3D images. The autonomous navigation, or autonav, analyses the images taken during a drive to calculate a safe driving path. Curiosity takes several sets of stereo pairs of images, and the rover's computer processes that information to map any geometric hazard or rough terrain [4]. The rover considers all the paths it could take to get to the designated endpoint and chooses the best one. Curiosity was the most autonomous and advanced rover ever designed landed in an extra-terrestrial body, it proved how power and independent rover can be and their enormous potentiality in the exploration of planets.

In 2021, Perseverance rover landed on MARS as part of NASA's Mars 2020 mission. This rover, figure 1.1, is the most advanced one currently operating on a extra terrestrial body, in this case Mars. Perseverance has the dimension of a car (2.9m x 2.7m x 2.2m) and is very similar to its predecessors Curiosity, with more advanced technologies. NASA's Perseverance rover drives autonomously on Mars using an enhanced auto-navigation system, AutoNav. According to NASA, the technology lets Perseverance take control of its wheels and drive by itself across the red planet, without rely on commands from Earth [5].
AutoNav system is equipped with an upgraded technology respect Curiosity rover. Perseverance can make 3D maps of the terrain ahead, identify obstacles, and establish a path in completely autonomy. This means a complete and more independent rover able to drive more and at much faster speeds. The AutoNav limit the human intervention to few commands, such as the final destination, it just increases the rover's autonomy. As the rover drives this path, it uses the map to identify which paths are safe to drive. Perseverance's self-driving software is a significant improvement over previous rovers which also had self-driving capabilities.

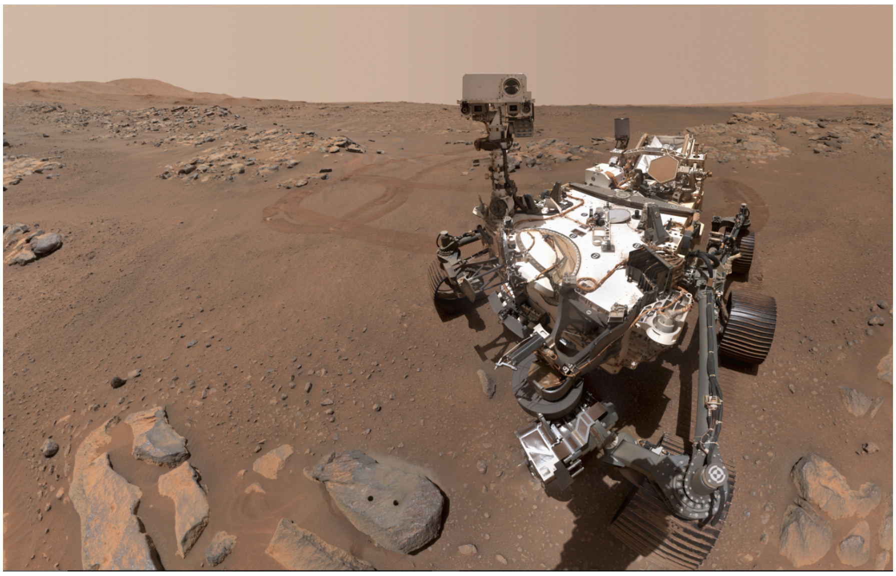


Figure 1.1: Perseverance rover on Mars

Perseverance is able to process and analyse images while the wheels are still in motion.
A key objective for Perseverance's mission on Mars is astrobiology, including the search for signs of ancient microbial life. The rover will investigate the planet's geology and past climate, and will be the first mission to collect and cache Martian rock [5].

The European Spacae Agency (ESA) together with the Russian Space Agency (Roscosmos) will also participate in planetary exploration missions with rovers in the coming years, with the ExoMars mission. The mission is an astrobiology and exobiology program, which allows the rover, Rosalind Franklin figure 1.2, to search for signs of past and present life on Mars, to investigate the Martian water and geochemical environment, to investigate the atmospheric gases, and to demonstrate the technologies for a future Mars sample-return mission [6]. To achieve this, the rover contains a highly autonomous navigation system capable of planning a safe and drivable path across the terrain. The rover will be controlled and monitored by the Rover Operations Control Centre (ROCC), located in Turin, Italy.
Two stereo camera pairs (NavCam and LocCam) allow the rover to acquire images to generate a 3D model of the next six meters of observable terrain, figure 1.3. The 3D map is analysed to understand which area are the most traversable for the rover. A "navigation map" is generated and the navigation systems plans a safe path of 2 meters long in the direction of the final destination specified by the ROCC. To enable more rapid progress, the ExoMars Rover's Guidance, Navigation

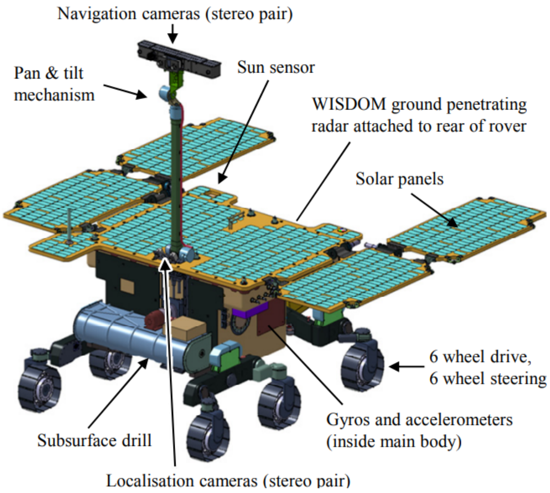

Navigation cameras (stereo pair)
Pan & tilt mechanism
Sun sensor
WISDOM ground penetrating radar attached to rear of rover
Solar panels
6 wheel drive, 6 wheel steering
Gyros and accelerometers (inside main body)
Subsurface drill
Localisation cameras (stereo pair)

Figure 1.2: Rosalind Franklin illustration

and Control (GNC) system uses vision-based methods to autonomously traverse Martian terrain without the aid of ground controllers [7].

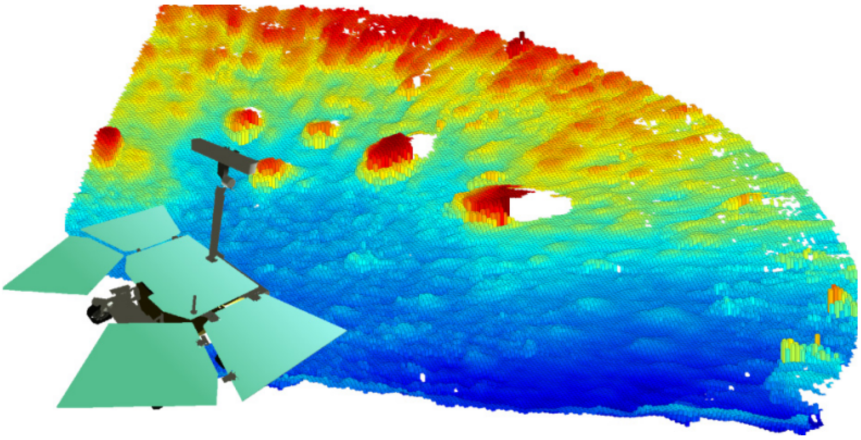


Figure 1.3: 3D map generated by Rosalind Franklin in test environment

1.3 Thesis Outline

We have an introduction of the main rovers used by different Space Agency to explore extra-terrestrial body, and what are the techniques used in autonomous navigation for planetary explorations.

This master thesis is organized as follows: in chapter 2 the navigation problem of robots and rovers is presented, analysing how from an image the robot localize itself and all the obstacles. The mainly methodologies on navigation eighter is space or not is presented; chapter 3 the State of the Art on path planning is discussed. Chapter 4 will present all the mathematics needed to understand and apply NURBS curves for path planning; chapter 5 will deeply analyse the implementation of a new path planner algorithm based of NURBS mathematics. Finally chapte 6 will show results, simulations and comparison between NURBS algorithm and A*, while chapter 7 will draw conclusions.

Chapter 2

Navigation

2.1 Introduction of mobile robots navigation

Navigation is the most important task for mobile robots and its non-realization will lead the robot to not carry out the activities for which it was designed and conceived. A relevant example, one of the most fascinated, may be the planetary exploration of a non-terrestrial body, in search of traces of life and for a greater understanding of the birth of the cosmos.
The robot needs to know where it is for reaching the desired final destination (goal), and often a match between the real-world environment and a map is established. A map is created where the robot and obstacles' position are mapped to have a unique representation of the surrounded environment. After introducing *maps* for mobile robots, a completely known environment or in a partial/unknown can be presents. Based on this, different methods of path algorithm are used.

A navigation algorithm without a map, such as Bug1, Bug2, and DistBug, is often used in a continuously changing environment or if a path has to be travelled only once and therefore does not necessarily has to be optimal. Contrary, if a map is provided, algorithms like Dijkstra or A^* can be applied to find the shortest path offline, before the robot starts driving. Navigation algorithms without maps operates in direct interaction with the robot's sensors while in movement, while with maps require a nodal distance graph that has to be either provided or needs to be extracted from the environment, such as Quadtree method.

Depending on the condition, the mobile robot/rover while travelling must be able to go to any selected place, trying to minimize a cost function, such as time or energy. First, the robot must be able to move fast enough, under control, and safely, avoiding static and moving obstacles (Motion Control problem), to collect knowledge of the environment (Word Modelling problem) and know where it is placed (Localization problem). Finally, it has to move optimally in a large area and some planning is necessary (Planning problem).

A first distinction on what the navigation algorithm exploits or just what features are used in navigation of robots can be made; the mainly two approaches are **Geometric navigation** or **Topologic navigation** [8].

1. **Geometric navigation**: All the environment, such as robot position, start, goal, and obstacles are defined by their own coordinates. The robot position is calculated by odometry (odometry is the use of data's, from motion sensors, to estimate the change in position over time), and often Kalman Filter techniques are used to estimate the position relative to a starting location. It is natural to think that this method generates some errors since the sensors are affected by noise.
   The environment is represented by *occupancy grids* and geometric maps, where elements such as doors, walls, and rocks are modelled. These maps are generated by robot sensors, fusing different images from different robot positions.
   The robot position has error due to the non-ideal sensors that will propagate to the map, leading to a non-perfect localization of the robot in the map. A way to solve this problem is to use a sort of prior information of the environment, such as satellite images or landmarks maps.

2. **Topologic navigation**: This kind of navigation is based on qualitative aspects. The information of the environment is mapped into a graph, where the nodes represent places in which the robot can pass through, while arcs represent a property that the robot owns by passing between the two respective nodes, such as speed or time.
   This kind of navigation focus more on the qualitative aspects of the objects, such as its colour or its shape, is more like humane-kind, we recognize something by its shape and colour.

A second important distinction on how navigation works can be made; it is based on the considered size of the environment. Mainly three different techniques are used: **local navigation**, **global navigation** and the latter is fusing the two previous techniques, **hybrid navigation** [8], [10].

- **Local navigation** is defined as local since uses only sensors and cameras capabilities of viewing the environment to navigate. The robot will have only a partial understanding of the surrounded unknown environment. Only particular algorithms of path planning can be used, since the path from start

to goal is not known. Animals in general has this ability, eyes give us a perception of the local enviroment till where they can see. Often it is based on recognize obstacles or places by qualitative features, such as colour, shape, and so on. In this way, robots can have some memory of the passed objects and on the moving direction. Navigation based on this features is called *local qualitative maps*, and are used to generate a path through the environment using a technique called *traversability*.
Traversability is a technique that given a map of the environment, possible traversable areas in which the robot can pass safety are identified based on analysing the terrain slope and roughness.

- **Global navigation** is used in a perfectly known environment to create a global map and find out a sort of graph, where the nodes stand for places. In general, **roadmaps** are used and the most important are the visibility graphs. These are in general methods that analyse the connectivity between free spaces in a map, which use prior information.

- **Hybrid navigation** combines the best quality of the local and global navigation to better optimize the path.

So far, a general introduction on robot navigation is done, and a complete path planner combines of two parts: ***localization*** and ***path planning***.

2.2 Classical localization

Before imposing a trajectory to be followed, from the initial point (start) to the final one (goal), the robot must know where it is. The problem of addressing robot's position is called ***localization***.

Localization depends on the physical environment and on the available technology, e.g., planetary exploration environment is different from classical application on earth. In an outdoor environment, satellite-based GPS to locate the position of the robot is available, while in an indoor one only local positioning systems such as infrared, sonar, laser or radio beacons can be used.
Robots are equipped with sensors to observe and sense the surrounded environment and monitoring its motion.
A distinction between localization on the earth and planetary exploration is presented. I will refer to the first type as **Classical localization** and to the latter as **Planetary exploration localization**.

Classical localization is the one used on earth applications, uses GPS, beacons or antennas technology to find the global position of the robot in outdoor

environment, while in an indoor, infrared, sonar, laser sensors are used.
In an outdoor environment a simply GPS signal will provide to the robot its global position, while in indoor the use of sensors make possible to find the robot coordinates knowing where the sensors are placed, as shown in figure 2.1.

Knowing the location of the beacons and supposing they send sonar or radio signals at the same time with different frequencies, the robot can determine its position by evaluating the time difference of the signal's arrival time. It is notable to see that with three beacons is possible to perfectly locate the robot position.
This method is useful to find position coordinates but not the orientation of the robot. Orientation can be found considering two consecutives' positions, like a vector, where its origin can represent a previous position and its arrow the actual one, this will give the direction of motion. Comparing the vector with a predefined one, degree difference is extrapolated and so the orientation. This is exactly the method used in GPS technique.
Another idea is to use light emitting homing beacons instead of sonar beacons, e.g., the equivalent of a lighthouse [9].
Depending of the on-board sensors and from the localization system present in the environment, different cases of positioning and orientation are derived, figure 2.2.

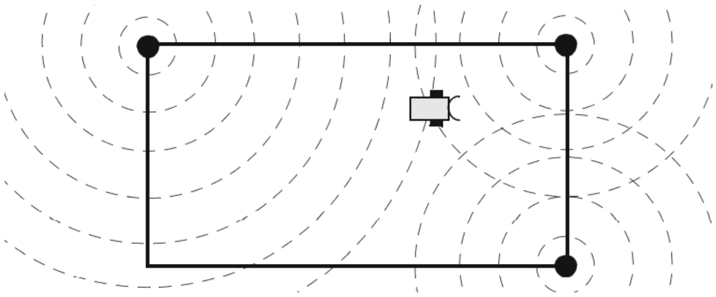


Figure 2.1: Global positioning systems [9].

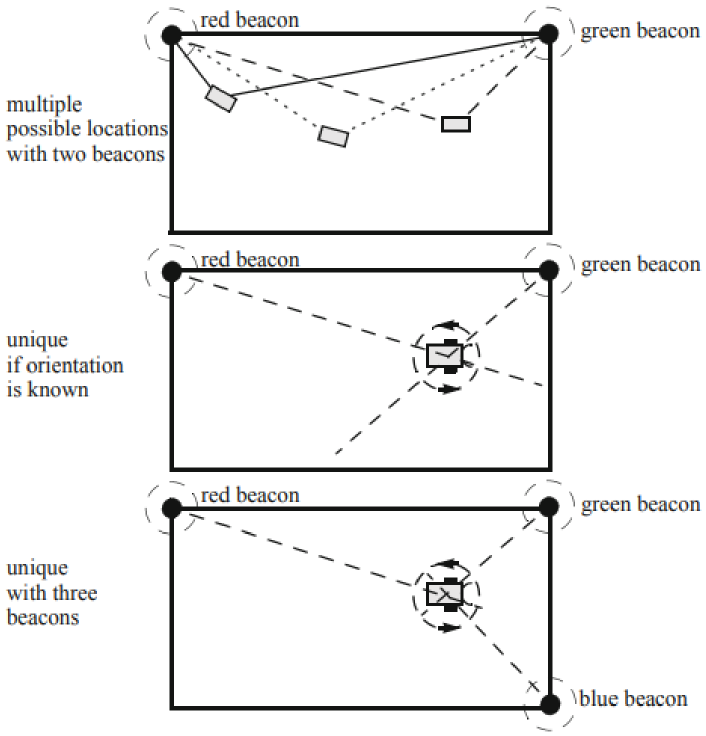

red beacon
green beacon
multiple possible locations with two beacons
red beacon
green beacon
unique if orientation is known
red beacon
green beacon
unique with three beacons
blue beacon

Figure 2.2: Examples of positioning of beacons sensors [9].

2.3 Planetary exploration localization

In planetary exploration environment such as Mars or Moon, the robots cannot rely on the previous explained approaches since they do not have any satellite providing GPS signals, any constellation of orbiting satellites, any sensors, or fixed antennas, like here on Earth.
Possible approaches for localization of the robot in an extra-terrestrial environment are based on using orbiter, lander, and rover sensors.
Two possible approaches are present: **global localization** and **relative localization**.

2.3.1 Global Localization

Global localization was not investigated as much as the relative one, and it is used predominantly for global path planning. During years, many approaches have been studied, such as **Skyline-Based** or **Particle Filter**.

Skyline-Based

Skyline-based approache is the first approach used to estimate the global position of a robot in a planetary exploration context, where skyline matching is done. The method consists in matching a perceive skyline provided by the rover or lander camera with a database of skylines pre-computed on the basis of DEM (Digital Elevation Map). The database encodes skyline signatures, that are matched with the skyline extracted from a panoramic image taken by the rover. This method is very useful to find the global position of the robot in a *lost in space* situation, when no initial position information is provided. The accuracy is in the order of 150 meters, depending on the resolution of the DEM.

Particle Filter

Particle Filter is a well-known framework used to estimate the global position of a robot, and is provided by Monte Carlo Localization (MCL) (Dellarert et al. 1999). In this method, the posterior probability density function, corresponding to the robot global position, is approximated by a set of samples, the so-called particle. By doing this, MCL can consider the multiple possible positions that arise in the lost-in-space problem. Particles are associated to weights which are computed using an observation model and a prior global map of the environment [13].
The key idea, in particle filters, is to represent the robot's belief as a set of N particles, collectively known as M, where each of them consist on a robot configuration x and a weight $w \in [0,1]$. Moving, the robot updates the j-th particle's configuration x_i by first sampling the PDF (probability density function) of $p(x_j \mid d, x_j^{'})$; typically a Gaussian distribution. After, the robot assigns a new weight $w_j = p(s \mid x_j)$ for the j-th particle, the weight normalization occurs such that the sum of all weights is one. Finally, resampling occurs such that only the most likely particles remain.

2.3.2 Relative Localization

This kind of localization is the most important and the most used in the planetary exploration. Different methodologies have been studied: **Dead Reckoning**, **Feature-Based** or **Visual Odometry (VO)**.

Dead Reckoning

Dead reckoning is one of the most important and used technology to understand locally in which direction the robot is moving. Using this kind of localization the robot must rely on its sensors, in particular the use of wheel encoders and/or gyroscopic sensors and calculate the current position by using a previously determined position time, incorporating estimates of speed, heading direction, and course over elapsed time. Naturally, little discrepancy between real and estimated position due to the uncertainty/noise present in all the sensors affects the localization position. This is the reason why this method is applied only for short-term localization.

A particular type of Dead Reckoning is the **Inertial navigation system** (INS) [14]. The INS is a self-contained navigation technique in which measurements provided by accelerometers and gyroscopes are used to track the position and orientation of an object relative to a known starting point, orientation, and velocity. INS typically contains three orthogonal gyroscopes and three orthogonal accelerometers, measuring angular velocity and linear acceleration respectively. By processing signals from these devices, it is possible to track the position and orientation of a robot on which the INS device is mounted.

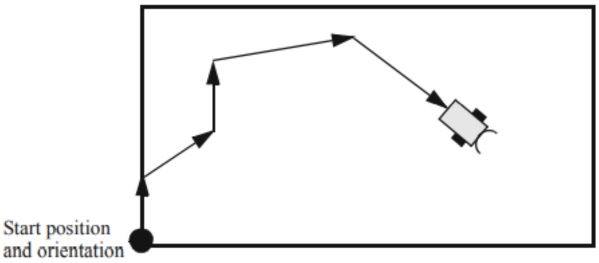


Figure 2.3: Dead Reckoning [9]

Feature-Based

Feature-based approach compares the images provided from the stereo cameras of the rover with images provided from an orbiter for matching of larges rocks, valley, and another unique feature of the environment. Surface rocks are extracted from 3D points cloud produced by the on-board stereo camera and the rover position is determined by finding 2D transformation that gives the maximum number of matches between the rocks detected from the two sources. This method suffers of poor result when the environment has few rocks or is surrounded by many of them.

Visual Odometry

Visual Odometry (V0) is based on analysing the images provided by the stereo cameras. The VO is mainly characterized by two steps:

- *Mapping* makes extensive use of dense stereo matching and subsequent merging of partial maps.

- *Visual Localization* refers to the use of images, in particular the succession of frames to determine and estimate position and orientation of the robot, this kind of localization refers to visual simultaneous localization and mapping (vSLAM).

This kind of method is more robust and accurate since no sensors are used to estimate position of GPS signal or IMU data.

These kind of techniques are used by European Space Agency in the SPARTAN (SPAring Robotics Technologies for Autonomous Navigation) and SEXTANT (Spartan EXTension Actibity - Not Tendered) programs. These programs were born to develop computer vision algorithm for visual navigation, mapping and localization.
An example of application is seen in the ExoMars mission [11].
The localisation on board of the ExoMars rover is composed of the **Absolute Localisation** (AbsLoc) and **Relative Localisation** (RelLoc) modules.

- **The Absolute Localisation (AbsLoc)** module updates the on-board attitude estimate when the rover is stationary, e.g. before starting to follow a commanded path. The module is capable of performing a gravity (roll and pitch) update of the estimated attitude using the accelerometer measured average acceleration as an estimate of the gravity vector. AbsLoc does not have the capability to estimate the heading autonomously, but it can perform a heading update with a heading value provided from ground.
- **The Relative Localisation** module is responsible to update the estimates of the position and attitude of the rover relative to the MLG frame while it is driving. The rover defines a Mars Local Geodetic (MLG) frame, with its origin at the rover's initial location, z axis pointing up, the x-y axes in the local horizontal plane, the x axis pointing east. Motion of the rover is tracked in this frame.

2.4 Coordinate Systems

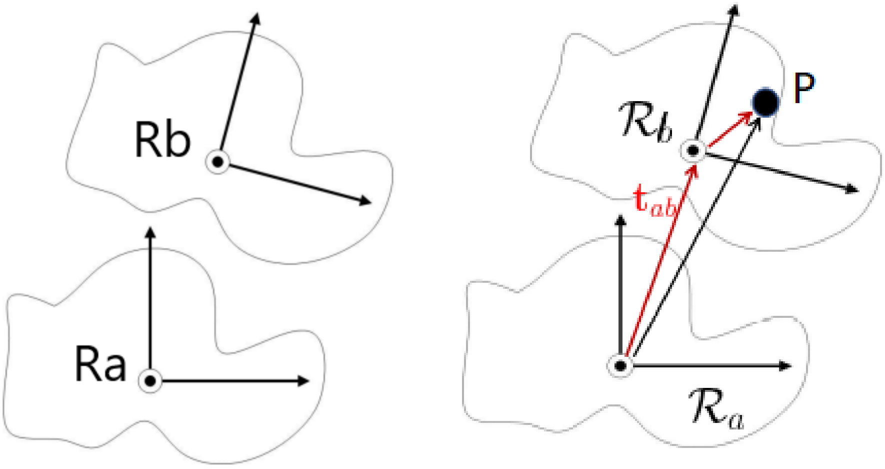
In several applications it is important to establish a map or to plan a path. The path is generally specified in *global or world* coordinates, while the robot's position and orientation refers usually in its *local coordinates*.
Since the path is specified in global coordinates, a method to convert the robot local coordinates in a global one is needed and **Rototranslation matrices** are used, since every transformation in the plane can be expressed as translation + rotation, this kind of representation is called **Homogeneous Transformation**.

2.4.1 Rotation Matrix

Firstly, let's analyse the rotation between two frames, in particular consider two orthonormal frames (three perpendicular axis with three respective unit vector i,j,k).
Consider two frames R_b and R_a, figure 2.4(a):

$$R_a = O_a i_a j_a k_a \quad R_b = O_b i_b j_b k_b$$

Frame R_a and R_b are related by the so called *Rotation Matrix*, that gives the possibility of switching from one frame to the other, referring coordinates of frame B in frame A and vice versa. Frame B axis can be expressed in frame A by:



(a) *Rotation of two frames.* **(b)** *Rototranslation.*

Figure 2.4: Reference systems

$$\begin{cases} i_b = \alpha_1^1\mathbf{i}_a + \alpha_2^1\mathbf{j}_a + \alpha_3^1\mathbf{k}_a \\ j_b = \alpha_1^2\mathbf{i}_a + \alpha_2^2\mathbf{j}_a + \alpha_3^2\mathbf{k}_a \\ k_b = \alpha_1^3\mathbf{i}_a + \alpha_2^3\mathbf{j}_a + \alpha_3^3\mathbf{k}_a \end{cases} \rightarrow R_b^a = \begin{bmatrix} \alpha_1^1 & \alpha_1^2 & \alpha_1^3 \\ \alpha_2^1 & \alpha_2^2 & \alpha_2^3 \\ \alpha_3^1 & \alpha_3^2 & \alpha_3^3 \end{bmatrix}$$

Matrix R_b^a stands for the Rotation Matrix that converts coordinate of points of frame B in frame A.
Let's see how the process works. Consider a vector defined in frame B, its coordinates in the frame A are found by using the following passages:

$$\begin{aligned} v = v_x^b i_b + v_y^b j_b + v_z^b k_b = (\alpha_1^1 v_x^b + \alpha_1^2 v_y^b + \alpha_1^3 v_z^b) i_a + \\ (\alpha_2^1 v_x^b + \alpha_2^2 v_y^b + \alpha_2^3 v_z^b) j_a + \\ (\alpha_3^1 v_x^b + \alpha_3^2 v_y^b + \alpha_3^3 v_z^b) k_a = v_x^a i_a + v_y^a j_a + v_z^a k_a \end{aligned}$$

Vector a and b will be related the the formula below:

$$\overline{v}^a = \begin{pmatrix} v_x^a \\ v_y^a \\ v_z^a \end{pmatrix} = \begin{bmatrix} \alpha_1^1 & \alpha_1^2 & \alpha_1^3 \\ \alpha_2^1 & \alpha_2^2 & \alpha_2^3 \\ \alpha_3^1 & \alpha_3^2 & \alpha_3^3 \end{bmatrix} \begin{pmatrix} v_x^b \\ v_y^b \\ v_z^b \end{pmatrix} = R_b^a \overline{v}^b$$

There are three fundamentals rotation in a 3D environment and every Rotation matrix can be expressed as a rotation about the three fundamental axis of a frame, x,y,z or i,j,k:

- Rotation of $\alpha°$ about z axis (k):

$$Rot(z,\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

- Rotation of $\alpha°$ about y axis (j):

$$Rot(y,\alpha) = \begin{bmatrix} \cos(\alpha) & 0 & \sin(\alpha) \\ 0 & 1 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) \end{bmatrix}$$

- Rotation of $\alpha°$ about x axis (i):

$$Rot(x,\alpha) = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

A rotation R in the 2D or 3D plane can be expressed by a summation of the three fundamental rotations:

$$R = Rot(z,\alpha) + Rot(y,\alpha) + Rot(x,\alpha)$$

2.4.2 Homogeneous transformation

Homogeneous Rototranslation Matrix (T_b^a) is an operation that joins a rotation and a translation.

Consider now a point P in frame B, its coordinates in frame A can be found through a Rotation matrix R_b^a ($Rot(\alpha)$) and a translation t_{ab} ($Trans(r_x, r_y)$).
I will make explicit reference to the figure 2.4 b.
Let's define O_a and O_b the origin of frame A and B respectively, coordinates of point P in frame A are given by a simple formula:

$$\overline{O_aP} = \overline{O_aO_b} + \overline{O_bP}$$

where,

$$\overline{O_bP} = x_P^b i_b + y_P^b j_b + z_P^b k_b$$

$$\overline{O_aP} = x_P^a i_a + y_P^a j_a + z_P^a k_a$$

Now, **Homogeneous Rototranslation Matrix** (T_b^a) comes out as sum of these two components.

$$\begin{pmatrix} x_P^a \\ y_P^a \\ z_P^a \end{pmatrix} = \bar{t}_{ab}^a + R_b^a \begin{pmatrix} x_P^b \\ y_P^b \\ z_P^b \end{pmatrix} \rightarrow \begin{pmatrix} x_P^a \\ y_P^a \\ z_P^a \\ 1 \end{pmatrix} = \underbrace{\begin{bmatrix} R_b^a & \bar{t}_{ab}^a \\ 0 & 1 \end{bmatrix}}_{T_b^a} \begin{pmatrix} x_P^b \\ y_P^b \\ z_P^b \\ 1 \end{pmatrix}$$

where, R_b^a is the Rotation Matrix 3x3 to transforms vector's coordinates, defined also as ($Rot(\alpha)$), while T_b^a is the Homogeneous 4x4 matrix for trasforming coordinates of points.

Example
After introducing the previous notation, let's apply it to a real case for switch local coordinates to global one.
Assume a robot global position [r_x,r_y] and a global orientation ϕ. Suppose the robot senses an object in its local coordinates at [$O_{x'}$, $O_{y'}$], the global coordinates of this object [O_x,O_y] will be given by:

$$[O_x, O_y] = Trans(r_x, r_y) + Rot(\phi) \cdot [O_{x'}, O_{y'}]$$

Consider the exemple in figure 2.5. The local object P, marked with the + symbol, has local coordinate [0,3]. The global robot's position is [4,2.5] and its global orientation is $\alpha = 30°$. Applying the theory seen above, we can write the global position of the object as:

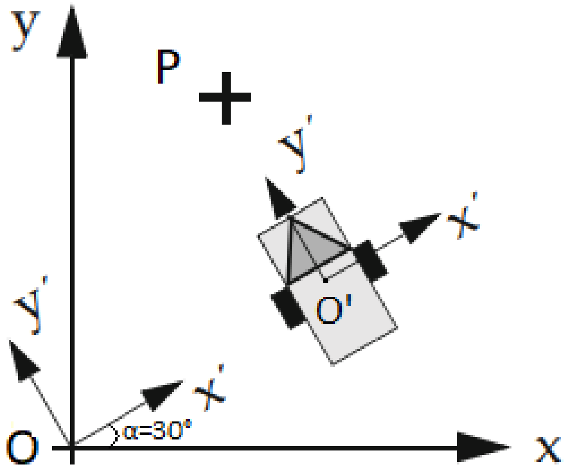


Figure 2.5: Global and Local coorsinate systems Transformation

$$\begin{aligned}[O_x, O_y] &= Trans(4{,}2.5) + Rot(30°) \cdot [0{,}3] \\ &= Trans(4{,}2.5) + [-1.5{,}2.6] \\ &= [2.5{,}4.6]\end{aligned}$$

Below all the detailed computation are present.

Proof. Since we are in 2D enviroment, in the above case, a 3x3 matrix is sufficient:

$$\begin{bmatrix} O_x \\ O_y \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2.5 \\ 0 \end{bmatrix} + \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} 0 \\ 3 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} O_x \\ O_y \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2.5 \\ 0 \end{bmatrix} + \begin{bmatrix} -3\sin(\alpha) \\ 3\cos(\alpha) \\ 0 \end{bmatrix}$$

for $\alpha = 30°$ this comes to:

$$\begin{bmatrix} O_x \\ O_y \\ 0 \end{bmatrix} = \begin{bmatrix} 4 \\ 2.5 \\ 0 \end{bmatrix} + \begin{bmatrix} -1.5 \\ 2.6 \\ 0 \end{bmatrix} = \begin{bmatrix} 2.5 \\ 4.6 \\ 0 \end{bmatrix}$$

□

2.5 Environment Representation

In a 2D environment there are mainly two basic representations; **configuration space** and **occupancy grid**. In configuration space, dimensions of the environment and position (coordinates) of the obstacles are given, and so a 2D environment can be created. In the other hand, in occupancy grid the 2D environment is not "continuous" with line segments but is represented in a discreate way, by a given resolution, where a single grid correspond to a portion of the real environment, white pixels represent free space while the black pixels obstacles. In figure 2.6 the two representations can be seen.

These two formats can be easily transformed into each other, by simply a discretization or combination of pixels.

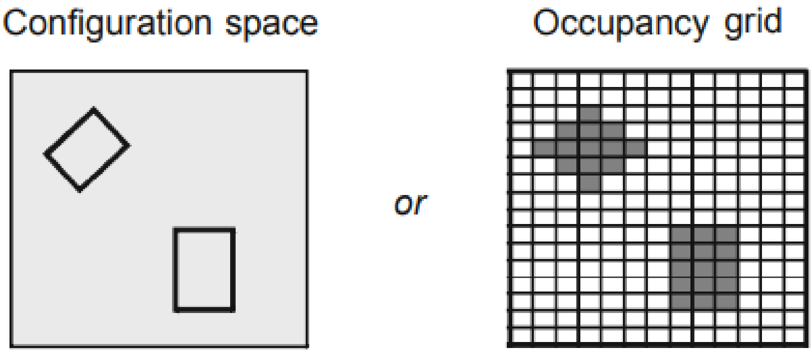


Figure 2.6: Basic environment representations

Many algorithms work directly on the environment description (configuration space or occupancy grid), some others, such as Dijkstra or $A^{\star}$, require a *node distance graph* as input.
A discussion of the two mainly methods for path planning is done: **node distance graph** and **traversability maps**.

2.6 Distance graph

A **Distance graph**, figure 2.7, is a description environment at a higher level, not all information is retained, but only elements useful to describe a path from initial to the final position. A node distance graph is simply composed of two elements, nodes, identifying position in the environment, and the weighted arcs, representing the distances between two nodes.

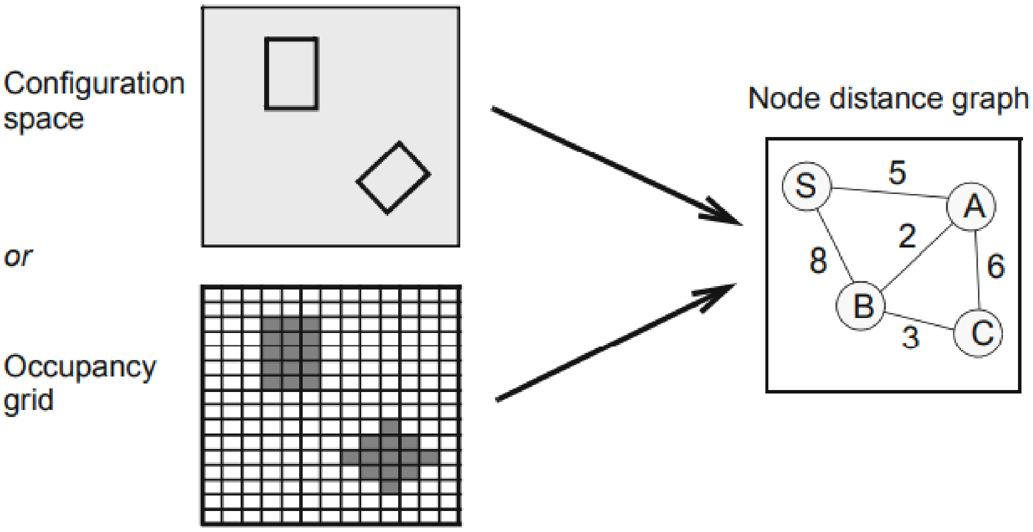


Figure 2.7: Node distance graph derivation [8].

Many algorithms can work directly taking as input the configuration space or occupancy grid, but the majority work taking as input a *Node Distance Graph.*

First, let's analyse how a distance graph from the environment representation can be derived.
An intuitive and easy way to proceed is to use occupancy grid to find out our distance graph. In the distance graph each node corresponds to a particular location in the grid, so, a brute force is to treat each pixel of the grid as a node in the graph. If the input is represented by a configuration space a conversion in occupancy grid is done by "printing it" on a canvas at the desired resolution.
Anyway, this approach has some problems, first due to the high number of the nodes obtained the graph will be very huge, for large environment this method is not feasible that means slower algorithm and high computational complexity. Second, the path generated will be suboptimal since the algorithm may stack in one of the many local optima present. Third, constraints on steering angle are present, since pixels are like a matrix only turning angles that are multiples of 45° are possible [8].
More efficient methods are presented for constructing a distance graph from the environment representation:

- Quadreee
- Visibility Graph

2.6.1 Quadtree

To avoid the previous problems, a method called **Quadtree** can be used. A shorter graph will be created, and no steering constraint will be present.
A quadtree is a tree data structure in which each internal node has exactly four children. Quadtrees are the most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions.

To generate a quadtree, the 2D environment is recursively divided in four quadrants, where each of them become a node or a leaf. If the quadrant is free or full of obstacles, it becomes a terminal node, a leaf, while if the quadrant is partially full, it will be divided in another four quadrants recursively, until each quadrant becomes a leaf.

Example
I have created an example showing the procedure of recursively division in four quadrants, which can be seen in picture 2.8. After constructing the quadtree graph, we can pick all the free nodes and use them into the distance graph. The construction of the distance graph is made by linking each free node to the others, by the way, only nodes that can be link without intersecting an obstacle (in the example link c-e is not put in the graph).

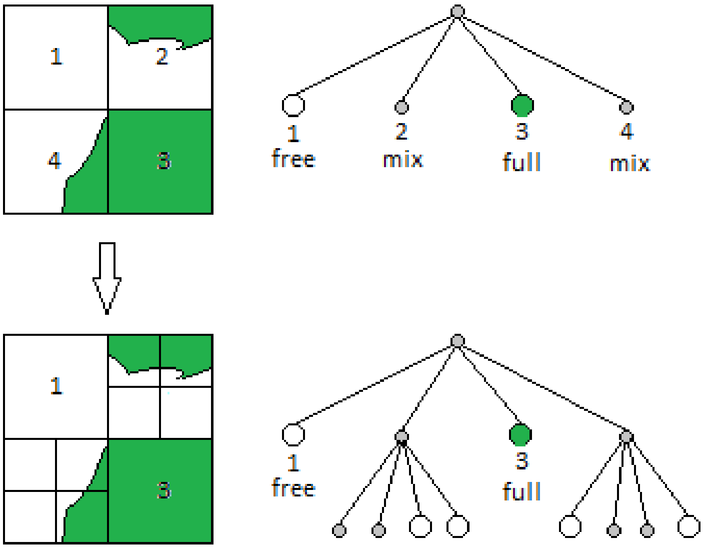


Figure 2.8: Quadtree graph

We can observe the mentioned passages in the picture 2.9.
After constructing this graph, a path planning algorithm can be applied to find a

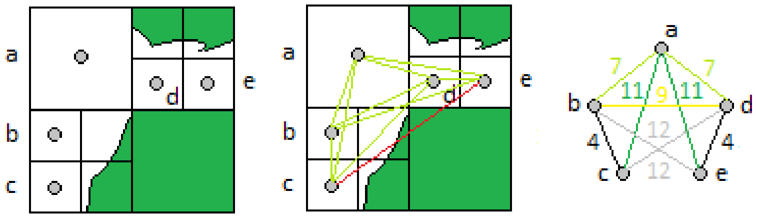


Figure 2.9: Distance graph from Quadtree

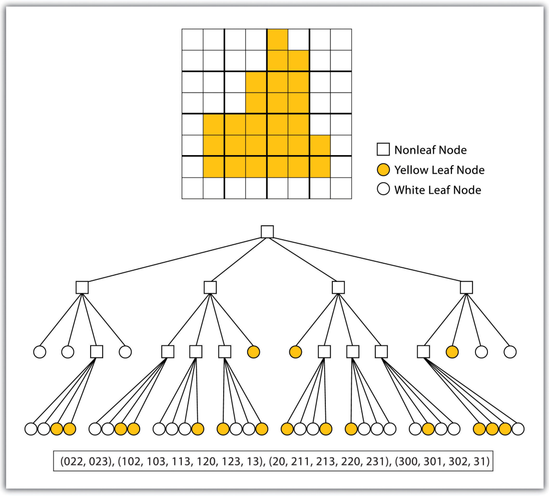


Figure 2.10: A more complex Quadtree example

route, like $A^{\star}$.

Another example of quadtree construction is shown below (2.10).

2.6.2 Visibility graph

A visibility graph is a graph of intervisible locations, typically for a set of points and obstacles in the Euclidean plane. Each node in the graph represents a point location, and each edge represents a visible connection between them.

While in the Quadtree the nodes represent free spaces in the environment, in Visibility graph the nodes are chosen in a different way. In particular, corner points of obstacles, start and goal position are used as nodes, and lines represent a free path between the two nodes.

As shown in figure 2.11, a graph is constructing by linking each node to the others, deleting thus lines that intersect an obstacle. The remaining lines, represent which paths can the robot follow to reach the goal. In red are represented thus lines that intersect an obstacle and so cannot be followed, while in green only the free paths.

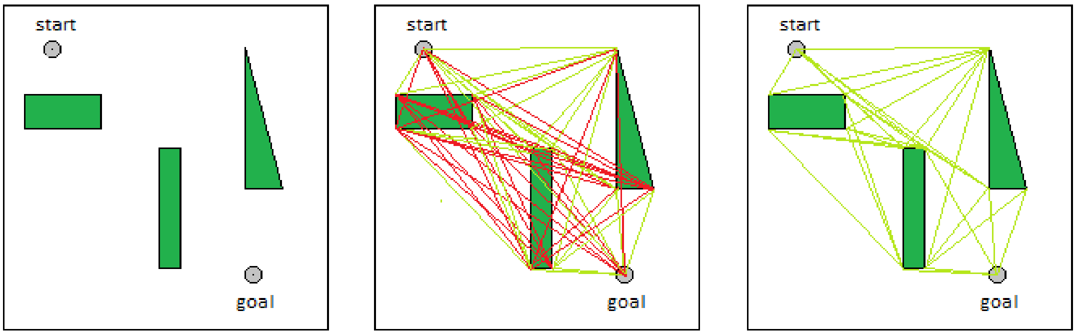


Figure 2.11: Node selection process for visibility graph

2.7 Traversability Map

Let analyse another method to understand from an image where the robot can pass through. In general, in planetary exploration context, there are many trajectories that a robot can follow from the start to the goal, for examples the one that minimizes the distance or the time. Usually, in planetary context environment, a path is choosing taking into account which is the safest for the robot.

A safety path can be chosen among different properties, such as less obstacles, less steep slopes and climbs or consistency of the terrain. Terrain consistency is very important since the robot may get stuck in soft soil, despite a harder terrain is preferable since more grip for the robot's wheel is present.

Traversability Map can help us in choosing which path is the safest for the robot.

2.7.1 Traversability Map and Traversability Index

Traversability Map are based on analysing the images provided by the cameras and constructing a sort of grid map using some logics based on the so called *traversability index*, which marks which areas are more traversable for the robot, and pick those grids that are the safest for the robot. The generated trajectory will be located within these grids.

First time, traversability index was introduced form mobile robots (rovers) by [16], where a detailed and advance explanation is done.

Traversabilty index is developed using the framework of fuzzy logic and is expressed by linguistic fuzzy sets that quantify the suitability of the terrain for traversal based on its physical properties, such as slope and roughness. The Traversability Index is used for classifying planetary surfaces and provides a simple means for incorporating the terrain quality data (out to about 10 meter) into the rover navigation strategy. A set of fuzzy navigation rules is developed using the Traversability Index to guide the rover toward the safest and the most traversable terrain. In addition, another set of fuzzy rules is developed to drive the rover form its initial position to a user-specified goal position. The two rules' sets are integrated in a two-stage procedure for autonomous rover navigation without a priori knowledge about the environment [16].

Traversability index was firstly used in in-door mobile robots operating in structured environments [17], [18], and only to the end of the 90's was apply to outdoor unstructured environments like planetary exploration [16].

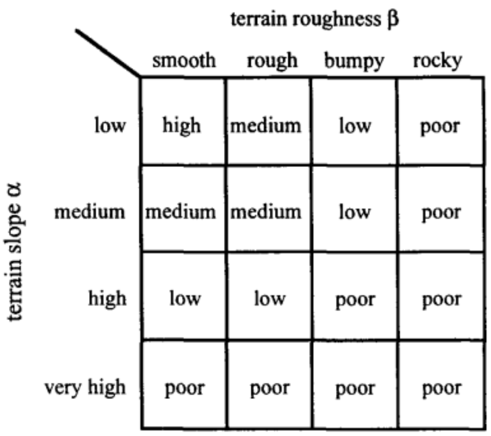
A basic approach is to evaluate the terrain by consider two parameters, α and β, representing respectively terrain slope and terrain roughness.

- **Terrain slope** α is measured by stereo camera mounted on the rover [19]. Given the slope of the terrain in degrees, four linguistic fuzzy sets, represented by α, can be produced; {LOW, MEDIUM, HIGH, VERY HIGH}. Each linguistic word refers to a specific degree range and can be either positive and negative, representing a mountains/hills or craters. With this kind of discretization, the precise slope of terrain is not needed since a single linguistic word represents a range of possible slopes.
- **Terrain Roughness** β can be estimated by using onboard stereo camera based of range map methods [19]. A simpler alternative approach is based collecting the information regarding the dimension and concentration of rocks [16]. Using images from cameras, estimation of rock concentration (density) and sizes, produces an average of the rock-size in the terrain. Rock density is defined over the set {LOW, HIGH}, while rock sizes by {SMALL, LARGE}.
  Combining the two previous set, parameter β is defined by the following fuzzy set: {SMOOTH, ROUGH, BUMPY, ROCKY}.

Now, having α and β, which represents the terrain properties, a fuzzy table is constructed, and will gives us the traversability index τ. Defining with $A = \{A_1, A_2, A_3, A_4\}$ the discrete output α and with $B = \{B_1, B_2, B_3, B_4\}$ the discrete output β, a cartesian product $T = A\text{x}B$ can be obtained, where T is the output linguistic set representing the traversability of the terrain {POOR, LOW, MEDIUM, HIGH}.
The resulting traversability table can be seen in figure 2.12, where a sixteen fuzzy table representing the traversability index is present. From this table we can understand that a terrain in non-traversable when is ROCKY or when slope is VERY HIGH and can be traversable for LOW slopes and SMOOTH terrain.
The four linguistic fuzzy sets for τ can be interpreted depending primary on the characteristics of the rover, since a rover with larger wheel can traverse grater rocks, means that the traversable output can be different, and secondary considering some personal interpretation of how safety is a soft terrain or which kind of slope is defined as LOW:

- **POOR** $\longrightarrow$ Highly-Impassable terrain
- **LOW** $\longrightarrow$ Impassable terrain
- **MEDIUM** $\longrightarrow$ Passable terrain

- **HIGH** ⟶ Highly-Passable terrain



terrain slope α \ terrain roughness β	smooth	rough	bumpy	rocky
low	high	medium	low	poor
medium	medium	medium	low	poor
high	low	low	poor	poor
very high	poor	poor	poor	poor

Figure 2.12: Rule set for Traversability Index τ [16].

2.7.2 Basic path planning using Traversability index

Traversability index is used for imposing rover heading and speed velocity, which are fundamental for controlling and for navigate the rover to the safest and most traversable terrain. Velocity v and heading angle change $\delta\theta$ are determined by an another fuzzy table involving as input the traversability index [16].

Definition of parameter v and heading angle $\delta\theta$

- First of all, a partition of the terrain around the rover is executed up to a radius r, usually 10 meters, in particular five different sectors are identified; front, front-right, front-left, right, left. The front one is straight to the rover, front-right and front-left are at ±45° while left and right at ±90°. Traversability index $\tau_f, \tau_{fl}, \tau_{fr}, \tau_r, \tau_l$ are computed for the five regions, higher index represents higher safety and traversable area. The rover chooses the higher index and will turn the wheel for that region, if two or more regions have equal value, closest region and so smallest turn angle is preferable. Since five regions are present around the robot, a fuzzy table with five fuzzy linguistic sets is establish; HARD-LEFT, LEFT, ON-COURSE, RIGHT, HARD-RIGHT.

- Velocity v is chosen based on the value of τ_{max} of the Traversability Index τ in the chosen region. Depending on the values of τ different velocity are imposed:

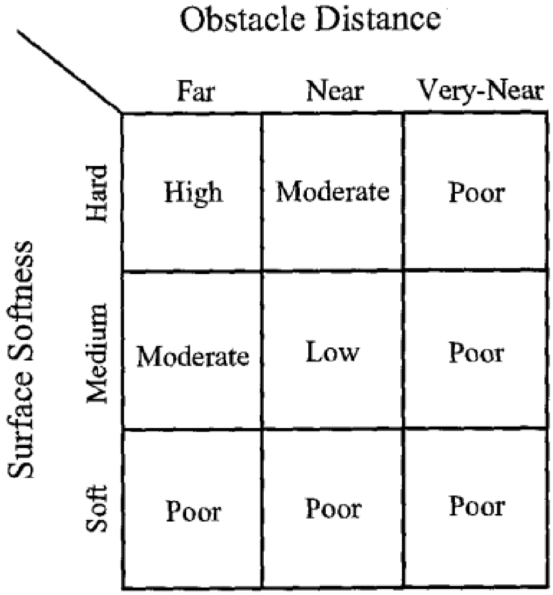
- If τ_{max} is POOR, then v is STOP
- If τ_{max} is LOW, then v is SLOW
- If τ_{max} is MEDIUM, then v is MODERATE
- If τ_{max} is HIGH, then v is FAST

2.7.3 Advance path planning using Traversability index

Advanced treatments were done in literature, where distinction between local, regional, and global Traversability are made [20]. **Local traversability** is related with obstacle detection and surface softness with the use of the sensors, **regional traversability** despite uses camera images for categorize the terrain, while **global traversability** index is obtained from the terrain topographic map, based on recognition of mountains, crates and so on, like in topological navigation, nature-feature are used.
All of this three Traversability index has got four linguistic labels {POOR, LOW, MODERATE, HIGH}, corresponding to: unsafe, moderately-unsafe, moderately-safe, safe.

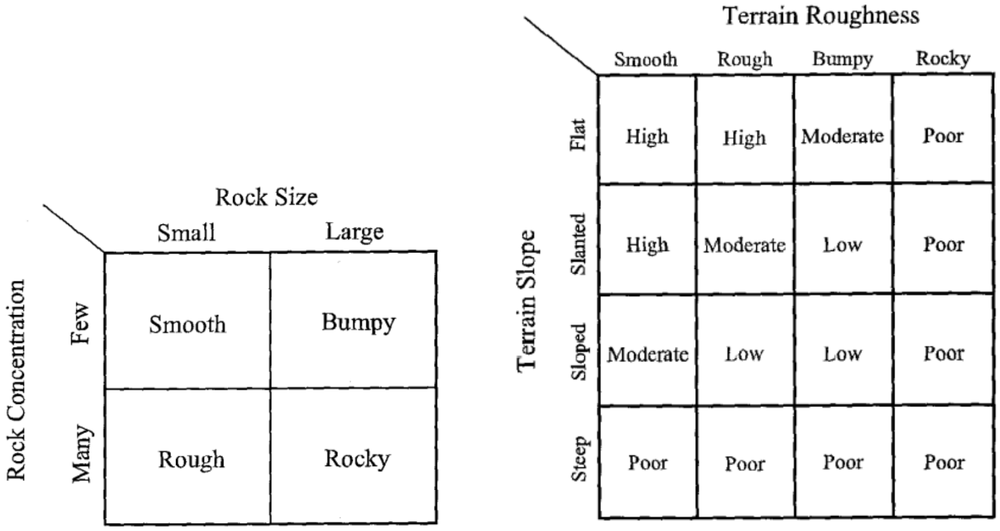
- **Local Traversability** is used up to 0.5 meters, so very close to the robot. It has got four fuzzy sets POOR, LOW, MODERATE, HIGH that depend of the two paramenters: *distance* d_o from the closest obstacle, categorized as VERY-NEAR, NEAR, FAR, and *surface softness* γ represented by SOFT, MEDIUM, HARD [20]. The combination of these two inputs give the Local index represented in figure 2.13.

- **Regional Traversability Index** covers a zone up to 5 meters and takes into account pysical properties of the terrain, such as slopes and roughness, that are extracted from camera images. As told before Terrain Roughness is defined using two parameter, rocks size and rocks concentration, that gives us four output fuzzy sets SMOOTH, ROUGH, BUMPY, ROCKY, figure 2.14(a). Now, combining terrain roughness and terrain slopes, regional traversability Index is find, figure 2.14(b).



Surface Softness \ Obstacle Distance	Far	Near	Very-Near
Hard	High	Moderate	Poor
Medium	Moderate	Low	Poor
Soft	Poor	Poor	Poor

Figure 2.13: Rule set for Local Traversability Index [16].



Rock Concentration \ Rock Size	Small	Large
Few	Smooth	Bumpy
Many	Rough	Rocky

(a) *Rule set for Rocks Concentration.*

Terrain Slope \ Terrain Roughness	Smooth	Rough	Bumpy	Rocky
Flat	High	High	Moderate	Poor
Slanted	High	Moderate	Low	Poor
Sloped	Moderate	Low	Low	Poor
Steep	Poor	Poor	Poor	Poor

(b) *Regional Traversability Index.*

Figure 2.14: Rule set for Regional Traversability

- **Global Traversability Index** operates up to tens of meters resolution and is based on terrain map. From an image representing all the environment, position of obstacles such as mountains, crates and in general natural features are extracted. Detection of hill/mountain peaks are identified by algorithm, like counter detection and so on [21]. This information are translated into a fuzzy linguistic sets for constructing the Global Traversability Map, that for each area is classified as POOR, LOW, MODERATE, HIGH, that indicates the quality of the traversable area. An important consideration can be made on global map, since they are global every obstacle is detected in a whole, different traversability index to the same obstacle can be present. For examples a mountain has a POOP traversability on the peak, while getting down the traversability index increases because of the decreasing slope, figure 2.15.

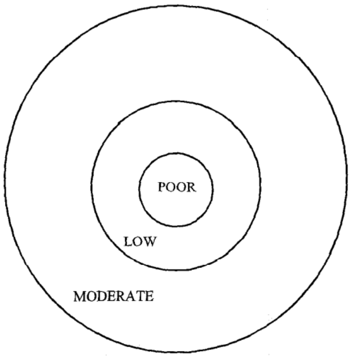
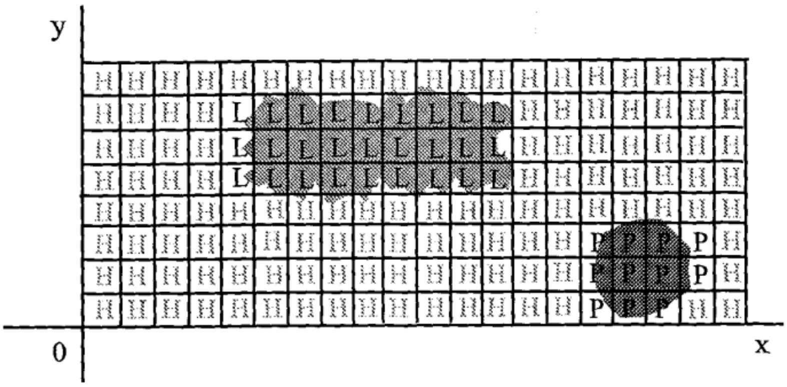


Figure 2.15: Traversability map of a mountain or hill [16]

Conclusion

Making data fusion between prior and posterior data's, a construction of a global map of the obstacles and of the area around the robot is created. In the global map representing the obstacles, a grid of the image is extracted and each grid is classified based on the lowest Traversability Index in that area, figure 2.16.

Having the map in figure 2.16, we can run a path planning algorithm where only the high traversable area are considered and the low/poor one are considered as obstacles.
Reconducting the problem of path planning on a grid map we can use all the algorithm based on **node distance graph**, remember that having the occupancy grid or space configuration representation both can be easily convert in a node distance graph.



H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	H	H	L	L	L	L	L	L	L	L	L	H	H	H	H	H	H	H
H	H	H	H	L	L	L	L	L	L	L	L	L	H	H	H	H	H	H	H
H	H	H	H	L	L	L	L	L	L	L	L	L	H	H	H	H	H	H	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	P	P	P	P	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	P	P	P	P	H
H	H	H	H	H	H	H	H	H	H	H	H	H	H	H	P	P	P	H	H

Figure 2.16: Traversability grid on a map [16].

Chapter 3

State of the Art

In this chapter, we are going to review the state-of-the-art approaches to the methods and algorithms used for path planning.
Navigation of robots and rovers are done by following the two mainly steps of *localization* and *path planning.* The first one, localization, was deeply discussed in chapter 2, while the second, path planning, is deeply investigated in this charter.

The path planning is firstly done by evaluating the obstacles position and then create a path that avoid them. In chapter 2 the mainly methods to localize the obstacle and the rover were presented, from the environment representation and node distance graph to traversability map used to created a 2D map grid where each grid is basically evaluated as traversable or not for the rover.

In this chapter a review and a study on NURBS methods for path planning algorithm generation is presented. There will be algorithm that works directly in the configuration space and others that use occupancy grid for generating the distance graph as input to the algorithm.
Despite, the method that will be presented in chapter 5 will used directly the configuration space to detect the obstacle and to create a sort of traversability map, in which regions that need to be avoided are highlighted.

3.1 Path planner Algorithms

Let's now analyse how a path is generate by an algorithm and which kind of algorithms are mainly used for this purpose. A path planning algorithm can work in different ways, what distinguish one from the other is what kind of input is the algorithm taken. In the following pages we will analyse different algorithms with different inputs, such as maps of the obstacles or distance graphs or eighter sensors.

3.1.1 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest path between nodes in a graph. Dijkstra's original algorithm find the shortest path between two given nodes, moreover in literature there are many versions and modifications of this algorithm.

The algorithm requires as input a node distance graph with all the information regarding the distances between nodes. Time complexity of the implementation is $O(e+v)^2$, where e are the edges and v are the nodes. Additional requirement is memory for storing the found path, bigger graph imply higher complexity and more memory space.

The algorithm starts with initializing all the distances between nodes to infinity, while the start node for definition has got distance equal zero. The second step is to store in a vector all the nodes already visited to not visit them again, and finally to loop on all the neighbours of each node for finding the closest.
A basic algorithm can be seen below [9]:

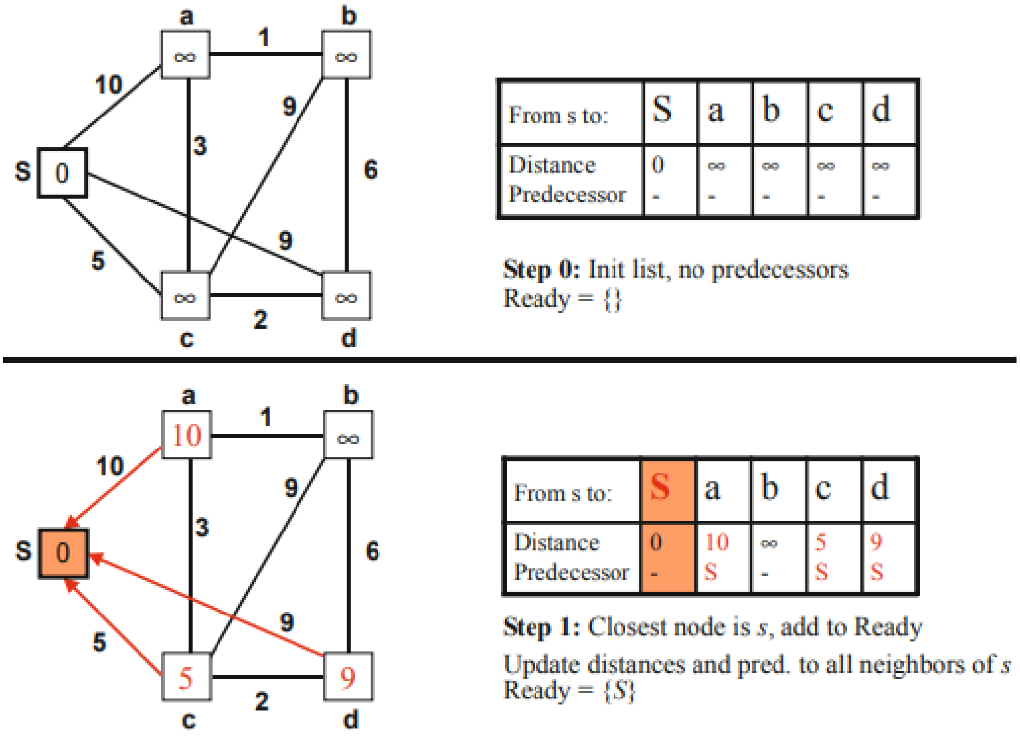
```
%1 Step (Initialization)
Set start distance to 0, dist[s] = 0
Set all the other distcnces to infinite , dist[i] = infinity
Set a vector called predecessor to 0, pre[i] = 0

%2 Step (Create variable for storing nodes)
Ready = {}

%3 Step (loop for neighbors)
FOR loop until all nodes are in Set_nodes
    Select node n with shortest distance that is not in Ready
    Ready = Ready + n

    FOR each neighbor node m of n
        IF dist[n]+edge(n,m) < dist[m] %evaluate shotest path
            dist[m] = dist[n]+edge(n,m)
            pre[m] = n
```

Let's apply the previous algorithm in a real node distance graph, analysing all the passages. Let's start by analysing figure 3.1 and figure 3.2 images that show in order all the steps and updates that the algorithm makes to each node [9].



From s to:	S	a	b	c	d
Distance Predecessor	0 -	∞ -	∞ -	∞ -	∞ -

Step 0: Init list, no predecessors
Ready = {}

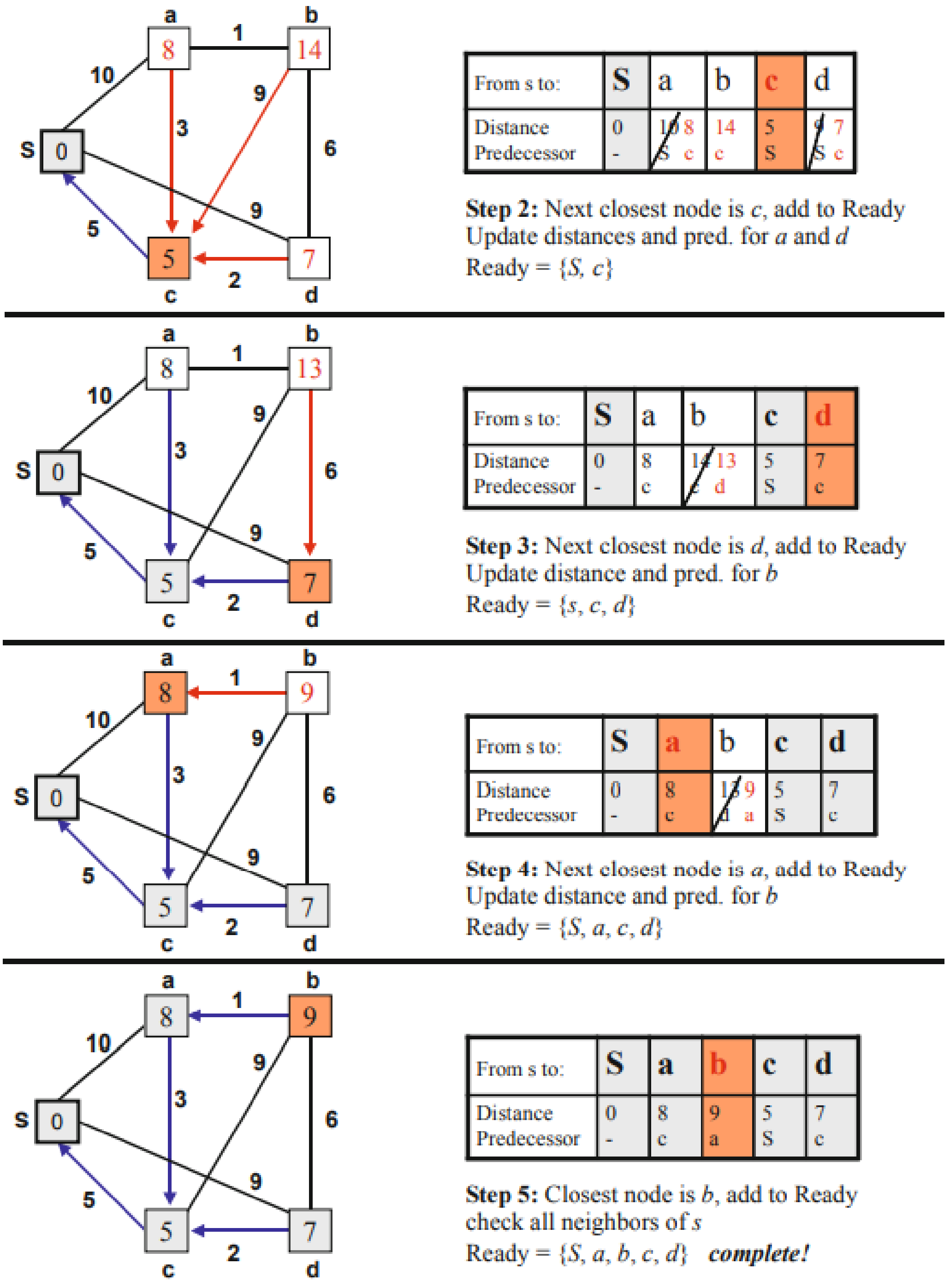
From s to:	S	a	b	c	d
Distance Predecessor	0 -	10 S	∞ -	5 S	9 S

Step 1: Closest node is *s*, add to Ready
Update distances and pred. to all neighbors of *s*
Ready = {*S*}

Figure 3.1: Dijkstra's algorithm step 0 in step 1

At the begin, the *Ready* set is empty and the algorithm stats imposing the *Start* node distance equal to zero and the others to infinity. Then, as show in step 1, the closest node S is taken, which will be the Start node with zero distance for definition. Node S is added to Ready set to not consider it again and a loop in its neighbour is performed to calculate the distances from it to nodes; a, c, and d by recording its predecessor and the relative distance by taking into account the sum of the edges needed to get to that node.

Next, in step 2, it is to consider which among the nodes not yet present in Ready is the closest, considering nodes a, b, c, and d. The closest will be c with distance 5, Ready = {S,c}. The table is updated according to the node c evaluating the distances from it and updating the predecessor accordingly. In step 3 we repeat the same steps as before, choosing the node d as closest, Ready = {S,c,d}. In step 4, since al the neighbours were visited, the algorithm returns to its predecessor, in this case node c, and evaluates the closest from it, following the same passages. The algorithm terminates when all nodes are present in Ready.



From s to:	S	a	b	c	d
Distance Predecessor	0 -	~~10~~ 8 ~~S~~ c	14 c	5 S	~~9~~ 7 ~~S~~ c

Step 2: Next closest node is *c*, add to Ready
Update distances and pred. for *a* and *d*
Ready = {*S*, *c*}

From s to:	S	a	b	c	d
Distance Predecessor	0 -	8 c	~~14~~ 13 ~~c~~ d	5 S	7 c

Step 3: Next closest node is *d*, add to Ready
Update distance and pred. for *b*
Ready = {*s*, *c*, *d*}

From s to:	S	a	b	c	d
Distance Predecessor	0 -	8 c	~~13~~ 9 ~~d~~ a	5 S	7 c

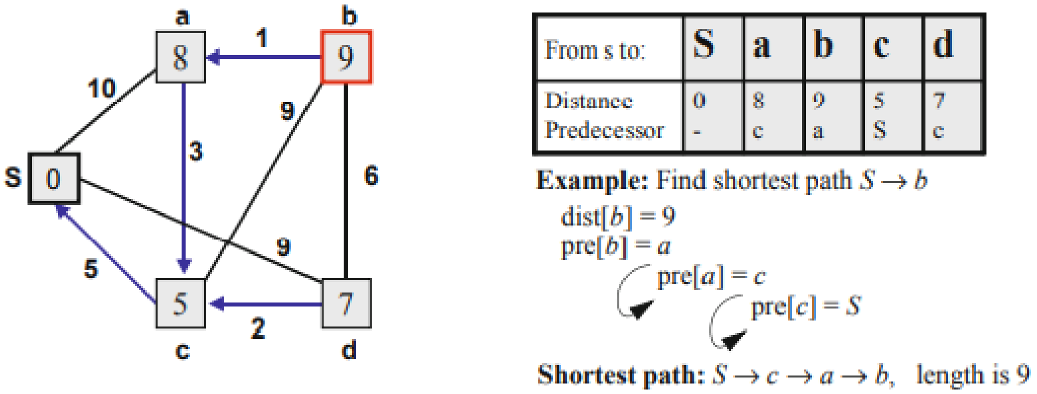
Step 4: Next closest node is *a*, add to Ready
Update distance and pred. for *b*
Ready = {*S*, *a*, *c*, *d*}

From s to:	S	a	b	c	d
Distance Predecessor	0 -	8 c	9 a	5 S	7 c

Step 5: Closest node is *b*, add to Ready
check all neighbors of *s*
Ready = {*S*, *a*, *b*, *c*, *d*}   ***complete!***

Figure 3.2: Dijkstra's algorithm from step 2 to step 5

When all nodes are present in Ready we found out a table like in figure 3.3, where all nodes S,a,b,c,d with their shortest distance to its predecessor are stored. In figure 3.3 an examples of path from node S to node b in presented [9].



From s to:	S	a	b	c	d
Distance Predecessor	0 -	8 c	9 a	5 S	7 c

Example: Find shortest path $S \rightarrow b$

$\text{dist}[b] = 9$

$\text{pre}[b] = a$

$\text{pre}[a] = c$

$\text{pre}[c] = S$

Shortest path: $S \rightarrow c \rightarrow a \rightarrow b$, length is 9

Figure 3.3: Shortest path calculation

3.1.2 $A^\star$ Algorithm

$A^\star$ algorithm is very similar to Dijkstra. It is a heuristic algorithm for computing the shortest path from a given start node to a given goal node.
Drawbacks similar to Dijkstra are present, storing intermediate paths, variables, and nodes take up memory. Complexity becomes $O(k \cdot \log_k(v))$ for v nodes with branching factor k and in worst case becomes quadratic.
The algorithm, in addition to the distance graph, requires a parameter called **lower bound**, which represents a heuristic value that estimates the cost of the cheapest path from a node n to the goal.
For this reason $A^\star$ is an informed search algorithm, or best-fit search. It is formulated in terms of weighted graph, which aims to find a path having the smallest cost (least distance travelled). The calculation of the cost is done summing two terms; cost of the path to reach a node n plus an estimate of the cost required to extend the path all the way to the goal from that specific node n.
In mathematical terms:

$$f(n) = g(n) + h(n)$$

where:

- n is the next node of the path.
- $g(n)$ is the cost of the path from the start node to n.
- $h(n)$ is a heuristic function that estimates the cost of the cheapest path from node n to the goal.

The heuristic function is problem-specific and is always underestimate, and if it is admissible, so is never overestimates the cost, it will return a least-cost path from start to goal.

$A^\star$ has three mainly properties:

- **Admissible**: If a solution exists, it is an optimal one.
- **Complete**: $A^\star$ is also a complete algorithm, meaning if a solution exists the path is found in a finite amount of time.
- **Optimal**: $A^\star$ is optimally efficient for a given heuristic optimal search algorithm. No other optimal algorithms will expand fewer nodes and find a better solution.

Consider figure 3.4, where a distance graph is represented with their relative distances and each node has its own *lower bound* distance to goal, notice that such lower bound is always an underestimate of the real distance to the goal b.

First step is to consider all distances from the start node S to its nearby nodes, in particular starting from S we have three choices:

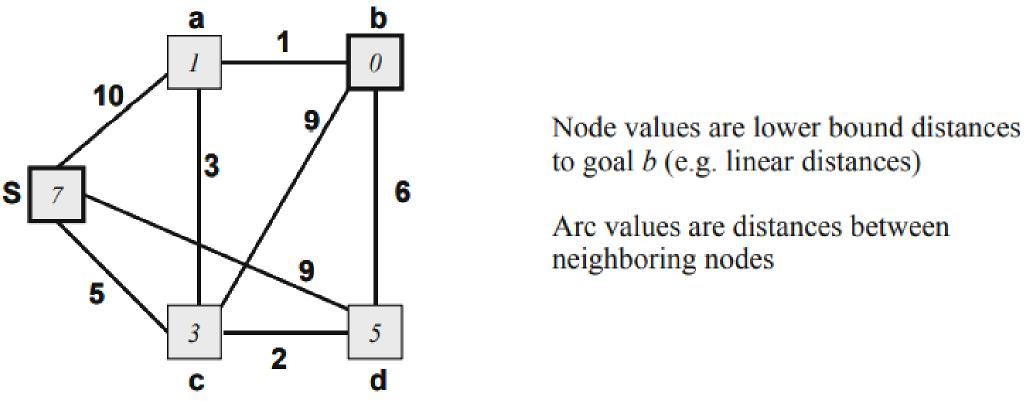
- {S,a} with min. length $10 + 1 = 11$
- {S,c} with min. length $5 + 3 = \mathbf{8}$
- {S,d} with min. length $9 + 5 = 14$

Since we are using a "best-first" algorithm, the shortest estimated path will be {S,c}, the next expansion from node c will be:

- {S,c,a} with min. length $5 + 3 + 1 = \mathbf{9}$
- {S,c,b} with min. length $5 + 9 + 0 = 14$
- {S,c,d} with min. length $5 + 2 + 5 = 12$

It turns out that {S,c,a} is the closest, so the expansion from node a is:

- {S,c,a,b} with min. length $5 + 3 + 1 + 0 = \mathbf{9}$



Node values are lower bound distances to goal *b* (e.g. linear distances)

Arc values are distances between neighboring nodes

Figure 3.4: $A^{\star}$ example

The algorithm terminates when goal node b is reached. The shortest possible path is {S,c,a,b}, and the corresponding distance is: $dist = 9$

The timing complexity of the algorithm depends of the estimate of the lower bound, closer is the estimates to the real distance shorter will be the execution time.

3.1.3 Potential Field Method

The previous two algorithms take as input a node distance graph, let's take a look to methods that use directly as input the environment representation, in particular *configuration space representation*.

Potential Field Method or PFM generates a map with virtual attracting and repelling forces. Start point, obstacles and walls are repelling forces since the robot must avoid these objects, while goal position is attracting; force strength is inversely proportional to object distance. Robot simply follow the force field, such as a charge particle moves in an electric field.
Figure 3.5 represents the distribution of a potential field in the environment. In general PFM generates a total force on the robot depending of its location, it can be expressed as:

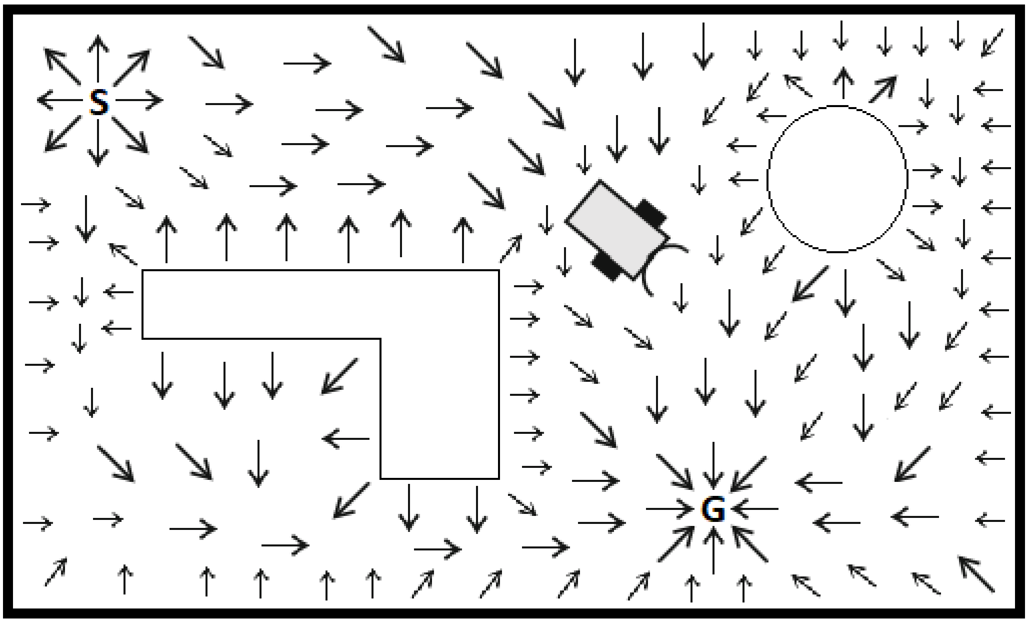


Figure 3.5: Potential Field

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

where,

$$U(q) = \text{artificial potential field}$$

$$U_{att}(q) = \text{attractive field}$$

$$U_{rep}(q) = \text{repulsive field}$$

We can pass from the potential field to the force by considering physics formula. The above potential field generates a force $F(q)$:

$$F(q) = -\nabla U(q) = -\nabla U_{att}(q) \cdot \nabla U_{rep}(q)$$

$$F(q) = F_{att}(q) + F_{rep}(q)$$

where,

$$F(q) = \text{artificial force}$$

$$F_{att}(q) = \text{attractive force}$$

$$F_{rep}(q) = \text{repulsive force}$$

PFM can be seen as sum of an attractive component and a repulsive one [15]:

Attractive Potential: It is like a convex set, a positive paraboloid that tends to zero when the robot gets closer to the goal, figure 3.6(a):

$$U_{att(q)} = 1/2\xi \|q - q_{goal}\|^2$$

$$F_{att}(q) = -\xi(q - q_g oal)$$

Repulsive Potential: Creates a potential barrier around obstacles that cannot be crossed by the robot, figure 3.6(b):

$$U_{rep}(q) = \begin{cases} 1/2\eta(1/p(q) - 1/p_0)^2 & \text{if } p(q) \leq p_0, \\ 0 & \text{if } p(q) > p_0. \end{cases}$$

$$F_{rep}(q) = \begin{cases} \eta(1/p(q) - 1/p_0)1/p^2(q)\nabla p(q) & \text{if } p(q) \leq p_0, \\ 0 & \text{if } p(q) > p_0. \end{cases}$$

where,

$$\eta \longrightarrow \text{Positive scaling factor}$$

$$p(q) = min_{q \in C_{obstacle}} \|q - q^{'}\| \longrightarrow \text{Distance from postion q to obstacle C}$$

$$p_0 \longrightarrow \text{Positive constant of the Obstacle C}$$

In figure 3.6 [15], the total potential field is represented.

This method has an important drawback, sometimes the robot can get stack in local minima. Local minima are points that has zero force, where the attractive and repulsive total force is zero. In these regions the robot will stop and will never reach the goal.

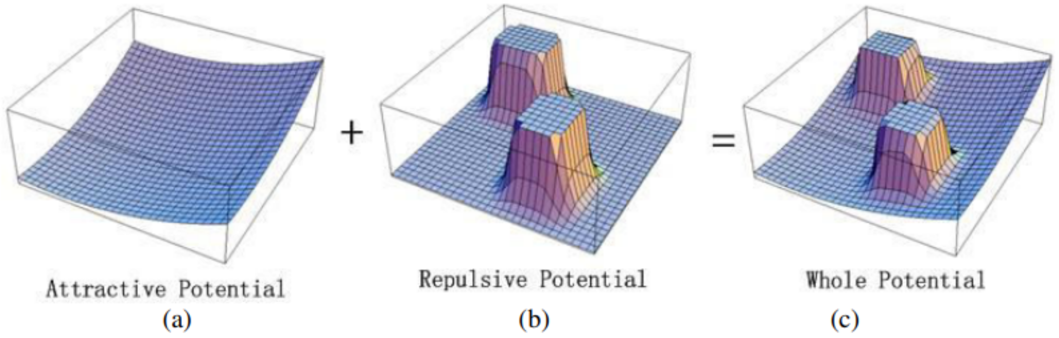


Figure 3.6: Combination of repulsive and attractive potential

3.1.4 Wandering Standpoint Algorithm

It is an algorithm for local path planning that uses as input distance sensor measurements. The algorithm is very simply, the robot simply try to reach the goal point in a direct line and when an obstacle is detected with the robot's sensors, turns left or right depending of the smallest angle and continue with boundary-following around the obstacle until goal direction is free, figure 3.7.
A drawback arises when the environment has many obstacles, the robot continues with boundary-following and never reaches the goal.

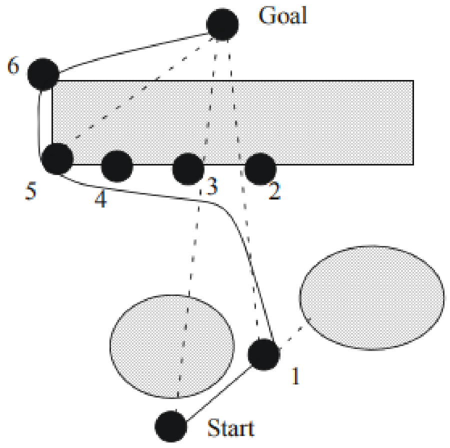


Figure 3.7: Wandering standpoint [8].

3.1.5 Bug Algorithms

Bug algorithms have mainly three kind of versions: **Bug1** [**22**]**, Bug2** [**22**] and **DistBug** [**23**].

These algorithms for local path planning guarantees converges and will return a path if exits or reports if the goal is unreachable. Inputs for algorithms are Odometry (sensor such as encoder wheel), goal position, and touch sensors for Bug1 and Bug2, while distance sensor for DistBug.

Bug1 working principle: The robot drives straight towards the goal until an obstacle is hitted (*hit point*), then it proceeds into boundary-following recording the shortest distance to goal (*leave point*) and when the hit point is reached again it drives to the leave point continuing straight to the goal, figure 3.8.

Bug2 working principle: The robot traces an imaginary straight line M from the start position to the goal and follows it until an obstacle is touched (*hit point*), then boundary-following is executed until a point on the M line is reached (*leave point*) and finally continue following the straight line M, figure 3.8.

DistBug working principle: Drive straight toward the goal until an obstacle is reached, then boundary-following until the goal is visible again or there is sufficient space toward the goal, continue straight for that point (*leave point*). If no point is found (*leave point*) the robot will reach the hit point, meaning the goal is unreachable, figure 3.9.

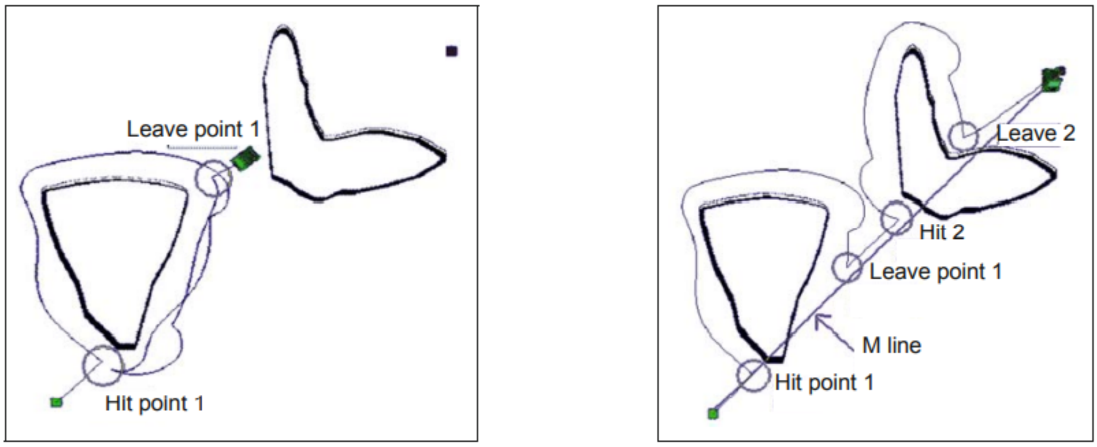


Figure 3.8: Bug1 (left) and Bug2 (right) comparison [24].

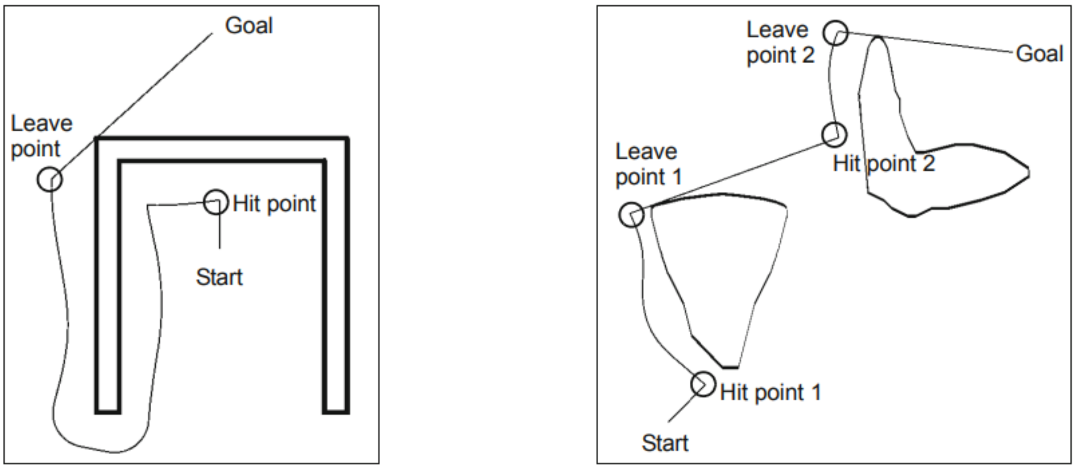


Figure 3.9: DistBug algorithm [24].

3.2 Path planner with NURBS

NURBS are a particular kind of curves where their shape can be controlled by particular points, a detail explanation is presented in the next chapter.
The algorithm will take as input a top-view image of the environment, and an obstacle detection algorithm will find the obstacles directly using the configuration space representation. Finally, a sort of *traversability map*, where the robot cannot pass is created, and a path is generated through an algorithm using NURBS curves. The algorithm uses some features of the previous methods, such as boundary-following, imaginary curves, or attractive/repulsive effects.

Chapter 4

NURBS mathematics

In this chapter, NURBS mathematics will be introduced and used as basis for a new *Path Planner Method* in the context of planetary exploration rover.
Path Planners based on these generalized curves have important properties, such as lower complexity, less usage memory to store the curve, greater accuracy respect to other curves of same degrees, and very smooth trajectory which always admits derivative on the curve.

NURBS, states for ***N**on **U**niform **R**ational **B**asis **S**plines*, is a mathematical model using basis splines (B-splines) for representing curves and surfaces.
NURBS is one of the most recent and advanced methods for interpolation of points based on spline interpolation, which generates curves with smooth and speedy features.

4.1 Background notions on curves

Before introducing NURBS, the most generic parametrization, a review on mathematical curve representations and on the main methods for creating them are presented. These concepts are fundamental to fully understand the mathematics behind NURBS and why they can be so important.

4.1.1 Mathematical form of curves

In geometric modelling, a mathematical branch for the representation of an object's geometry, there are maily three forms for representing curves: *parametric*, *implicit* and *explicit* form.

- **Parametric form:** A parametric curve is defined by its parametric equations. The number of parametric equations that express the curve depends on the space dimension in which the curve lives.
For simplicity let's consider a 2D plane, a curve can be expressed by the following two parametric equations:

$$\begin{aligned} x &= f(t) \\ y &= g(t) \quad a \leq t \leq b \end{aligned} \tag{4.1}$$

where the x-coordinate and the y-coordinate of a point on the curve are expressed as a mathematical function of a parameter t, that takes value in a closed interval $t \in [a, b]$, usually normalized as $t \in [0,1]$.
It is intuitive to understand that by changing the parametric value t is possible to move along the curve, from the start point ($t = 0$) to the end ($t = 1$), this will set the direction of travel.

Consider the circle in figure 4.1, having radius 1, the parametric form would be:

$$\begin{aligned} x &= \cos(t) \\ y &= \sin(t) \quad 0 \leq t \leq 2\pi \end{aligned} \tag{4.2}$$

The parametrization is not unique and many more can be found.
Equation (4.2) can be written in a more compact notation:

$$F = (x(t), y(t)) = (\cos(t), \sin(t)) \quad 0 \leq t \leq 2\pi \tag{4.3}$$

where F represents a function which express the pair of coordinates of a point on the curve.
Using parametric form, derivatives of equation (4.3) become:

$$F' = (x'(t), y'(t)) = (-\sin(t), \cos(t)) \quad 0 \leq t \leq 2\pi \tag{4.4}$$

Functions $x(t)$ and $y(t)$ are assumed to be continuous with a sufficient number of continuos derivatives. A parametric curve is said to be of **class** F^r if the function F has continuous derivatives up to order r.

- **Implicit form:** The curve is expressed by an implicit equation of the form: $f(x, y) = 0$, representing an algebraic equation.
$f(x, y)$ represents the implicit relation between x and y, and can have different degrees: linear, quadratic, and so on.
Let's have a look on how the same circle can be expressed in implicit form:

$$f(x, y) = 0 \quad \longrightarrow \quad x^2 + y^2 - 1 = 0 \tag{4.5}$$

- **Explicit form:** Explicit form is a particular case of the two previous representations. The independent variable t can be expressed as a function of x and y. The coordinates of a point will take an explicit form:

$$y = H(x) \quad or \quad x = K(y) \tag{4.6}$$

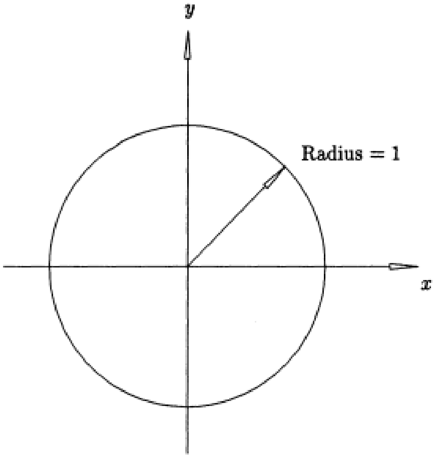


Figure 4.1: Circle of radius 1 in a xy plane

The parametric and implicit form will often be used to represent B-Spline polynomials and generic curves.

4.1.2 Methematical methods for creating curves

Different mathematical methods for creating curves are present, depending on the cases.
If the shape of the curve is already known, a prior equation can be directly associated, like for the circle example above. In the other hand, often only points in 2D or 3D plane are present and tracing a curve that interpoletes or approximates them is necessary.

The need for **interpolation** of points comes out and four mainly classes of interpolation exist in mathematical modelling:

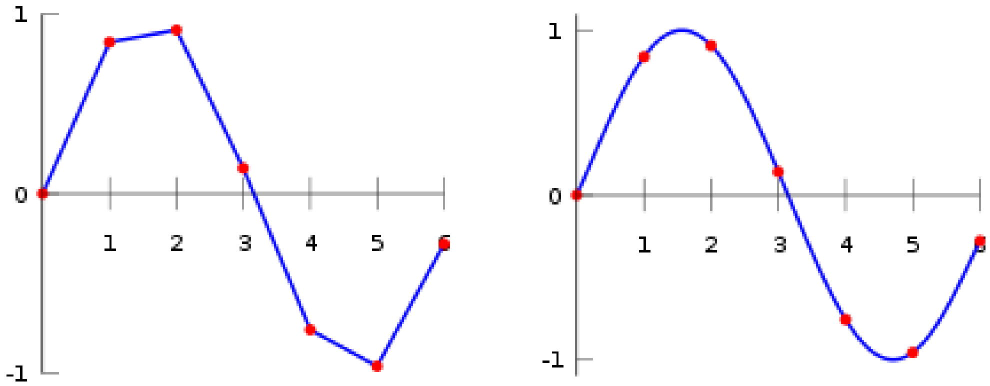
- **Piecewise constant interpolation:** It is the simplest method and consists in taking as estimate of a point its specific value, assigning it to its neighbourhood. It is preferable in terms of **speed** and **semplicity**.

- **Linear interpolation:** It is the mainly method used when a straight line need to pass between two points.
  This kind of interpolation takes as input two different points, $P_a(x_a, y_a)$ and $P_b(y_b, y_b)$, and traces a straight line through them using two terms: *offset* and *angular coefficient*.

  The general form of a line on a xy-plane is; $y = ax + b$, where a represents the angular coefficient while b the offset. To find it out consider two points, P_a and P_b, and the following passages:

$$\begin{aligned} y = y_a + (y_b - y_a)\frac{x - x_a}{x_b - x_a} \\ \frac{y - y_a}{y_b - y_a} = \frac{x - x_a}{x_b - x_a} \\ \frac{y - y_a}{x - x_a} = \frac{y_b - y_a}{x_b - x_a} \end{aligned} \tag{4.7}$$

  Linear interpolation is a good choice to represent linearly the points.
  Drawbacks appear when the degree is higher than one, **low approximation** is present, and **no differentiable points** are found where two different slops intersect.
  The above drawbacks can be seen in figure 4.2, where a sin function is interpolated. The linear function is the sum of different linear functions over different intervals with non-differentiable points at the junction of two linear functions.



(a) *Linear interpolation of a* sin *function.* (b) *Polynomial/Spline interpolation of a* sin *function.*

Figure 4.2: Interpolation examples.

- **Polynomial interpolation:** Linear interpolation uses a linear function while polynomial uses higher degree to interpolate points, leading to a **lower approximation error**. Despite, this kind of interpolation is more **complex** and **computationally slower**.
  In figure 4.2, the sin function in now interpolated with a 6^{th} degree polynomial function [Wikipedia]:

  $$f(x) = -0.0001521x^6 - 0.003130x^5 + 0.07321x^4 - 0.3577x^3 + 0.2255x^2 + 0.9038x$$

- **Spline Interpolation:** Since interpolation of complex curves or functions leads to high degree polynamial function, spline interpolation can solve this problem.
  In fact, using spline interpolation, single polynomial functions are used to interpolate only finite intervals, which will then be sum up to obtain the entire curve. This method will lead to a **lower polynomial degree**.
  An example is shown below, when trying to interpolate the same sin function:

  $$f(x) = \begin{cases} -0.1522x^3 + 0.9937x, & \text{if } x \in [0,1] \\ -0.01258x^3 - 0.4189x^2 + 1.4126x - 0.1396, & \text{if } x \in [1,2] \\ 0.1403x^3 - 1.3359x^2 + 3.2467x - 1.3623, & \text{if } x \in [2,3] \\ 0.1579x^3 - 1.4945x^2 + 3.7225x - 1.8381, & \text{if } x \in [3,4] \\ 0.05375x^3 - 0.2450x^2 - 1.2756x + 4.8259, & \text{if } x \in [4,5] \\ -0.1871x^3 + 3.3673x^2 - 19.3370x + 34.9282, & \text{if } x \in [5,6] \end{cases}$$

  It is easy to see the differences between the two types of interpolation, polynomial and spline, where the first has a degree of six while the second has got a maximum of three.
  In practice, the same interpolation with a lower degree is archive.

- **Parametrization:** Another method used to generate curves is to parametrize points with an independent variable, constrained in an interval. Same operation seen before in section 4.1.2

4.2 Evolution of curves

After introducing in which way curves are generated, I will focus on **spline interpolation** and **parametrization** since are the most flexible in term of computational effort, complexity, speed, degree, and very low approximation error. These are the reason why generic curves, and in particular NURBS, are important.

Before introducing NURBS curves, Bernstein polynomial, Bèzier curve and B-Spline parametrization are introduced since NURBS are a generalization of these.

4.2.1 Bernstein Polynomial

Bernstein polynomials are a kind of polynomial interpolation based on a linear combination of *Bernstein basis polynomial.*
Bernstein basis polynomial are polynomials of different degree with general expression:

$$b_{i,n}(x) = \binom{n}{i} x^i (1-x)^{n-i} \quad \text{for } \; i = 0, \dots, n$$

$$\downarrow$$

$$b_{i,n}(x) = \frac{n!}{i!(n-i)!} x^i (1-x)^{n-i} \quad \text{for } \; i = 0, \dots, n \tag{4.8}$$

Now, Bernstein polynomial B_n can be computed using the formula below:

$$B_n(x) = \sum_{i=0}^{n} \beta_i b_{i,n}(x) \tag{4.9}$$

where n represents the degree of the Bernstein polynomial curve, which is given by $n+1$ basis sum of degree at most n, while β_i are called weights coefficients.

Example of 3^{rd} degree Bernstein curve
Let's start writing its basis using the previous formulas:

$$b_{0,3} = (1-x)^3, \quad b_{1,3} = 3x(1-x)^2, \quad b_{2,3} = 3x^2(1-x), \quad b_{3,3} = x^3$$

As expected for a 3^{rd}, $4(n+1)$ basis are present, figure 4.3
Bernstein polynomial will be given by the summation of its basis, weighted by beta coefficients:

$$B_n(x) = \beta_0 b_{0,3} + \beta_1 b_{1,3} + \beta_2 b_{2,3} + \beta_3 b_{3,3}$$

Figure 4.4 shows a demostration of two possible Bernstein curves (3^{rd} degree) created using two distinct vectors as coefficients beta (weights) in an interval between [0,1], generated by a MATLAB script:

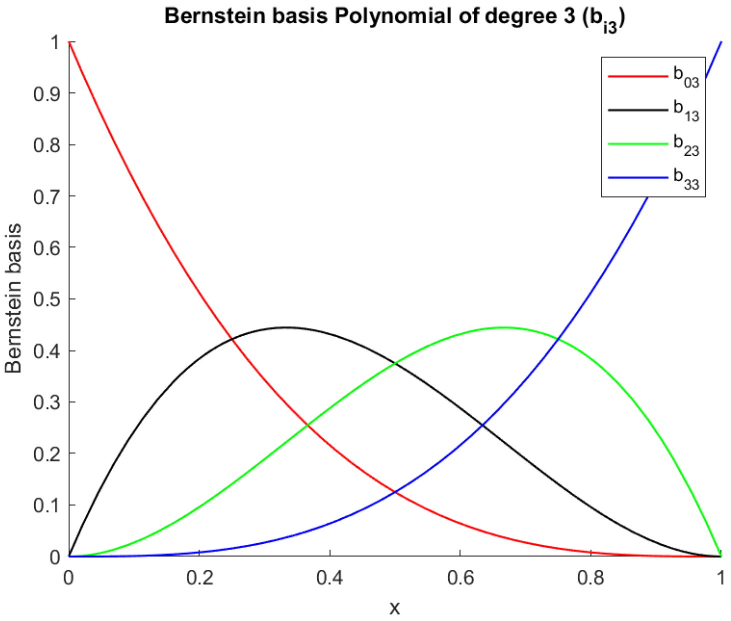
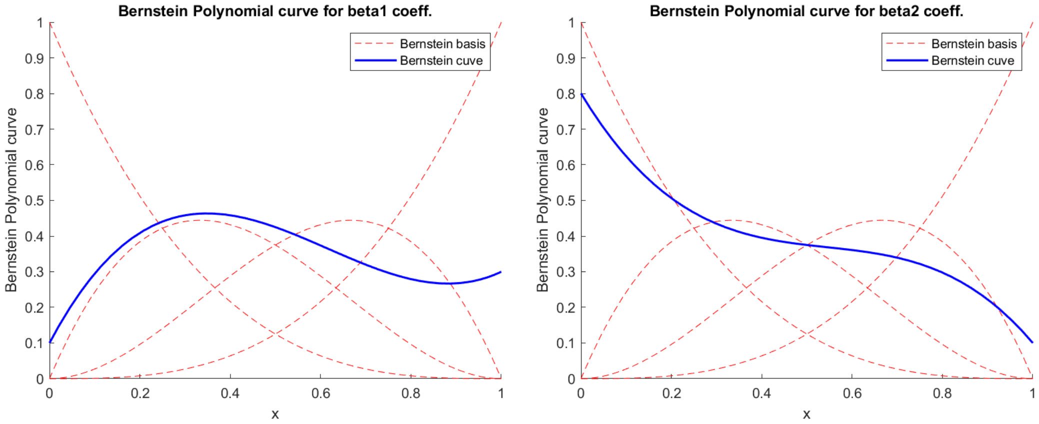


Figure 4.3: Bernstein polynomial basis (3^{th} degree)



(a) *Bernstein Curve with* β *=[0.1 0.9 0.1 0.3]*

(b) *Bernstein Curve with* β *=[0.8 0.1 0.6 0.1].*

Figure 4.4: Bernstein Polynomial Curves

4.2.2 Bézier Curve

Bézier curves are parametric curves that use Bernstein polynomial as basis and a set of discrete points, called *control points*, to weight the curves. They are expressed by a function that uses an independent variable $t \in$[0,1].
Often, Bézier curves are used to approximate real shape or to give a mathematical

formulation of such curves that are too complex.
General expression of Bézier curve can be defined as follows:

$$q(t) = \sum_{i=0}^{n} P_i b_{i,n}(t) \quad 0 \le t \le 1 \tag{4.10}$$

where P_i are the control points, called also Bézier points, which are defined in the space's curve, while $b_{i,n}$ are the Bernstein basis defined in section 4.2.1, figure 4.3. By connecting consecutively all the control points, the **control polygon** is obtained, where inside can be found the parametric curve. The control polygon is a **convex** polygon, this property will be inherited from NURBS.

Example of Bézier parametric curve:
From (4.10) is intuitive to understand that for a n^{th} degree Bézier curve, $n+1$ points and $n+1$ Bernstein basis are needed.
Consider now a 3^{th} $(n = 3)$ degree Bézier curve $q(t)$, it will be given by the summation of four terms $P_i \cdot b_{i,n}$, where:

$$b_{0,3} = (1-x)^3, \quad b_{1,3} = 3x(1-x)^2, \quad b_{2,3} = 3x^2(1-x), \quad b_{3,3} = x^3$$

and

$$P_0 = (0{,}0.25) \;\; P_1 = (0.33{,}0.75) \;\; P_2 = (0.67{,}0.33) \;\; P_3 = (1{,}0.58)$$

Bézier curve $q(t)$ will be:

$$q_3(t) = P_0 b_{0,3} + P_1 b_{1,3} + P_2 b_{2,3} + P_3 b_{3,3}$$

Final result can be seen in figure 4.5.

Two important properties can be seen from figure 4.5:

1. Start and end curve points always coincide with P_0 and P_n, and can be written in mathematical form as:

$$q(t=0) = P_0 \quad \text{and} \quad q(t=1) = P_n$$

2. The obtained Bézier curve is always contained in the control polygon. The control polygon is always a *convex hull* set.
Definition of convex hull: *A set is defined as convex, if for any pair of points, the straight line connecting them is always inside the set. In particular if the connecting line is strictly inside the set, means that no point of the line (except the start and the end) touches the control polygon's contour, a strictly convex set is present.*

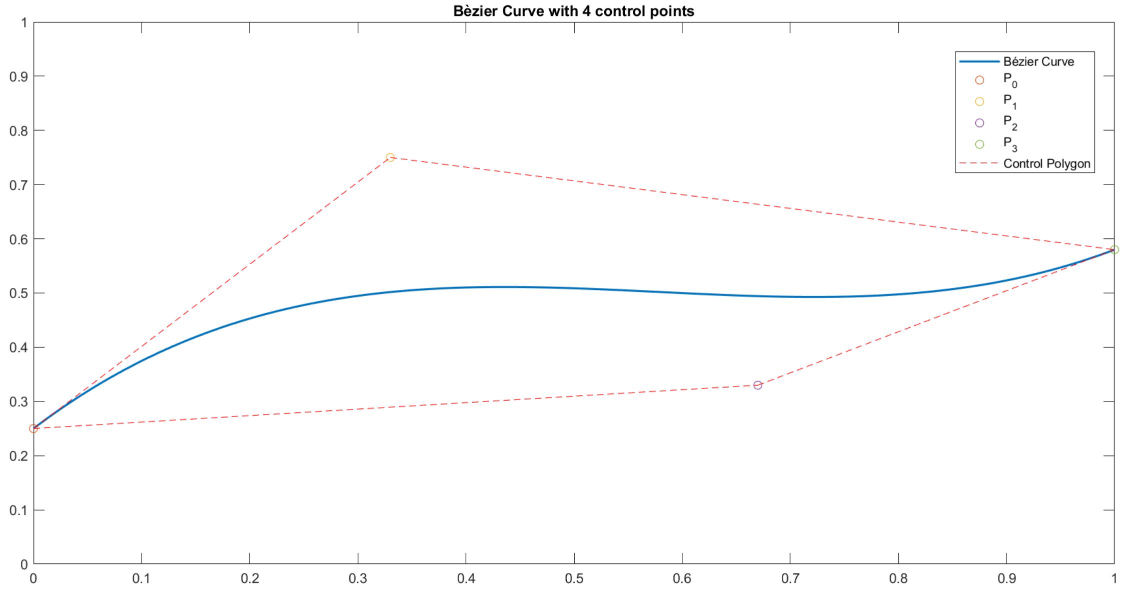


Figure 4.5: 3^{th} degree Bézier curve with four control points

Control points, introduced with bézier curves, gives the possibility of modifying a curve shape by moving their position. This method is computationally easier and simpler, both for visualizing and future changes. Despite moving one point will modify all the curve and this is one of the mainly drawbacks. NURBS curves will change only locally and so all the process is more controllable.

4.2.3 B-Spline Basis Functions

In sections 4.2.1 and 4.2.2, polynomial curves (Bernstein polynomial) and parametrized polynomial curves (Bézier curve) are introduced. They offer several benefits, in particual a simpler concept of curves and a more reliable way, despite these two methods has some drawbacks:

- The two previous representation don't show a local variation when changing the β coefficients or the control points, but only a global one. Often this is an undesirable effect.
- To pass through n points, a n^{th} degree curve both Berstein and Bézier is necessary. When the number of points become huge, a high degree is needed and obviously this directly affects the computational complexity.
- Often a high degree is needed to approximate real or needed shape.

B-Spline method, as cited in the section 4.1.2, is useful to lower the degree by considering a lower degree polynomial in a specific limited interval and then sum

up to obtain the entire curve.
A cubic polynomial with three segments is shown below in figure 4.6. A spline

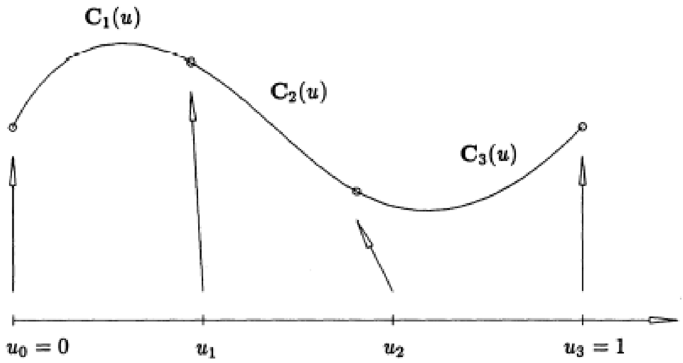


Figure 4.6: A piecewise cubic polynomial curve with three segments [28].

can be defined as: *a composite curve created by joining with continuity polynomial curves.*
In general, to join different curves a C^2 continuity is applied; last point of the first curve must coincide with the fist point of the second curve and so on till the last, in addition same tangent (first derivative) and same curvature (second derivative) are imposed.

B-Spline can be divided in three types:

- **Uniform B-Spline:** parametrized on a [0,1] interval.
- **Non Uniform B-Spline:** parametrized on different intervals.
- **Non Uniform Rational B-Spline (NURBS):** defined as a ratio between polynomial curves.

Let's analyze what *uniform* and *rational* means in B-spline.

Rational or non-rational is associated with the weights of the control points. When a curve's control points have all the same weight, usually 1, the curve is non-rational. Otherwise, the curve is rational.
Uniform is related with to the knot vector, this vector represents the points where the different segments join. It is uniform when is equally spaced between its elements, while non-uniform in the other case.
Knot vector [0,0,0,1,2,3,4,4,4] is uniform since the vector is equally spaced, while [0,0,0,1,2,4,7,7,7] is non-uniform. NURBS can be eighter uniform or not.

General definition of a B-Spline is:

$$S(t) = \sum_{i=0}^{n} P_i N_{i,k}(t) \tag{4.11}$$

where:

- $P_i : i = 0, \dots, n$ are the **control points** that control the shape of the curve.
- $N_{i,k}(t)$ are the **normalized B-Spline basis function**, described by a degree k, and by a non-decreasing sequence of real number, called knot vector, where each element its denoted as knot and represents the junction points of the different segments that generates the spline.
- k represents the **degree** of the Basis Spline $N_{i,k}$, while the **order** of the polynomial is $k + 1$ $(order = degree + 1)$.

B-Spline curves are constructed by joining different Bézier curve with a slightly modification, the basis spline $N_{i,k}$ are not the same, in particular are not Bernstein basis but other kind of basis calculated with recursive formulas (4.12) and (4.13):

for $k = 0$,

$$N_{i,0}(u) = \begin{cases} 1 & \text{if u} \in [u_i, u_{i+1}] \\ 0 & \text{otherwise} \end{cases} \tag{4.12}$$

for $k > 0$,

$$N_{i,k}(u) = \frac{u - u_i}{u_{i+k} - u_i} N_{i,k-1}(u) + \frac{u_{i+k+1} - u}{u_{i+k+1} - u_{i+1}} N_{i+1,k-1}(u) \tag{4.13}$$

B-Spline basis are defined on a vector called **knot vector**, that will be indicated with the letter U, which contains a sequence of non-decreasing real numbers called **knots** and indicated with u_i.

Fundamental constraint
The degree k, number of knots $m + 1$, and the number of control points $n + 1$ are related by the following equation:

$$m = n + k + 1 \tag{4.14}$$

Equation 4.14 is an important relation and it must always be satisfied for existence of the B-Spline curve.

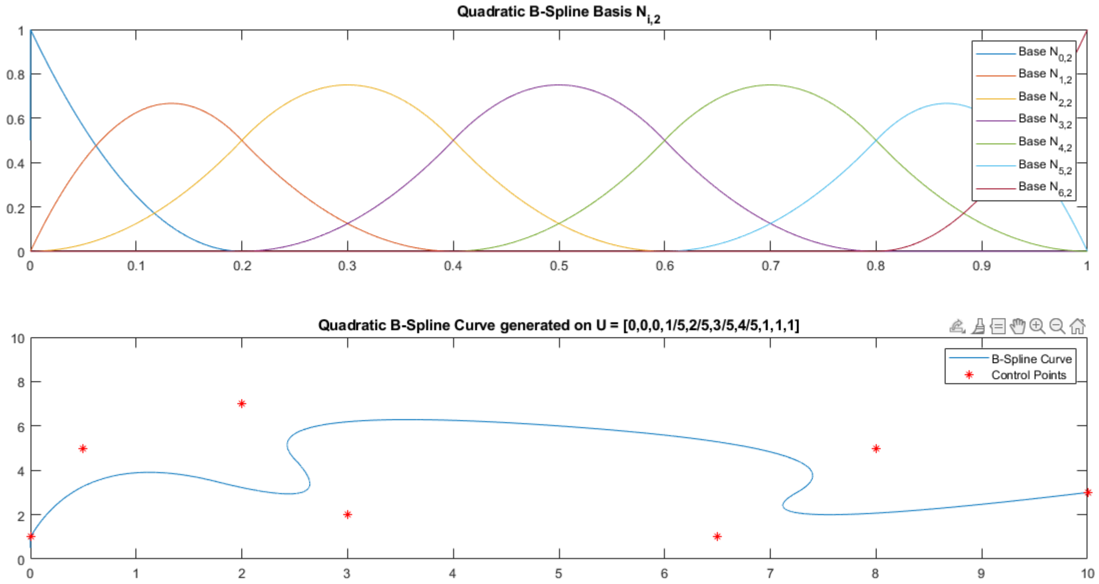
Example
Consider a quadratic B-Spline defined on the knot vector:

$$U = [0,0,0,\frac{1}{5},\frac{2}{5},\frac{3}{5},\frac{4}{5},1,1,1]$$

and the following seven control points:

$$P_0 = [0,1] \quad P_1 = [0.5,5] \quad P_2 = [3,2] \quad P_3 = [2,7]$$
$$P_4 = [8,5] \quad P_5 = [6.5,1] \quad P_6 = [10,3]$$

First of all, calculation of B-Spline basis $N_{i,k}$ for $k = 0,1,2$ is done. After, the formula (4.11) is applied to create the B-Spline curve using the above control points.
Results can be seen in figure 4.7, generated by a MATLAB script.


Quadratic B-Spline Basis $N_{i,2}$

Quadratic B-Spline Curve generated on U = [0,0,0,1/5,2/5,3/5,4/5,1,1,1]

Figure 4.7: B-Spline basis and curve

4.3 NURBS theory

NURBS states for **Non Uniform Rational Basis Splines**, is a mathematical model using basis splines (B-splines) for representing curves and surfaces.

NURBS are based on B-Spline theory, in fact they are parametric curves that use B-Spline as basis. They are the most generic curves possible, and for this reason are the most flexible.

4.3.1 The importance of NURBS curves

The **advantages** of NURBS curves are many:

- They took all the advantages present on Bèzier parametrization and B-Spline curve, such as lower degree, control points, basis parametrization on a vector (knot vector) and normalized B-Spline basis.
- It can interpret irregular surfaces, and very accurately depict most of the geometry forms to achieve smooth flow effects of curves and surfaces. NURBS gives to the designer a concept of freeform curves and surfaces.
- The NURBS curve has the advantage of inserting the new control point into the curve segment that only affects the insertion area, and has no effect on other line segments.
- Smooth trajectories are archive, in particular for autonomous navigation, like planetary exploration.
- Less memory is used to store the curve, in particular respect to mesh approximation, since only its control points need to be stored.
- Grater accuracy is achieved using the same degree respect to other kind of curves.
- Different trajectories are obtained respect to A^* path planning algorithm. Smooth trajectories and lower degree curves are archieved.

NURBS curves and surfaces are used in many fields, from geometry representation in mechanical design (CAD), to engineering such as robotics and self-driving cars. In particular areas that deal with **geometry representation**, both curves and surfaces, that requires **trajectory generation and smoothing**.

Example of applications
In literature, many applications to real world cases are present, such as industrial

design, where CAD software based on NURBS technology is a primal and fundamental tool for industrial designers [25].
With years, NURBS technology was applied in different area, such as mathematical optimization, where finding a way of modifying curves and surfaces is extremely important in terms of aerodynamics or energy consumption [26]. In company where smooth trajectories or trajectories optimization are required, NURBS play an important role in terms of costs, time, and resources. A relevant example is spying industry, where autonomous robot must track a trajectory for spraying the colour in an optimized way, optimization of colour quantity, cost, and time are keys of such kind of applications [27].

Now, in this thesis, NURBS curves are applied to design and draw new trajectories in the context of **autonomous planetary exploration rover**. These curves give the possibility to achieve smooth trajectories with lower degree curves.

4.3.2 Definition and Properties of NURBS curves

NURBS curves are defined as a ratio between two polynomial curves:

$$r(u) = \frac{\sum_{i=0}^{n} w_i P_i N_{i,k}(u)}{\sum_{i=0}^{n} w_i N_{i,k}(u)} \tag{4.15}$$

where:

- $P_i : i = 0, \ldots, n$ are the **control points** that control the shape of the curve.
- w_i represents the weight associated to a control point P_i, this recall the *Rational* in the NURBS definition.
- $N_{i,k}(u)$ are the **normalized B-Spline basis function**, described by a degree k, and by a non-decreasing sequence of real number, called knot vector, where each element is denoted as knot and represents the junction points of the different segments that generates the Spline.
  These basis are the same of the B-Spline:

  for $k = 0$,

  $$N_{i,0}(u) = \begin{cases} 1 & \text{if u} \in [u_i, u_{i+1}] \\ 0 & \text{otherwise} \end{cases} \tag{4.16}$$

  for $k > 0$,

  $$N_{i,k}(u) = \frac{u - u_i}{u_{i+k} - u_i} N_{i,k-1}(u) + \frac{u_{i+k+1} - u}{u_{i+k+1} - u_{i+1}} N_{i+1,k-1}(u) \tag{4.17}$$

- k represents the **degree** of the Basis Spline $N_{i,k}$ and of the NURBS curve, while the **order** of the polynomial is $k+1$ ($order = degree + 1$).

Formula 4.15 is defined over a knot vector U:

$$U = [a, \dots, a, u_{k+1}, \dots, u_{m-p-1}, b, \dots, b] \qquad a \le u_i \le b$$

with usually $a = 0$ and $b = 1$. As in the B-Spline, the degree k, number of knots $m+1$, and number of control point $n+1$ are related by the following formula:

$$m = n + k + 1 \tag{4.18}$$

This constraint must be satisfied to design the NURBS curve.

A more compact notation can be used, incorporating the weights and the denominator in a factor called **rational basis function** $R_{i,k}(u)$, and leave only the dependency on the control points:

$$r(u) = \sum_{i=0}^{n} R_{i,k}(u) P_i \qquad \text{where: } R_{i,k}(u) = \frac{w_i N_{i,k}(u)}{\sum_{i=0}^{n} w_i N_{i,k}(u)} \tag{4.19}$$

Properties of the knot vector U
This vector defines shape and behaviour that the basis $N_{i,k}(t)$ need to possess in their own interval. It is a sort of weight for the basis, and it is very complex to manipulate since a little change in its values can produce a totally different curve. The simplest way to manipulate this vector, since its size and values depends on the formula (4.18), and must be a non-decreasing vector, is to impose a new integer value in an increasing way.

Another important property is the so called **multiplicity** of a knot value in the knot vector. The multiplicity is the number of times a knot repeats itself consecutively in the knot vector. Depending on it, the properties of the curve changes, especially the curve's class F^r that define the continuity of its derivatives. Another property that NURBS curve must have in the context of trajectories generation is that the initial point and the end point of the curve must correspond to the start and goal point of the robot's path. These constraints are imposed by the multiplicity of the first and last value of the knot vector, that has be equal to the order of the generated NURBS.
Having a NURBS's degree of thre, means order four, the knot vector U:

$$U = [0,0,0,0, x_1, x_2, ..., x_n, 1,1,1,1]$$

must have this shape, the zero's and one's are repeated exactly four times, as the order.

It is intuitive to see that B-Spline and NURBS are more efficient and complete. It can be easily manipulated to adapt the curve by changing the control points position (B-Spline) and their relative weights (NURBS). Since NURBS has got the advantage of changing the curve's shape without changing the control points position it is clearly a more reliable and efficient method, which gives one more degree of freedom in manipulating the curve.

Example and comparison between B-Spline and NURBS

I developed a MATLAB code that shows the advantages of using NURBS despite of B-Spline.
Both curves use the following knot vector U:

$$U = [0,0,0,0,\frac{1}{4},\frac{2}{4},\frac{3}{4},1,1,1,1]$$

and the same control points position. Moreover, NURBS has one parameter more that represents the weights of the control points, $W = [1,1,0.5,1,1,1,1]$.
A degree of three is chosen and the NURBS generated will have order four, in fact the multiplicity of the first and last knot is four.

Image 4.8 shows the NURBS/B-Spline Basis, while figure 4.9 the two curves. Since in the NURBS basis the factor $\sum_{i=0}^{n} w_i N_{i,3}$ is present, the i-th weight will affect the i-th $N_{i,3}$ base in its shape. In particular, since the weight vector has got only ones with only 0.5 in the third position, all the bases will be the same except for the third one.

A similar discussion can be made on the two curves, figure 4.9, B-Spline curve represented in red has got all the control points weights equal to 1, while the blue one, the NURBS curve, since has got 0.5 on the 3^{rd} position of W, will be less affected by the 3^{rd} control point, the one in position (3,2).

This slightly modification can be useful in case around of the third point (position (3,2)) there is a massive object or valley or a particular soft terrain, in general every situation that can put the rover into a dangerous situation. To avoid the region, the weight associated with that specific point can be changed without changing its position.
The idea is to put the control points exactly on the obstacles or in specific regions around them. The control points as can be seen in figure 4.9 has two effects, the first is to attract the curve to its neighbourhood, and the second is to have a

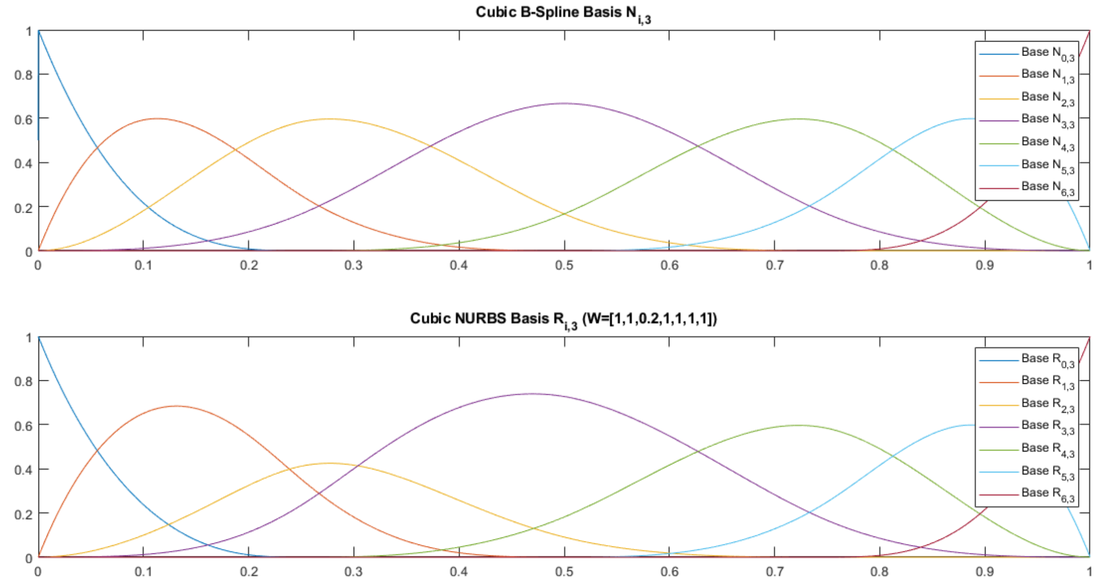


Figure 4.8: Comparison between cubic NURBS and cubic B-Spline Basis

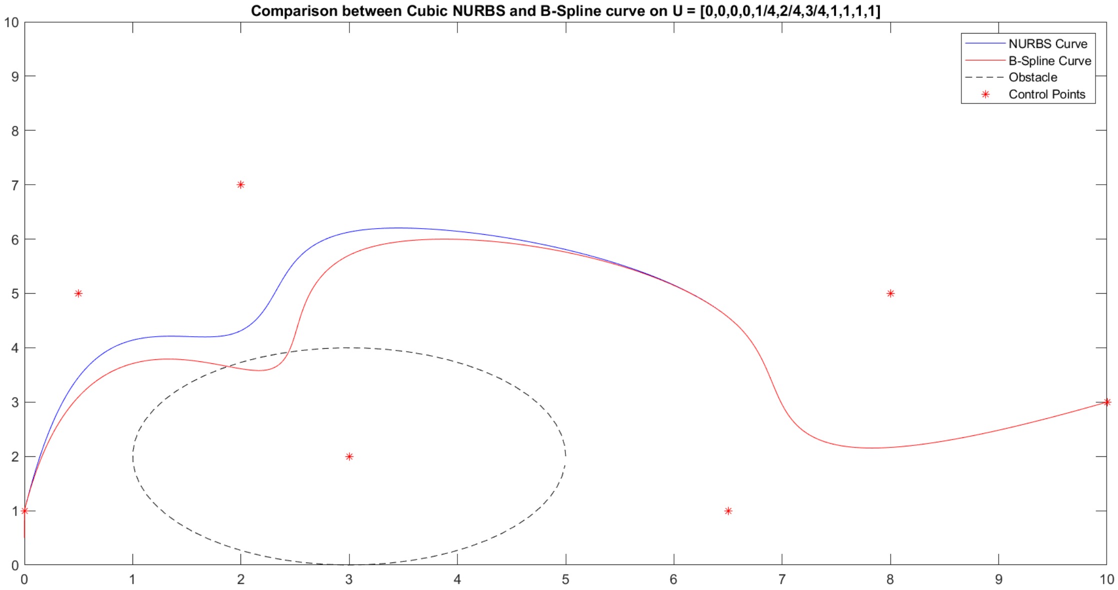


Figure 4.9: Comparison between cubic NURBS and cubic B-Spline curve

repulsive effect on the curve. By playing with two kind of behaviour the curve can be adapted for path planning purpose, avoiding obstacles.

Chapter 5

Algorithms and Methods

This charter will discuss about methods and algorithms developed for implementing a completely new path planner for planetary exploration rover, that can be eighter applied in normal robot navigation.

The implementation of the new path planner method is based on three subsystems:

1. Obstacle detection algorithm.
2. Graphic Viewer (based on OpenGL).
3. Path generation algorithm based on NURBS mathematics.

5.1 Obstacle detection

The implementation of the algorithm is based on standard **computer vision**, in particular using the **OpenCV** library and is written in **C++**.

OpenCV is an image processing library. It contains a large collection of image processing methods. Typically these functions are used to manipulate images with several types of transformations.

The algorithm takes as input a top-view image of the surrounded environment around the rover and provides as output a txt file, which contains for each row the information of a single obstacle. In particular, three parameters that describe the position of an obstacle: x-coordinate and y-coordinate of the obstacle's center and its radius.

5.1.1 Implementation

The working principle is very simple, the images is read, saved and manipulated as a matrix, encoded inside a **Mat object** variable.

Mat object is basically a class with two data parts: the matrix header (containing information such as the size of the matrix, the method used for storing, at which address is the matrix stored, and so on) and a pointer to the matrix containing the pixel values (taking any dimensionality depending on the method chosen for storing). The matrix header size is constant, however the size of the matrix itself may vary from image to image and usually is larger by orders of magnitude.

Each cell of the matrix represents a specific pixel of the image.
A flowchart of the implementation method is showed below in figure 5.1, and a detailed analysis is done in the following pages.

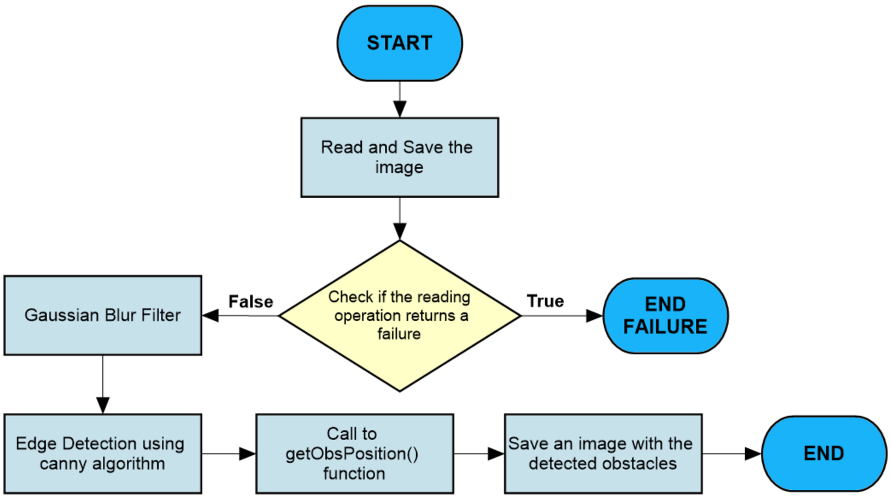

START
Read and Save the image
Check if the reading operation returns a failure
False
True
END FAILURE
Gaussian Blur Filter
Edge Detection using canny algorithm
Call to getObsPosition() function
Save an image with the detected obstacles
END

Figure 5.1: Flowchart of the Obstacle Detection Algorithm.

Procedural passages

The algorithm is developed in mainly **five steps**:

1. **Read and Save the image:** First of all, the image is read and save directly in a Mat object. If the operation doesn't return an error, the algorithm proceed further, otherwise it will end with a failure error.

2. **Gaussian Blur Filter:** Now a Mat variable will represent our image in a matrix format, where each position of the matrix corresponds to a pixel of the image, and takes a value between0 0 to 255. A Gaussian Blur filter is applied

to the image, means that depending on the specified parameter's filter, it will filter out the high frequencies in the image, and distortions will be reduced. The Gaussian Filter works like a low pass filter, deleting the high frequencies depending on its cut frequency. Gaussian Blur will return a grey-scale image.

In figure 5.2, the gaussian blur is applied to an image of mars surface, *Hardley crater* taken from ESA.
The produced Blur image has less noise since the high component are reduced. This kind of operation is based on gaussian function distribution:

$$G(x, y) = \frac{1}{\sqrt{\pi\sigma^2}} \exp^{-\frac{x^2+y^2}{2\sigma2}} \tag{5.1}$$

where x is the distance from the origin in the horizontal axis, y is the distance from the origin in the vertical axis, and σ is the standard deviation of the Gaussian distribution. Having a grey-scale pixel values with a standard deviation of σ_i, after a blur filter with a standard deviation of σ_b, the reduced standard deviation σ_f will be:

$$\sigma_f = \frac{\sigma_i}{\sigma_b 2\sqrt{\pi}} \tag{5.2}$$

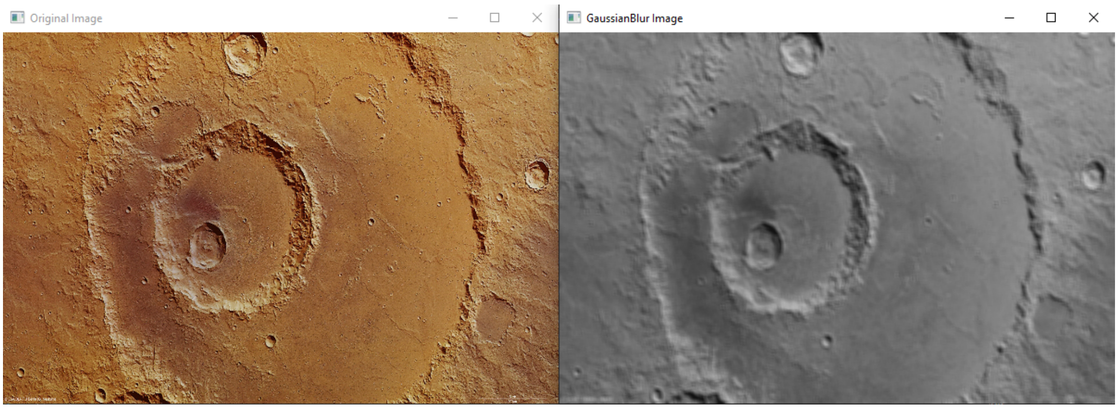


Figure 5.2: Gaussian Blur filter on the **Hardley crater**, martian's surface.

3. **Edge Detection:** This step will detect the edges in the image, **Canny** function of OpenCV is used to detect this kind of features. *Canny Edge Detection* is an edge detection operator that uses a multi-stage algorithm to detect a wide range of edges in images. It detects the pixels at which the image brightness changes sharply, or discontinuities are present. Depending on the parameters passed to the canny function, different edge can be detected,

in particular edge length and intensity detection can be tuned to only detect the needed edges.

The canny algorithm works in four steps:

(a) Find the intensity gradients in the image, means discontinuities and brightness changes.

(b) Apply a gradient magnitude thresholding to get rid of spurious response to edge detection.

(c) Apply double threshold to determine potential edges.

(d) Track edge by hysteresis, suppressing all the other edges that are weak and not connected to strong edges. This stage decides which are all edges are really edges and which are not. For this, we need two threshold values, minVal and maxVal. Any edges with intensity gradient more than maxVal are sure to be edges and those below minVal are sure to be non-edges, so discarded. Those who lie between these two thresholds are classified edges or non-edges based on their connectivity. If they are connected to "sure-edge" pixels, they are considered to be part of edges [Canny documentation]. Otherwise, they are also discarded, figure 5.3.

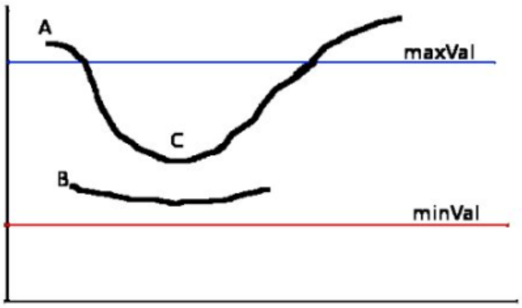


Figure 5.3: Hysteresis threshold in canny detection.

The edge A is above the maxVal, so considered as "sure-edge". Although edge C is below maxVal, it is connected to edge A, so that also considered as valid edge and we get that full curve. But edge B, although it is above minVal and is in same region as that of edge C, it is not connected to any "sure-edge", so that is discarded. So it is very important that we have to select minVal and maxVal accordingly to get the correct result [Canny documentation].

Example of application applied to gaussian blur image obtained in the previous step is show in figure 5.4.

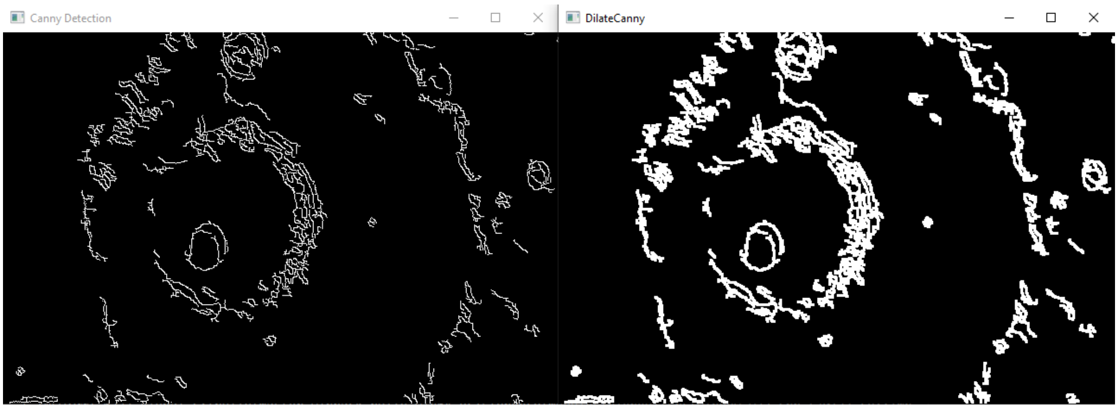


Figure 5.4: Canny Detection and Canny dilate for robustness.

On the left of figure 5.4, a binary image is obtained by applying the canny algorithm in which are highlighted the edges detected. On the right the same image is represented, but the edges are wider for robustness purpose, to avoid the situation of a non-detection of small edges by the rest of the algorithm.

4. **getObstaclePosition function:** This routine, schematize in the flowchart 5.5, will save the coordinates of the obstacle's centers with their own radius on a txt file. The canny image obtained at the previous step (Binary image) is the input of the function to extract all the closed contours of obstacles. Only parental contours are found, means that if an obstacle is contained within another detected contour it will be ignored since is a fictitious case, obstacle in an obstacle. After the extraction of all the contours, a for loop is perform on each contour and a **convex hull** approximation is made, figure 5.6. Having a convex hull is more simple to perform the center and the maximum distance from it thought basis mathematical formula. After computing the x-y coordinate of the center and its radius, a save is performed into the file.

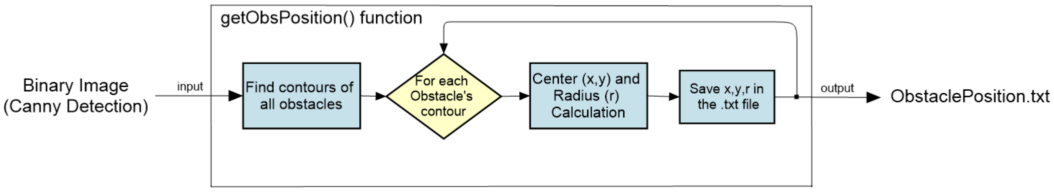

getObsPosition() function
Binary Image
(Canny Detection)
input
Find contours of all obstacles
For each Obstacle's contour
Center (x,y) and Radius (r) Calculation
Save x,y,r in the .txt file
output
ObstaclePosition.txt

Figure 5.5: Flowchart of getObstaclePosition function.

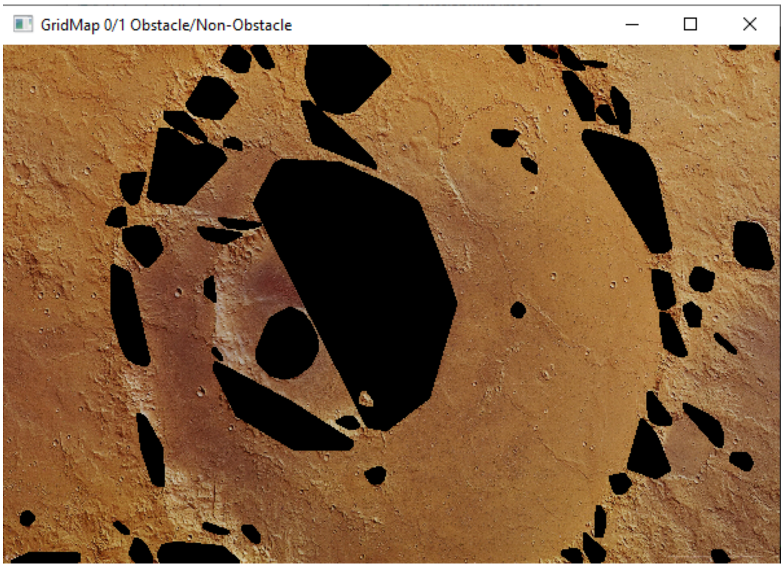
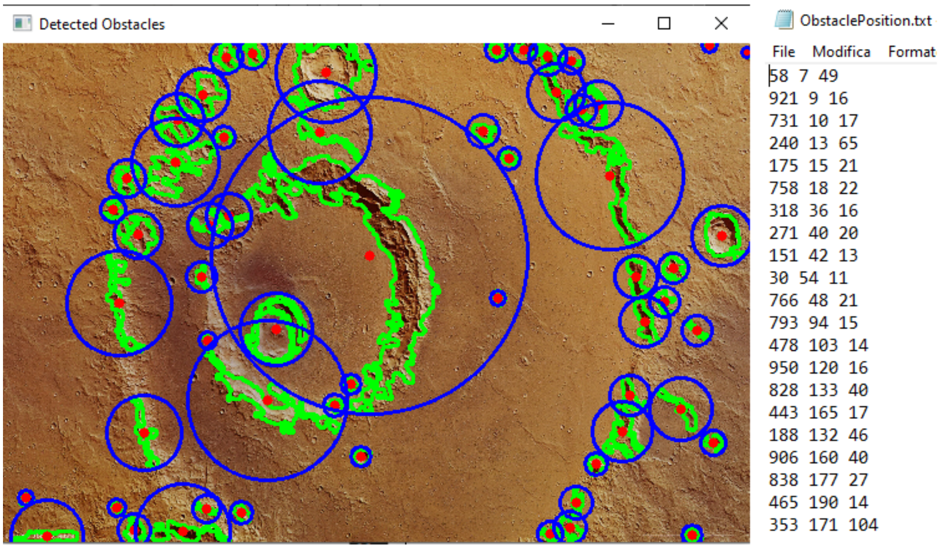


Figure 5.6: Convex hull of the detected obstacle, in black.

5. **Save an image of the obstacles:** Here simply the detected obstacles are printed on the original image, figure 5.7. In green are highlighted the contours of the obstacles, then a red star represents the centers and a blue circle the circumference that inscribes the obstacle. Next to the image can be seen in what format is the txt file written. In each row of the file we will have three parameters for each obstacle; x,y and r. The radius values are rounded up with the ceil function to have an integer value.

This is mainly how the implementation to detect the obstacles works. Obviously, the algorithm takes care of the image's dimension, and convert the position of pixels in the correct format used by the path planner algorithm. The input size image can be any, the returned position is expressed in a specific size format, 1000x600 used from the path planner. The algorithm internally adapts any size of the input image to a predefined size, 1000x600, in which the obstacles are detected.



```
58 7 49
921 9 16
731 10 17
240 13 65
175 15 21
758 18 22
318 36 16
271 40 20
151 42 13
30 54 11
766 48 21
793 94 15
478 103 14
950 120 16
828 133 40
443 165 17
188 132 46
906 160 40
838 177 27
465 190 14
353 171 104
```

Figure 5.7: Detected obstacle with next to it the txt file.

5.2 Graphic Viewer based on OpenGL

OpenGL is an *application programming interface (API)* that allows the user to draw and create 2D and 3D graphics. It is used as an external library that can be imported in the programming environment.
OpenGL has several functions to visualize, render, and manipulate figures and points.
The library provides a software interface between a programming language, such as C++ in this case, and the hardware. OpenGL works pretty well for the representation of curves and points since it works on manipulating single points, in fact by joining points it is possible to create complex shapes of any kind.

5.2.1 Introduction to OpenGL

Let's look to the main function used to visualize the paths, obstacles, and rover's position.
For simplicity let's focus on 2-D representation, since is the main used in this thesis.

There are three mainly library developed for OpenGL programs, which are used in this thesis:

- **OpenGL basic library**: It is a collection of functions useful for specifying

graphics primitives, attributes, geometric transformations, viewing transformations, and many other operations.
Are prefixed with *gl* letters followed by the function name, with capital first letter:

glClear glMatrixMode glColor3f glVertex

The above functions require one or more parameter as symbolic constant specifying, for instance, a parameter name, a value, or a mode. All such constants begin with the uppercase letters GL, such as:

GL_2D GL_RGB GL_PROJECTION GL_POINTS

The OpenGL functions also expect specific data types. For simplicity, OpenGL defines data types that have the same size on every machine, totally different despite integer, double and float in classical definition on programs:

GLboolean GLint GLfloat GLdouble

- **OpenGL Utility (GLU)**: Provides routines for setting up views and projection matrices, describing complex objects with line and polygon approximations, surface-rendering operations or other complex tasks. All GLU functions name start with the prefix *glu*, like *gluOrtho2D()*, used to set width and height of the projected 2D environment.
- **OpenGL Utility Toolkit (GLUT)**: It provides a library of functions for interacting with any screen-windowing system. The GLUT library functions are prefixed with *glut*, and contains also methods for describing and rendering quadric curves and surfaces.
  Since GLUT is an interface to other device-specific window systems, the programs will be device-independent.
  Some examples are:

  glutInit glutCreateWindow glutDisplayFunc glutMainLoop

5.2.2 Integration with the path planner

The path planner algorithm and the graphic render are executed within the same thread, having in common the same main function.
They are executed together for a graphical reason, since without this operation it is

impossible to visualize the path and the obstacles. It is important to remark that the path planner algorithm is completely independent and works without OpenGL.

First, the **main function** is described in detail, figure 5.8.
The configuration is very simple, details of each block is presented:

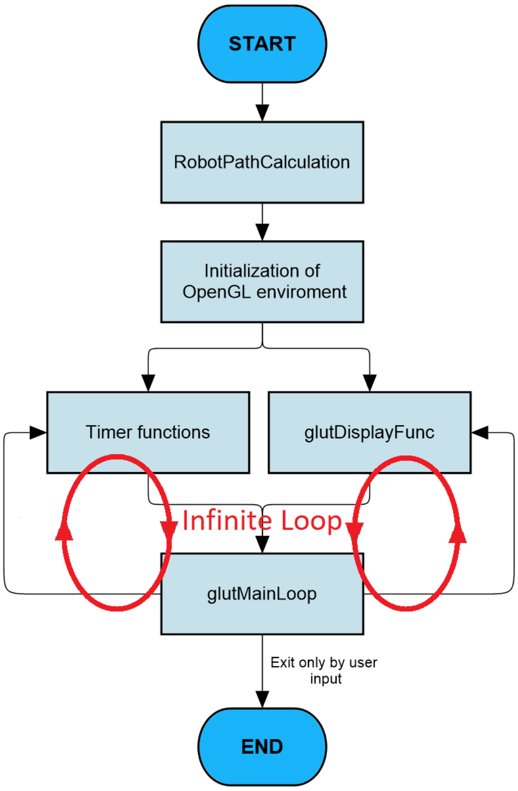


Figure 5.8: Main function of the program.

1. **Path generation**: The algorithm developed in this thesis is executed, and it will compute the path from the start to the goal using NURBS curves, which the robot need to follow.

2. **OpenGL initialization**: First of all, an initialization on the parameters used to display an OpenGL windows on the monitor screen is done. In this specific case, the following function was called, using the specified order:

   - **glutInit** function will initialize the GLUT library and negotiate a session with the window system. Typically, is the first function we call when rendering with OpenGL.

- **glutInitWindowSize(x,y)** function is used to tell OpenGL what size the window should have. The two input parameters, x and y, represent the width and the height in pixels of the window.
- **glutInitWindowPosition(x,y)** is used to tell OpenGL the location of the displayed windows. x and y express the top-left corner position in pixel.
- **glutCreateWindows** is the command that tells that a window need to be created on the screen with given caption for the title bar. All the previous commands can be seen in the image 5.9.

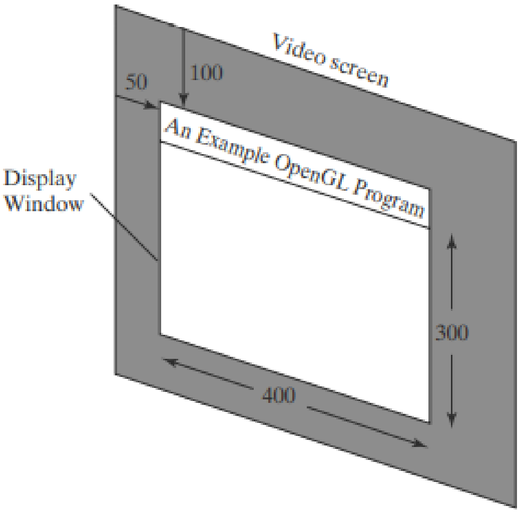


Figure 5.9: Window displayed from OpenGL [29].

- **init2d()** is a custom function that collects methods, which will set other options for the display window, such as buffering, choice of colour modes, projection type and background window colour.
  A **RBG** (red, green, blue) colour format is used to express which colour objects in images need to have. Usually every colour is expressed by an integer value between 0, minimum intensity, to 255, maximum one.

  A **double buffer** render images technique is used to refresh window pixels, where obstacle, path, and rover position will be displayed. This solution achieve good performance in term of FPS (frame per second) and a continuous and smoother video is rendered, since a successive flow of frames. *Double buffer* works as storage image.
  To achieve high FPS and so a video without perceiving the refresh of the

pixels, a refresh rate frequency higher that the human eyes is mandatory. For this reason, when an image is displayed on the screen, the program simultaneously evaluates the next frame and saves it in a buffer memory A and when the next frame needs to be displayed in the screen, OpenGL will take the image previously computed from the buffer A and in parallel will be evaluated and saved the next image, putting it in the buffer memory B. This swap of buffer between A and B will increase in a dramatic way the video frame rate, otherwise without this technique the displayed video will be slow, and a lag effect will be visible to the user.

Buffer swapping is done by the command:

glutSwapBuffers()

Double buffer technique is schematize in figure 5.10.

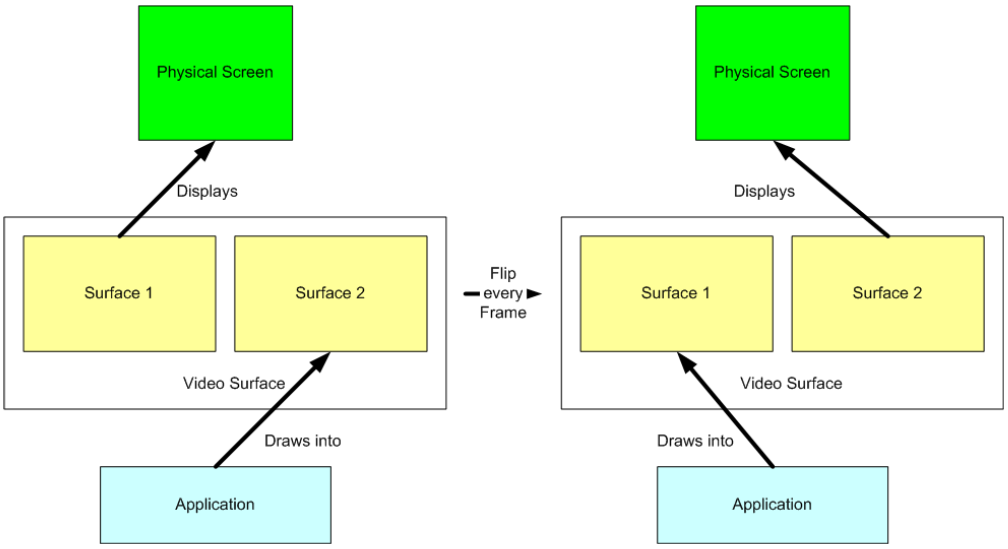


Figure 5.10: Double buffer mechanism for higher FPS.

Other two important settings are done in the init2d function:
Thought the commands:

glMatrixMode(GL_PROJECTION)
gluOrtho2D(0.0, width, 0.0, height)
glClearColor(0.0f, 0.0f, 0.0f, 0.0f) and
glClear(GL_COLOR_BUFFER_BIT)

is set in which way OpenGL should display the images on the screen. In OpenGL, 2D representation is particular case of 3D, in fact can be

seen as a projection on a plane, it will treat each image as a projection on a plane, where the plane is represented by the screen. The second command is used to indicate that the projection area has a particular size, 1000x600.
Thought the last command, the definition of the window background colour is set. The first three parameters are 0's and represent the black colour in a RBG format. This command will save the colour in the buffer bit and used on the displayed window.

3. **glutDisplayFunc**: It is an internal OpenGL function that must be used to display objects on the screen. Command:

   glutDisplayFunc(display)

   will accept as input a **callback function** (display callback). Function display will contain the description of the objects that have to be rendered, such as path.
   This particular function call can be seen redundant since it is a call to a function nested within another function. This strategy is used because glutDiplayFunc is a particular function that will be periodically called until the user decides to end the program.

4. **Timer functions**: A timer is a function that is called periodically till the end of the program every x seconds or milliseconds.
   Let's make an example:
   Command line:

   glutTimerFunc(x,timer,0)

   will create a timer that every x milliseconds calls the function timer. Every x milliseconds OpenGL will execute in parallel the function timer. This mechanism is very useful and is exploited to change global variables in the program, that are used from the display function, which actually display on the screen objects that will change using the timer periodicity.

5. **glutMainLoop**: This is the function that will put our program in an infinite loop till the user will stop it. Since the program enters in a loop mode, is clearly understandable that it has to be the last function called in the program. When glutMainLoop in executed, OpenGL will remain stack in a loop where only particular function will be called, such as timers and glutDiplayFunc functions. Each loop duration depends on the time that the program needs to completely executes the loop. Is now easy to understand that for dynamic

sequence of images, means basically a video, displayed images in high refresh frequency that are different frame to frame is necessary.
All codes regarding the dynamical change of the image must be put only in those function that glutMainLoop will call, this the reason why the path generation algorithm is executed as a function inside the display callback.

In the program there are three different timers; *timer*, *timer1*, and *timer2*.

Timer: The timer function will evaluate when the rover is in a waypoint by continuously checking, every 10 milliseconds, its position along the path. If the rover is in a waypoint the global flag *wayPointsFlag* is set true. The flowchart 5.11 describe clearly the process.

Timer1: The timer1, every 100 milliseconds, will change the displayed dimension of the rover to obtain a pulse effect of its position.

Timer2: The timer2 continuously checks, every 3 seconds, when the global flag *wayPointsFlag* is true. When positive evaluation is obtained the timer2 will call the path generation algorithm (pathGeneration function) and at the end will set the previous flag to false.
Flowchart represented in figure 5.12.

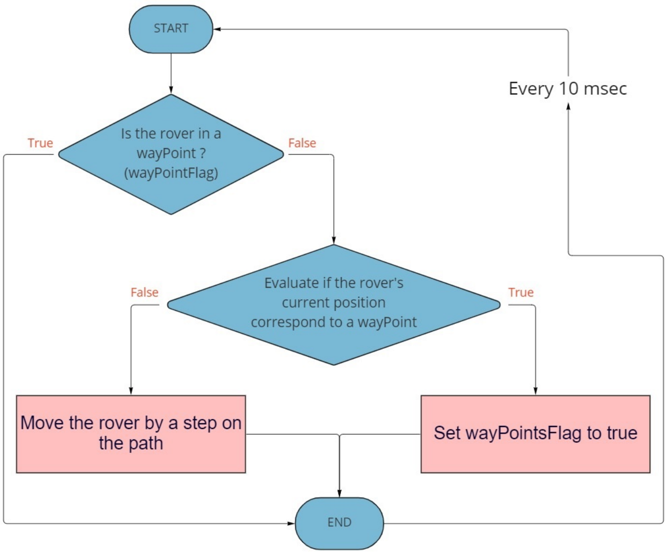

START
Every 10 msec
True
Is the rover in a wayPoint ? (wayPointFlag)
False
False
Evaluate if the rover's current position correspond to a wayPoint
True
Move the rover by a step on the path
Set wayPointsFlag to true
END

Figure 5.11: Timer/Timer0 function.

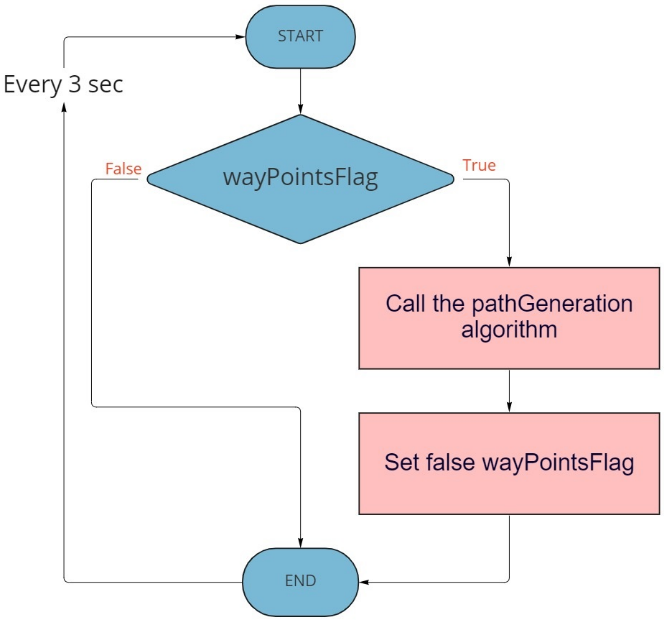


Figure 5.12: Timer2 function.

5.3 Path Generation

The core of the program in naturally the path planner algorithm, which is implemented as a function, *pathGeneration* function.

Due to OpenGL, the algorithm is called as a function by the timers at the start and in the waypoints.

The algorithm is written in C++ and is based on **SISL library**, which has been gradually developed and enhanced for more than three decades by the geometry group at SINTEF in Oslo and on a **SISL toolbox** developed by Antonino Bongiovanni.

5.3.1 Introduction on SISL library

SISL stands for ***S**intef **S**pline **L**ibrary* and is a geometric toolkit to model with curves and surfaces. It is a library of C functions to perform operations such as the definition, intersection, and evaluation of NURBS (Non-Uniform Rational B-spline) geometries. Since many applications use implicit geometric representation such as planes, cylinders, etc. SISL can also handle the interaction between such geometries and NURBS.
This library is written K&R C style for historic reasons and contains a thousand of function for creating, manipulating, and evaluating curves and surfaces. It is easily to understand that since thousands of functions are present is difficult to deal with and for this reason a manual is provided.

In the other hand, SISL toolbox library developed by Antonino Bongiovanni is built on top of the basic SISL library. It is written in C++ and makes the usage of basic SISL library easier and more understandable. Creating curves object with attributes and methods simplifies the management of the library and can be seen as a translation of the Basic SISL, from C to C++. It it an easy way to manage it with improved features.

In this thesis, both libraries were used. Primary the Antonino Bongiovanni's library is used since classes' structure in C++ are more reliable, understandable, and manageable. Basic SISL functions are used in particular cases when the toolbox does not provide it.

5.3.2 Assumption and Personal choice

A first assumption in the algorithm is to consider only **circular obstacles**. This choice is done because it is easier to deal with circle for three mainly reasons:

1. **From an obstacle detection point of view**. It is easier to inscribe any obstacle's shape in a circle, finding the barycenter and the maximum distance of the obstacle from the latest. Barycenter will become the center of the circle and the maximum distance the radius.
   This method is a brute approximation by excess, which is very conservative. In the other hand, the best possible solution is to inscribe the obstacle is a convex hull that will minimize the area and a better optimization of the space environment is obtained.

2. **From a fast SISL computation**. In fact, SISL routines provide a description of circles in SISL parametrization. This will end up in a very compact data structure easily manageable and controllable by SISL routines. This is not true for convex polygons, SISL routines does not provide any representation of them, and a own creation will produce a complicated data structure difficult to create and manage.

3. **From a path planner point of view**. It is easier to put control points around a circle respect to a convex polygon, since a constant radius is present.

At the beginning can be difficult to deal with NURBS, since the control points must be chosen in such a way to control the direction of the curve.

Important property: Remember from section 4.2.2, that a NURBS curve can be inscibed in a convex polygon defined by its control points. This is a fundamental concept that ensure the possibility of undestanding where the NURBS will go.

Figure 5.13 shows two examples of control polygons formed by the vertices; START, P2 and GOAL. These polygons are convex and the NURBS can be found inside it. In the first example, by controlling the **weights** of P1 and P2 it is possible to avoid the obstacle. In the other hand, the second example shows a different convex polygon; START, P3 and GOAL. Its easy to undestand that for any weights of P1 and P3 the NURBS cannot avoid the obstacle since it will be constrainted inside the convex polygon.

In this thesis, an external circle, around each obstacle, has been added. These **external circles** make possible to add control points, such as P2, that bring the NURBS curve outside the obstacle. The external radius are initialize at a value

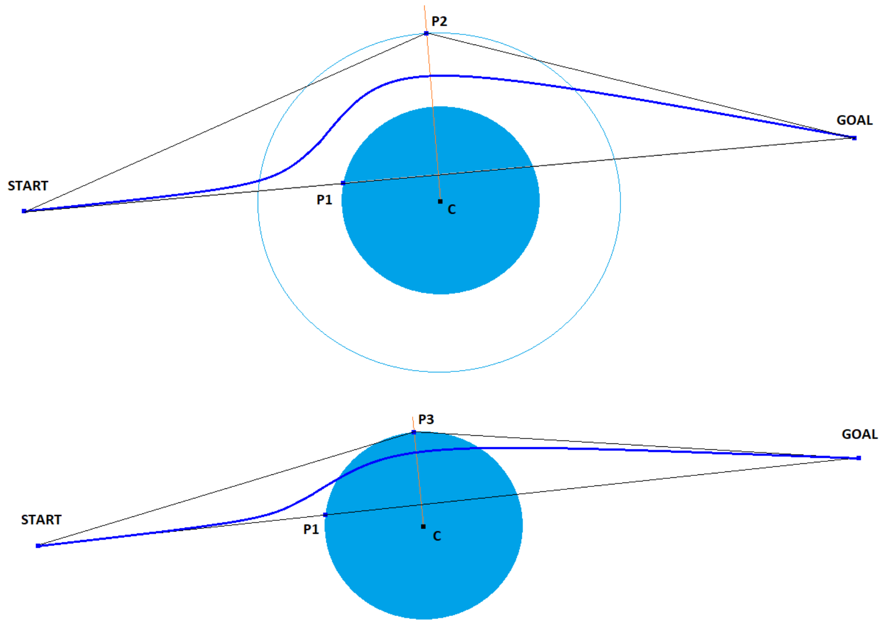


Figure 5.13: Convex polygon (START-P2-GOAL) and NURBS curve in blue.

that makes the NURBS to be smooth and possibly stay away from the obstacles. The position of P2 is not chosen at random but optimized in such a way to minimize the number of control points. It will be explained in details later.

The latter assumption considers a **divisible obstacles** in the environment. Means that each obstacle does not intersect any, overlap of obstacles are avoided.
This kind of distinction can be done in the obstacle detection algorithm. When two or more obstacles intersect each other a bigger circle radius can be inscribed inside these obstacles resulting to the path planner only a big one obstacle.

5.3.3 Robot dimension in the environment

To better optimize the trajectory, the **robot dimension** can be taken into account. This step can be done in two different ways:

1. **Radius enlargement**: This can be eighter considered in the obstacle detection algorithm or in the path planner algorithm.
   In the obstacle detection, when detecting the radius, means the maximum distance of the obstacle from its barycenter, a *delta* can be added to consider the robot dimension. No more a radius r but $r + delta$ is considered, where delta can be the robot width. By doing this passage the real radius would

be *r* while the path planner will see the bigger one. The trajectory will be evaluating taking into account a bigger obstacle and when the rover will pass close to an obstacle, the real distance will be bigger since the real obstacle is actually smaller.
In the path planner algorithm, the same passage can be done when reading the .txt file and adding the same *delta* to the original radius.

2. **Control point less sensitive**: By reducing the effect of the control point, the trajectory will be further away from the obstacle. This method is harder since not always reducing the weight the NURBS will stay farther from the obstacle.

An example is shown in figure 5.14, the red NURBS is affected by the position of the red point while the blue one by the blue.

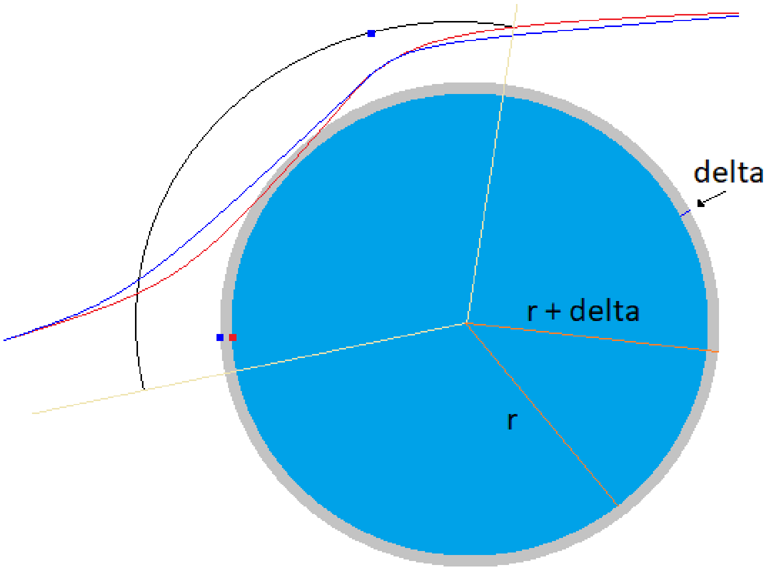


Figure 5.14: Effect of the robot dimension on the obstacle radius.

5.3.4 Path planner implementation

The main steps of the algorithm can be summarize in the following flowchart.

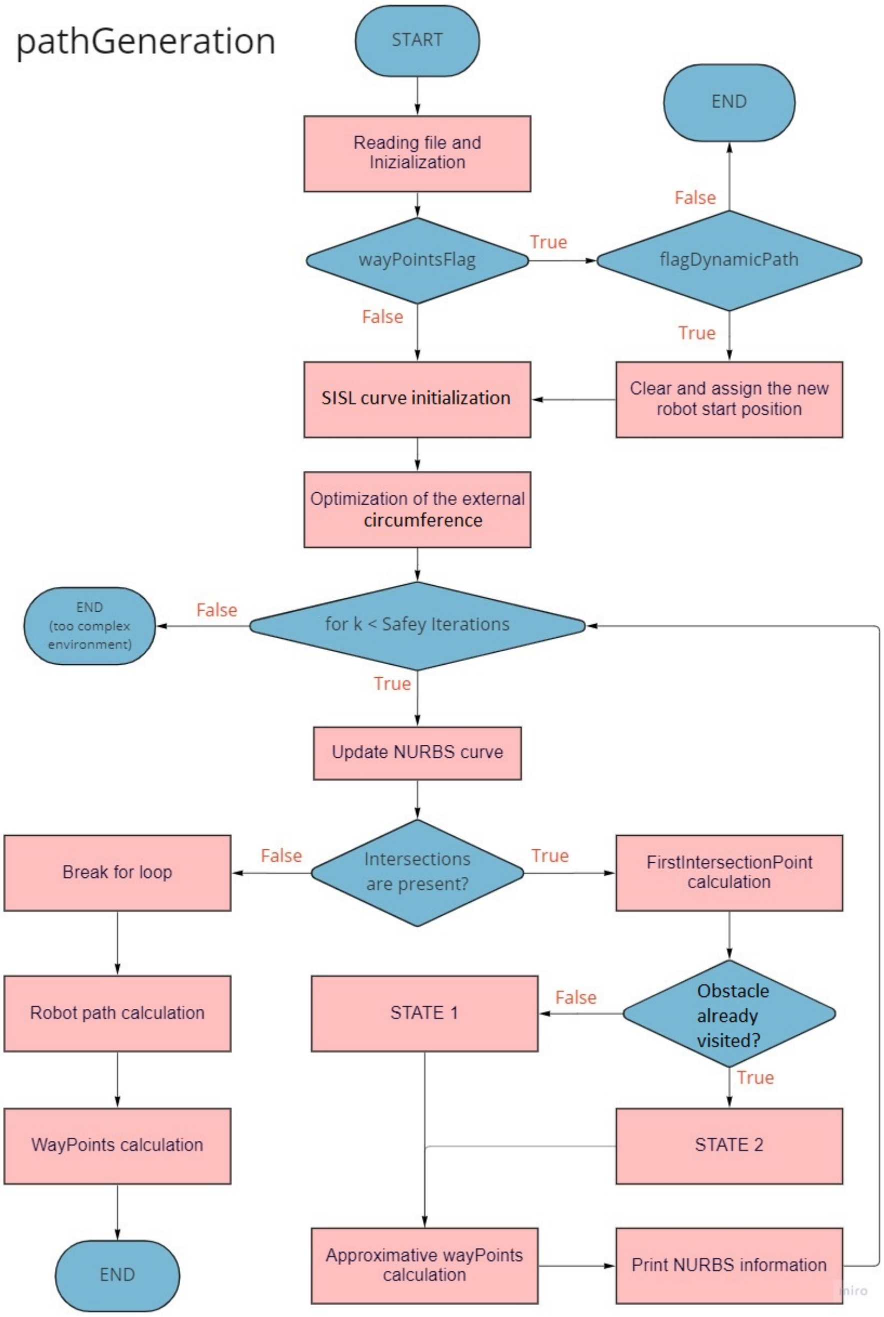

pathGeneration
START
Reading file and Inizialization
wayPointsFlag
True
False
flagDynamicPath
False
END
True
Clear and assign the new robot start position
SISL curve initialization
Optimization of the external circumference
for k < Safey Iterations
False
END (too complex environment)
True
Update NURBS curve
Intersections are present?
False
True
Break for loop
Robot path calculation
WayPoints calculation
END
FirstIntersectionPoint calculation
Obstacle already visited?
False
True
STATE 1
STATE 2
Approximative wayPoints calculation
Print NURBS information

Figure 5.15: Flowchart of the path planner algorithm.

In the following pages, a detail explanation of the implementation is present. The explanation will refer to the previous flowchart and will be organized in sub-sections to have a better organization respect to list of states.

Reading file and Initialization

As cited in section 5.1.1, the path generation algorithm takes as input a .txt file containing crucial information regarding position and radius of each obstacle. The output is a pointer to a NURBS object containing all the information describing the object, like degree, knot vector, weights, control points and another pointer to the SISL NURBS curve.

The *pathGeneration* algorithm is executed ones at the beginning and periodically called in the Waypoints along the trajectory.
First time, the obstacle centers and radius are read and saved in the following variable:

```
static vector<Eigen::Vector3d> ctrlPoints\_All;
```

ctrPoints_All will contain all the obstacles centers, in x-y-z coordinates. It is defined as a **local variable** since only this function has to access it, and as **static** since the function is called periodically and its past values need to be recorded, otherwise a non-static variable will be destroyed at the end of the function and the information will be lost. The variable type is a vector of *Eigen::Vector3d*, vector since dynamic allocation of memory is needed and *Vector3d* since each center is seen as a point in three coordinates, where x and y express the position of the obstacle while z is set to zero.
The respective radius are saved in:

```
static vector<int> obs\_radius{};
```

In the other hand, when pathGeneration is called in the Waypoints a comparison between the .txt file and the old (ctrlPoints_All) obstacles is done. If new obstacles are detected, the local flag **flagDynamicPath** is set to true otherwise it will remain false.

Flags evaluation

It this part, the algorithm will evaluate two flags; *wayPointsFlag* and *flagDynamicPath*. Remember that the first one, wayPointsFlag, is set true by a *timer0*

function when the rover position will match with a Waypoint and consequently the pathGeneration is called. At this point, if wayPointsFlag is true, the algorithm will evaluate an additional flag, *flagDynamicPath*, set in the previous subsection, depending if new obstacles are detected.
If *flagDynamicPath* is true, the algorithm will substitute the start point with the waypoint just found, and a path will be generated from the new start to the goal.

```
ctrlPoints_All.at(0) = wayPoints.at(0);
RobotPosition = 0;
```

Otherwise, if false, the algorithm terminates. The *pathGeneration* will return without generating any kind of path. This is done because the trajectory that would have been generated would be the same as before.

SISL curves initialization

At this point the algorithm is ready to instantiate in memory the SISL curves representing the obstacles and their relative external circumferences, and all the variables for generating the NURBS. Let's analyse the command and the variable used for generating a NURBS SISL curve:

1. **Dimension**: Means the dimension where the NURBS lays, in this case three and not two since some SISL function will only works in three dimensions.

```
const int dimension = 3;
```

2. **Knot Vector**: Having the first two control points, the knot vector can have a predefined size m, depending on the number of control points n and on the degree k. Remember the fundamental formula:

$$m = n + 1 + k$$

To find a knot vector that respect the previous formula, with the constraint $n = 2$, degree k must be imposed:

- With degree 1, means order 2: $k = 1 \rightarrow m = 2 + 1 + 1 = 4$.
Now the multiplicity of the knots of the vector need to be imposed, for let the NURBS pass in the start and goal points. Having order 2 and $m = 4$, and a non-decreasing vector will be: [0,0,1,1]. This vector satisfies all the properties.

- If for absurd a degree of 2 or higher is imposed: $k = 2 \rightarrow m = 2+1+2 = 5$. Five values are needed in the knot vector, but the multiplicity now becomes equal to the order, means 3, so a minimum of $3 + 3$ values are needed, in contrast with the dimension $m = 5$.
  Is not possible to find another knot vector having only two control points as constraint.

```
vector<double> knots{};
```

3. **Degree**: From the previous consideration the degree is 1.

```
int degree = 1;
```

4. **Weights**: The weights of the two control points are imposed equal to 1, and never will be changed, since they represent start a goal position.

```
vector<double> weights{};
weights.push_back(1.0);
weights.push_back(1.0);
```

5. **Degree**: From the previous consideration the degree is 1.

Next, the generation of the obstacles and the external circumference are generated. Data structure in SISL format are created, SISL objects representing the obstacle are created defining all the variable needed, such as center, radius, degree and dimension.

```
int degree = 1;
obs.push_back(make_shared<CircularArc>(2 * 3.1415926, axis,
    obs_points.at(i) + Eigen::Vector3d{(double)obs_radius.at(i), 0.0,
    0.0}, obs_points.at(i), dimensionObs, orderObs));
```

In the second line, *obs* is a vector that contains the pointers to each obstacle circle. The same operation is done for creating the external circumference with the difference that the radius is: obs_radius.at(i)+safety_distance, where safety_distance represents the gap between the two radius.

Optimization of the external circumference

This passage is done to have only two or three near obstacles, means that the external circumference of each obstacle intersects a maximum of one or two obstacles. An obstacle is considered near to another if the two external circumferences intersect. This is done to consider only small group of obstacles, since many near obstacle will become difficult and complex to manage. Remember that external circumference is added to optimize the path, bringing the curve away from the obstacles.
A group of maximum 3 near obstacle is the best for the algorithm.

Iterations

A for loop iteration is done until no more intersections are found. If intersections still present after a number of *safety iteration*, means that probably the environment is too complex to generate a path, and a more advanced concept of path generation is needed.

Update NURBS curve

Update NURBS means generation of the NURBS curve using the actual variable saved in memory. It will generate, in the first iteration, and update, in the next iterations, effectively the SISL NURBS curve in memory with the defined variable; degree, dimension, knot vector, control points and weights.

```
UpdateNurbs(degree,knots,weights);
genericCurve = make_shared<GenericCurve>(degree, knots, ctrlPoints,
    weights, coeff, dimension, degree + 1)
```

UpdateNURBS is a function that returns the updated knot vector given the degree and the control points. Contains a simple logic to create a non-decreasing vector of a predefined size.
The *genericCurve* variable is the NURBS object. The *coeff* variable is an empty vector that is used internally from the NURBS. At each cycle the variables describing the NURBS are updated based on the intersected obstacle and an update of the NURBS is done.

Intersection evaluation

The updated NURBS is used to evaluate if intersections with obstacles are presents. All the intersections are found and saved in *intersections_All* vector, in casual order.

```
intersections.clear();
intersections = genericCurve->Intersection(obs[j]);

if (!intersections.empty())
    position.push_back(j);

for (const auto &ele: intersections)
    intersections_All.push_back(ele);
```

In figure 5.16 are highlighted, in color yellow, an example of four points (A,B,C,D) intersected.

Vector *intersections_All* will contain all four points in casual order, i.e. C,D,B,A.

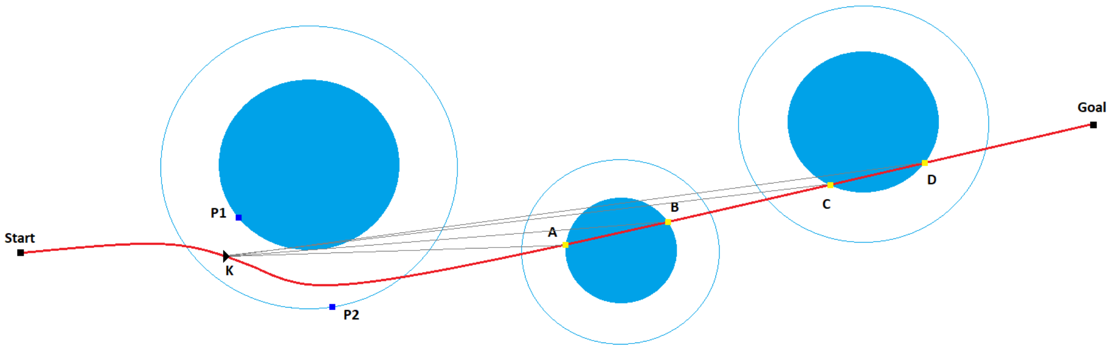


Figure 5.16: NURBS intersections in points: A,B,C,D.

If *intersections_All* is empty, means a path from start to goal is generated avoiding all the obstacles and no more intersections are detected, the for loop is brake and the Waypoints are calculated from the approximative Waypoints.

Since the NURBS at each iteration changes its shape by a little variation (figure 5.22), because of the addition of control points, can happen that the previous Waypoints calculated are no more in the path. The closest points in the path from the approximative Waypoints are taken as definitive Waypoints.

FirstIntersectionPoint calculation

If intersections are present, the algorithm will evaluate which among all is the first one, saving it in *FirstIntersectionPoint* variable, using the *firstIntersection* function.

Procedure of *firstIntersection* function: The function will take as input the *intersection_All* vector and will evaluated which among its points is the first that the NURBS intersects, based on a geometric decision.

Making explicit reference to figure 5.16, the function will compute four distances measures. Point K will move along the curve, thought its parametrization, and four distances are computed at each point K: $\overline{KA}, \overline{KB}, \overline{KC}, \overline{KD}$. Point K moves along the curve and there will be one time instant where one of the four distances will become very small (small than a defined epsilon), in this case point A.
This process is done since the intersection points are saved randomly. After point A is detected, the function will delete point A from *intersections_All* vector, that will become {C,D,B}.
Point A will be saved in *FirstIntersectionPoint* and its parametrization value along the curve in the vector *paramOrder*.

```
Eigen::Vector4d temp1 = firstIntersection(intersections_All,
    genericCurve, false);
FirstIntersectionPoint = {temp1.x(),temp1.y(),temp1.z()};
paramOrder.push_back(temp1.w());
```

The above procedure can be applied recursively to find the order of the intersection points. If another run of the function is performed, point B will end up as *SecondIntersectionPoint*, it will be deleted from the vector and only {D,C} will remain, and so on.

Finally, the algorithm will evaluate throught *FindObstacleFromPoint* function, to which obstacle does *FirstIntersectionPoint* belong, saving it for later computation.
Procedure of *FindObstacleFromPoint* function: It is a function that takes as input a point and an obstacle and returns a flag depending if the point belong to the obstacle's circumference or not. The passage is done verifying that the point belong in the interval: $[obstacle.radius + eps; obstacle.radius - eps]$, like in figure 5.17.
When the returned flag is true, the obstacle passed as input is the one that intersects the point, point A or B in figure 5.17, the first detected obstacle is saved in a variable called *pos_first*.

Evaluating obstacle

Each intersected obstacle is saved in a vector called *visited*. If the actual intersected obstacle is not present in this vector, means that the NURBS intersects for the first time this obstacle and **State 1** is executed. Otherwise, if already visited, **State 2** is executed.

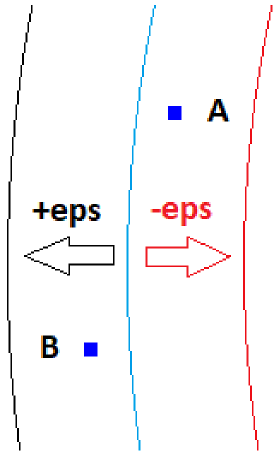


Figure 5.17: Interval to belong in the obstacle's circumference.

```
auto itV = find(visited.begin(),visited.end(),pos_first);
```

itV is an iterator vector that returns the position of the searched element in a vector. If the returned iterator is equal to the end of the vector, means that the element is not present and State 1 is executed.
State 1 is executed if the following condition is verified otherwise State 2 is executed.

```
if(itV != visited.end())
```

Approximative wayPoints calculation

When the algorithm exits from State 1 or State 2, some control points were added to avoid an obstacle or a group of nearby obstacles that was previously detected as first intersection and a single Waypoint is calculated and saved in variable *wayPointsClosest*. Waypoints are defined in particular position along the path, they are put immediately after overcoming an intersected obstacle or a group of obstacles, so the path is re-calculated after passing an obstacle.
They are called approximative since they are not definitive. The NURBS curve will change its shape iteration by iteration, and previous Waypoints may not belong anymore to the new path.
Only at the end, when the final NURBS is generated, the definitive Waypoints will be calculated.

Print NURBS information

In this state, simply the information regarding the NURBS are printed to the user. The information are: degree, knot vector, control points and weights.

This is how in general the algorithm works during the iterations. Now the process of adding a control points from a new obstacle or to an already visited is discussed.

5.3.5 State 1

In *State 1*, the algorithm enters when the intersected obstacle is visited for the first time.
The flowchart can be seen in figure 5.18.

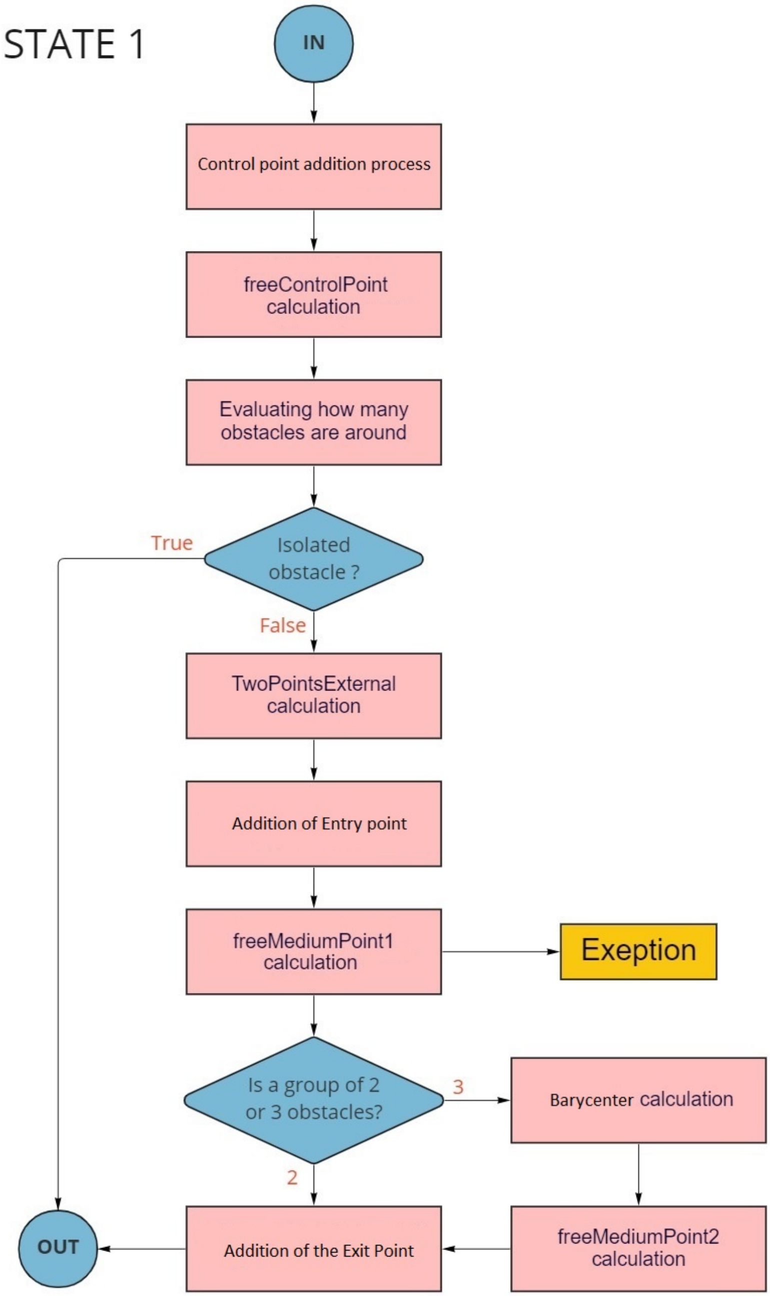

STATE 1
IN
Control point addition process
freeControlPoint calculation
Evaluating how many obstacles are around
Isolated obstacle ?
True
False
TwoPointsExternal calculation
Addition of Entry point
freeMediumPoint1 calculation
Exeption
Is a group of 2 or 3 obstacles?
3
Barycenter calculation
2
freeMediumPoint2 calculation
Addition of the Exit Point
OUT

Figure 5.18: State 1 flowchart

Control point addition process

The first operation is to add the detected obstacle in the *visited* vector:

```
visited.push_back(pos_first);
```

where *pos_first* refers to the first detected obstacle's number.
After that, the *FirstIntersectionPoint* needs to be added to the control points vector with the respective weight equal to 0.8. A weight lower that one is assign to those points that need to have a repulsive effect on the curve.

```
ctrlPoints.insert(it + index, FirstIntersectionPoint);
weights.insert(it1 + index, 0.8)};
```

The *it* variable stands for an iterator, pointing to the first element of the vector.

```
it = ctrlPoints.begin();
```

A function that finds the correct *index* where to insert the control point and its weight in the respective vectors; *ctrlPoints* and *weights* is called.

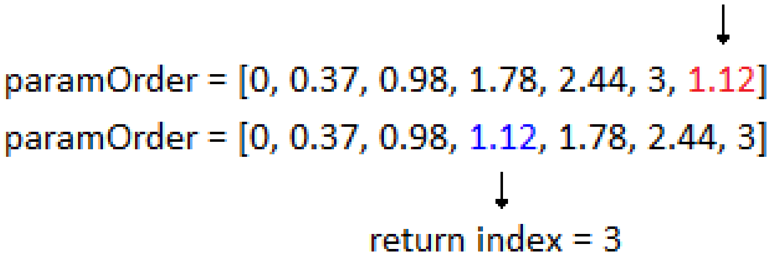
Let's now see how a control points is added in the *ctrlPoints* vector in the right index position, done by *IndexParameterOrder* function.
IntexParameterOrder is a function that takes as input a vector called *paramOrder* that contains all the orthogonal projections of the control points as parameters values on the curve, and returns the index position where to add a new control point.

```
index = IndexParameterOrder(paramOrder);
```

When a new control points has to added in the vector, the algorithm firstly calculates its parametrization thought a projection on the curve, and will put this value as last element of the *paramOrder* vector, figure 5.19.

The function takes the *paramOrder* vector as input and mainly two tasks are done:

1. **Arrangement in a non-decreasing way**: Means the function will take the last value of the *paramOrder* vector, that correspond to the new control points that as to be added, and will evaluate its position in the vector thought a comparision of each element. A non-dereasing vector is obtained:



```
paramOrder = [0, 0.37, 0.98, 1.78, 2.44, 3, 1.12]
paramOrder = [0, 0.37, 0.98, 1.12, 1.78, 2.44, 3]
return index = 3
```

Figure 5.19: Arrangement of the *paraOrder* vector.

In figure 5.19 is represented the *paramOrder* vector containg all the parametrization of the control points added up to here, in the last position the number 1.12 represents the control points which still needs to be added and for this reason the function *IndexParameterOrder* is called, to understand the index and to reorder the vector.

2. **Return**: The function return the *paramOrder* vector ordered in a non-decreasing way as can be seen in figure 5.19. A second parameter is returned, the index position where to insert the control points associated with the parameter 1.12, in this case 3, remember that in C/C++ the starting counting number is 0.

Finding out the *index*, the control point and its weight is added to the respective vectors. After that, the *UpdateNURBS* is called, that will evaluate the new knot vector based on the degree and the control points, and a new NURBS curve will be generated with the command seen before.
Finally, a re-calculation of the *paramOrder* is done since the curve has changed its shape with the addition of a new control point, and so the parametrization values saved before are no more acceptable, since error and inversion points can occur.

paramOrder update problem: The *paramOrder vector* is very important. It defines the order of the control points, and so the shape of the curve. If for absurd two control points are swapped, a totally different curve is obtained, often wrong.
In figure 5.20 an example of the wrong order of the control points is shown. In the second example, figure 5.21, the grey NURBS intersects the second obstacle while intrinsically avoiding the first. The algorithm will put two control points in order to avoid the obstacle but doing this the updated NURBS in red will now interest the first obstacle. A reasonable way is to do the same think by adding to the vector other two control points. The ctrlPoints vector will have P1, P2, P3, P4 in that specific order and this will produce the blue NURBS, totally wrong, the

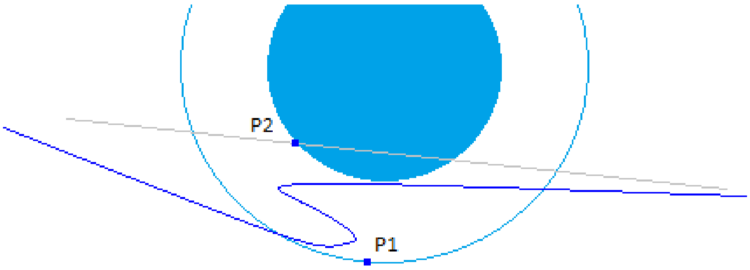


Figure 5.20: Inversion of control points (a).

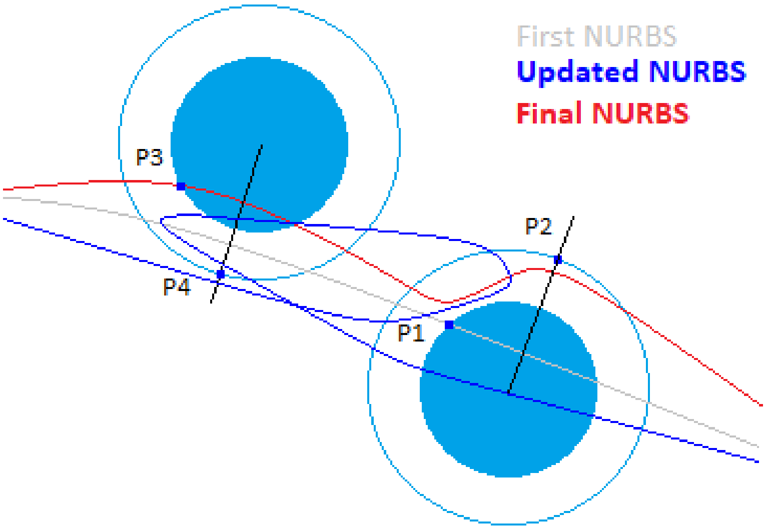


Figure 5.21: Inversion of control points (b).

control of the NURBS is lost.

To avoid this kind of behaviour an update of the ctrlPoints vector has to be done, and in particular the parametrization of the projection of the control points in the curve need to be saved. When a new NURBS is generated, the projection process has to be repeated to update the parametrization values of the points, figure 5.22. By doing this updating process, the control points will invert their position base of thier parameter values (increasing way), and a smooth NURBS is obtained, figure 5.23.

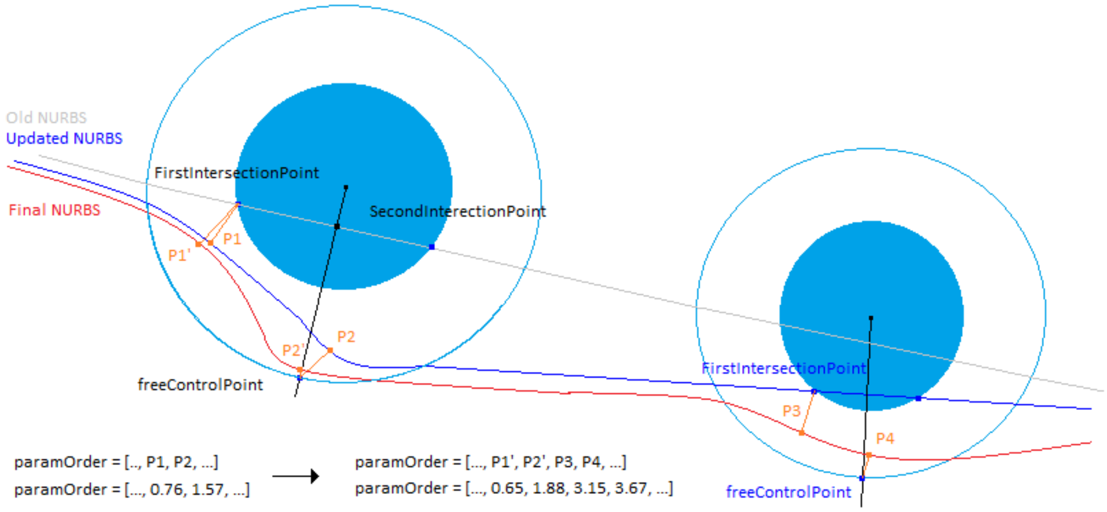

Old NURBS
Updated NURBS
Final NURBS
FirstIntersectionPoint
SecondInterectionPoint
P1
P1'
P2
P2'
freeControlPoint
FirstIntersectionPoint
P3
P4
freeControlPoint
paramOrder = [.., P1, P2, ...]
paramOrder = [..., 0.76, 1.57, ...]
paramOrder = [..., P1', P2', P3, P4, ...]
paramOrder = [..., 0.65, 1.88, 3.15, 3.67, ...]

Figure 5.22: Projection of control points and the relative paramOrder vetor

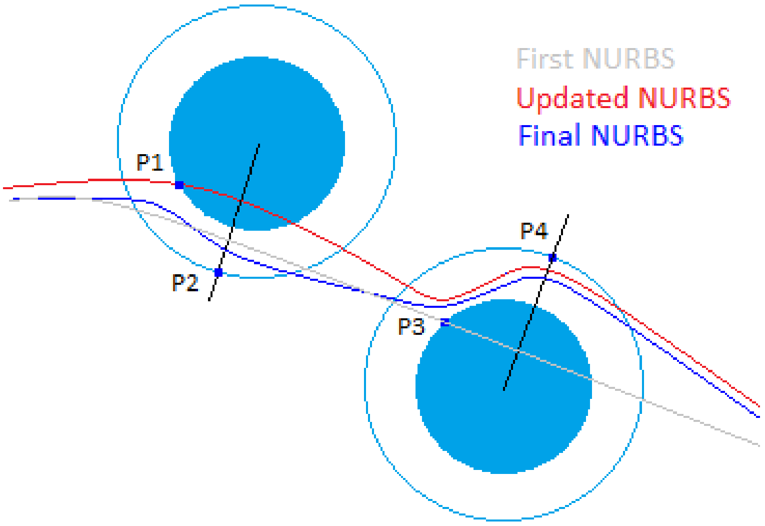


Figure 5.23: Invertion of control points correction.

FreeControlPoint calculation

Here the *freeControlPoint* is calculated and added. It was indirectly introduced in the images above. Let's analyze its importance.

From image 5.24, the *freecontrolPoint* P2 is added to bring the NURBS in the opposite direction.

Firstly, a projection of the center C on the curve is done to find the closest point K. Now a segment $\overline{CK}$ is found and from it, the angular coefficient is extrapolated

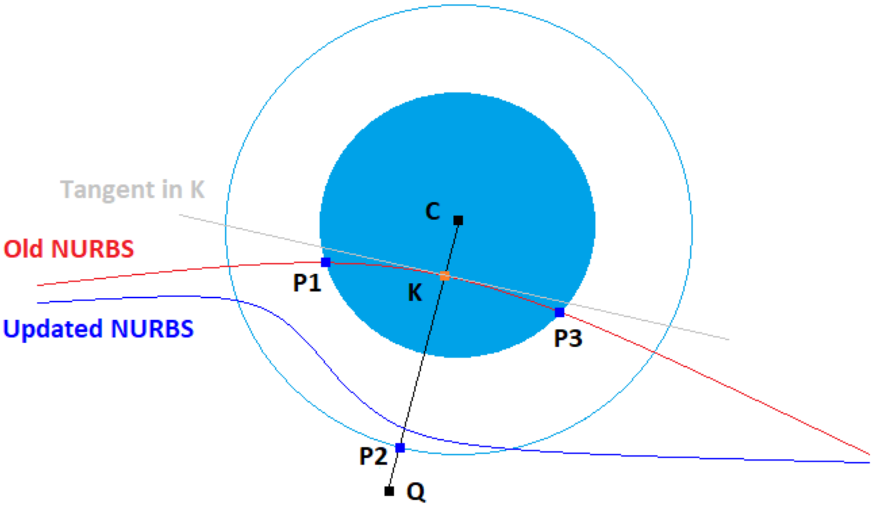


Figure 5.24: Construction of freeControlPoint P2.

to create a longer segment $\overline{CQ}$. The intersection between segment $\overline{CQ}$ and the external circumference is found in point $P2$.
The above passages are preferred because are in some sense optimal. Point K represents how close the curve pass from C and the *freeControlPoint* is added exactly in that direction to bring the closest point away fron the obstacle. This kind of operation will minimize the number of points needed to avoid the obstacle. The associated weight must be a number greater than 1 to attract the curve in that direction, my choice is to put 2.

Since a new control point has be added, the algorithm will executes the same routines as for *FirstIntersectionPoint*: find parametrization, find out the index, addition of the control point, updating the NURBS and finally recalculation of the *paramOrder*.

The *freeControlPoint* is added only if it is not contained in any obstacle. It is an important check specially when nearby obstacles are present. This check is done calling the function *EvaluatePointInObstacle*.
Procedure of *EvaluatePointInObstacle*: The function will take a point as input and internally will recall the function *FindObstacleFromPoint* for every obstacle. If for a particular obstacle *FindObstacleFromPoint* returns true, the obstacle's number is saved and the *EvaluatePointFromObstacle* will return that number, otherwise if the flag will always return false, the returned parameter remains 0 from the initialization.

Evaluating the number of obstacles around

The algorithm will set two flags; *flagFirstMediumPoint*, *flagSecondMediumPoint* and a couter *cnt*. Let's refers to the image 5.25.

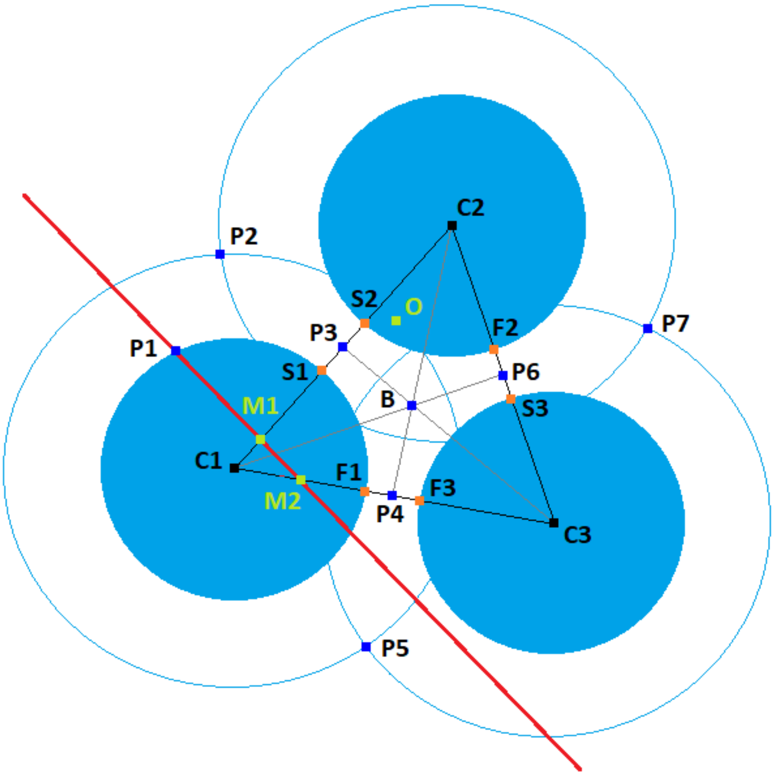


Figure 5.25: Gometric construction for NURBS in obstacles.

When the algorithm evaluates this kind of operation, it knows exactly what is the first intersected obstacle (pos_first), calculated before.
In figure 5.25 the curve detects **C1** as first obstacle, and put the first two control points; **P1** as FirstIntersectedPoint, and **O** as to freeControlPoint. From now on, when considering nearby obstacles, the *freeControlPoint* is not added to the control points vector since there are sufficient and optimize points to create a path through the obstacles, it will add only noise.

Now intersection between external circumference C1 with the other obstacles is performed. As can be seen in figure, C1 intersects C2 and C3. This two obstacles are saved in a vector called *VecObstacleIntersection* used for next computation. Next, segments creation is done, joining the centers; $\overline{C1C2}, \overline{C1C3}$ only.

Flag setting: The red NURBS is figure 5.25 represents the curve that intersects an obstacle when trying to reach the goal, is the fundamental passage to understand if intersections are still present or not.
The red curve intersects segment $\overline{C1C2}$ in M1 and $\overline{C1C3}$ in M2, from this is clearly understandable that the NURBS needs to pass on the half-right of the C1 obstacle through two pairs of obstacles, C1-C2 and C1-C3.
Through *firstIntersection* function the algorithm understand what is the first point intersected, M1, and so the first pair of obstacle, C1-C2, while M2 and C1-C3 will represents the second pair. The first pair obstacle is saved in *FirstMedianObstacle* variable, used for later computation.

- *flagFirstMediumPoint* represents that the NURBS has to pass between the first obstacle intersected C1, and the first pair obstacle C2.
- *cnt* represents how many times the NURBS intersects a segments that starts form C1, such as $\overline{C1C2}$ and $\overline{C1C2}$, it can be 0,1 or 2.
- *flagSecondMediumPoint* represents that the NURBS has to pass between the first pair obstacle, C1-C2, and between the pair of obstacles saved in the *VecObstacleIntersection*, means C2-C3.

Referring to figure 5.25, the NURBS intersects a the segment from C1, means the flag *flagFirstMediumPoint* is set to true, and a counter cnt is set to 1 or 2 depending if only one or two segments are intersected, in this case 2.

If the vector *VecObstacleIntersection* contains more than one obstacle, the algorithm will evaluate the additional flag *flagSecondMediumPoint.* For doing this, intersection between NURBS and segment $\overline{C2C3}$ is evaluated. If no intersection is present, means the NURBS is not going in that direction, the flag will remain false, otherwise it will be set true.

flagSecondMediumPoint is very important, and very important is to set it true only when cases similar to figure 5.25 arises. In figure 5.26, there are two examples where the first detected obstacle (C1) intersects two obstacles while the NURBS has to pass only between two of them and not three.

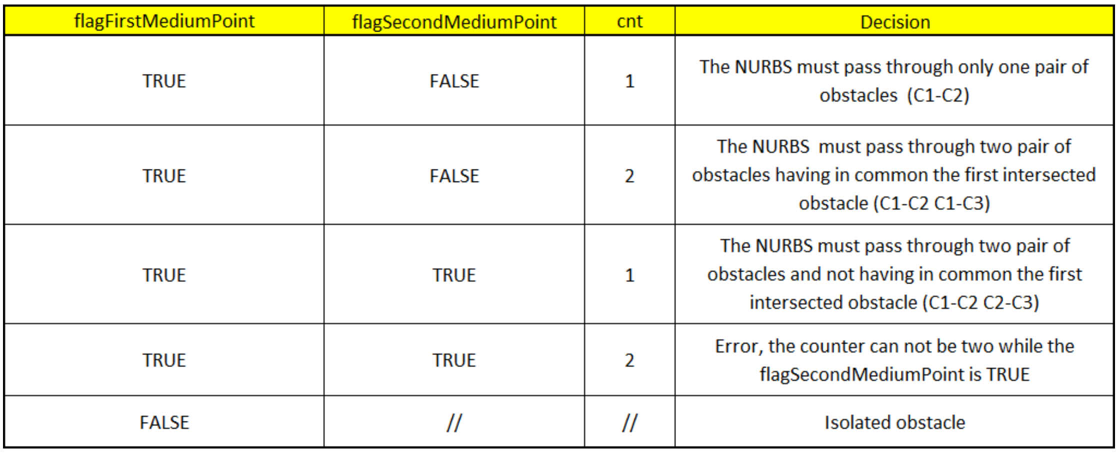
Referring to figure 5.26 left, C1 represents the first intersected obstacle, and when evaluating which obstacle does the external circumference intersects, two obstacle are detected, C2 and C3. As can be seen, the NURBS must pass through only a pair of obstacles (C1-C2), but three were detected.
The NURBS intersects only one segment from C1, so the cnt will be set to one and

the flagFirstMediumPoint is true. Despite, flagSecondMediumPoint seems to be true since the curve intersects the other segment (C2-C3), but this behaviour is wrong. To adjust this behaviour in the left picture the parametrization is involved, means that point B must always be found for first, and then point A. Looking at the parameter A and B along the curve is easy to impose as condition; parametrization value of B must be lower that A.

Now, another case can arise, picture on the right. The pametrization of the two point A and B seems to be fine, but this condition is not sufficient and one more condition to turn true the *flagSecondMediumPoint* is necessary. This condition considers whether the two obstacles in the *VecObstacleIntersection* intersects each other, in particular if at least one of the two external circumferences intersect the other obstacle.
The condition that has to be satisfied to turn true the *flagSecondMediumPoint* will become:

```
if(!intersections.empty() && (temp1.w() >= FirstMedianParameter) &&
    (!intersections1to2.empty() || !intersections2to1.empty()))
```

intersections refers to the point given by the intersection between NURBS and the segment created by joining the centers of the obstacles present in *VecObstacleIntersection*. *temp1.w()* refers to the parameter of the previous point that has to larger that the FirstMedianParamter point. intersections1to2 and intersections2to1 refer to the intersections between C2 and C3.
In figure 5.27, a table representing the condition for turning on the two flag and set the counter is shown, refers to image 5.25.



flagFirstMediumPoint	flagSecondMediumPoint	cnt	Decision
TRUE	FALSE	1	The NURBS must pass through only one pair of obstacles (C1-C2)
TRUE	FALSE	2	The NURBS must pass through two pair of obstacles having in common the first intersected obstacle (C1-C2 C1-C3)
TRUE	TRUE	1	The NURBS must pass through two pair of obstacles and not having in common the first intersected obstacle (C1-C2 C2-C3)
TRUE	TRUE	2	Error, the counter can not be two while the flagSecondMediumPoint is TRUE
FALSE	//	//	Isolated obstacle

Figure 5.27: Table for evaluating where the NURBS have to pass

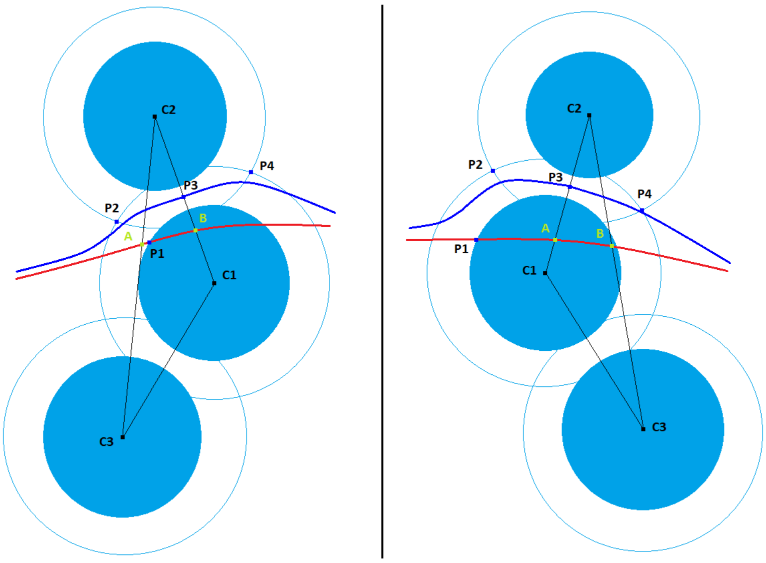


Figure 5.26: Three near obstacles detected but only two are used.

If the algorithm ends up in the 5^{th} case of table 5.27, the obstacles is treated as isolated as is shown in image 5.28. This choice is done based on the red NURBS that not intersects any segment, and have to pass on the half-left of C1.

TwoPointsExternal calculation

For two points external calculation, I refer to the calculation of important two control points; the first to enter in group of obstacles and the last to exit from the group. Depending of the table 5.27 different cases arise.

1. Case 1: A single pair of obstacles is present. It end up in calculating the intersections points between external radius, figure 5.29

   Since the red NURBS intersects only one segment, in this case in M, the cnt is 1 and the flagFirstMediumPoint is true. Intersection between the two external radius are found and reordered based on their parametrization is done, to understand which of them is the first to be put in the ctrlPoint vector. Point O represents the *freeControlPoint* and is easy to see that is inside of C2, for this reason when consider near obstacle this point is not considered, it will be overwritten be the **Entry Point**.

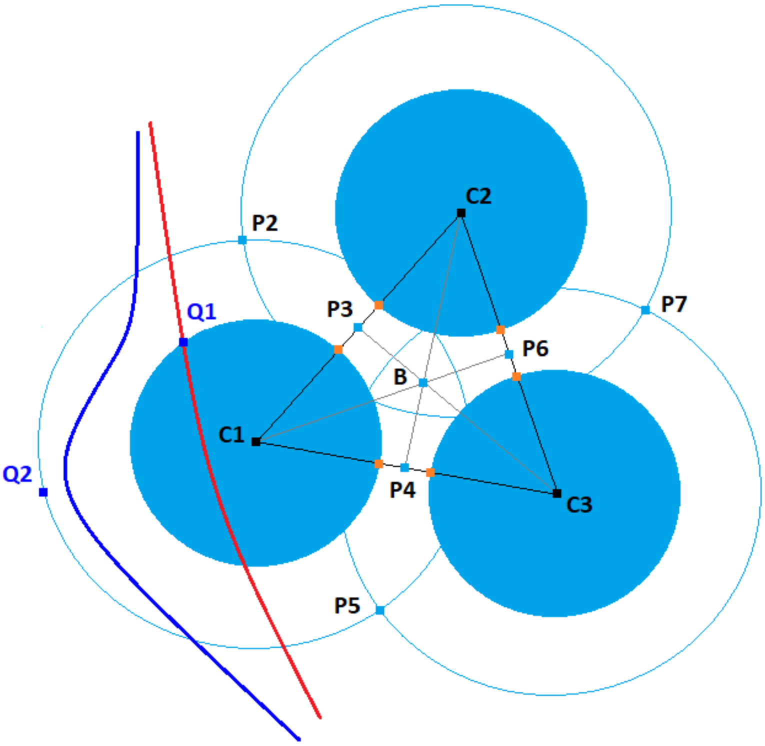


Figure 5.28: Obstacle C1 treated as isolated.

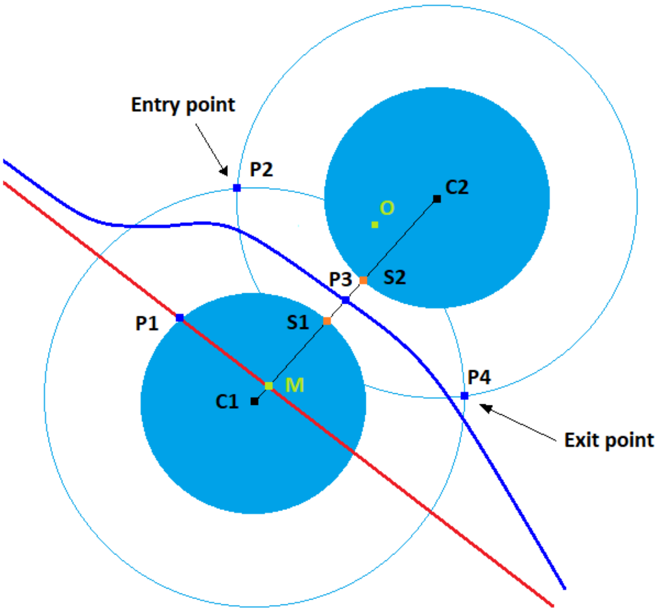


Figure 5.29: One pair obstacles

2. Case 2: In this case, cnt is 2 since the red NURBS intersects two segments from C1 in M1 and M2. Here obviously the *flagSecondMediumPoint* remains false.

   In figure 5.30, to detect the entry point and the correct exit point, the algorithm saves in a vector the points obtained intersecting the first obstacle detected (C1) with the elements of the *VecObstacleIntersection* which contain C2 and C3.
   The entry point is found by considering the distances between *FirstIntersection* point P1 or *freeControlPoint* (depending if this last one is inside of an obstacle) and the points of intersections. The minimum distance will turn on to be P2. The exit point works in the same way, considering now distances from the goal.

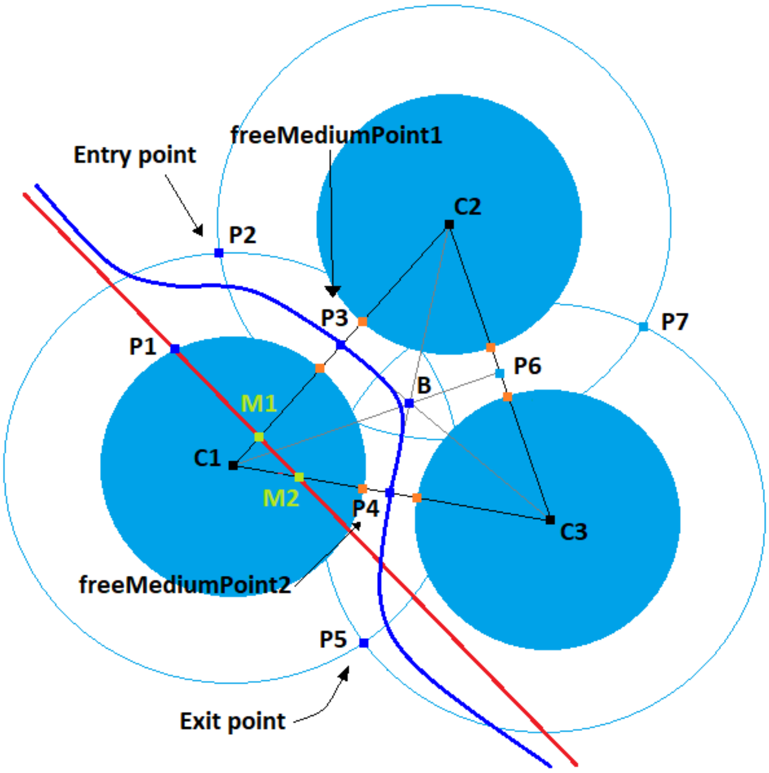


Figure 5.30: Two pair obstacles having in common the first obstacle detected, C1, case 2.

3. Case 3: The red NURBS now intersects only once the segments from C1, means cnt equal 1, and will cross segment $\overline{C2C3}$ in point M2, means *flagSecondMediumPoint* is true, figure 5.31.

The procedure for detecting the two external points is the same as before, but the points are different. In this case intersections between C1 and C2, and between the two element of the *VecObstacleIntersection*, C2 and C3, is considered.

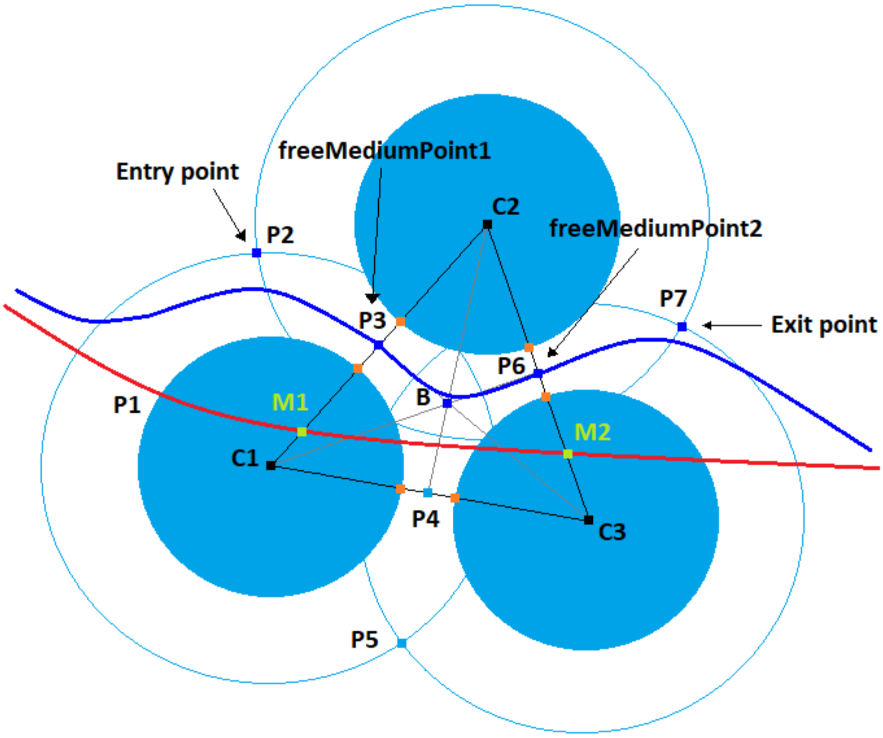


Figure 5.31: Two pair obstacles not having in common C1, case 3.

4. Case 4: This case is an error, since it is impossible that the NURBS will exit from both passages.

5. Case 5: Since the Q1 and Q2 already has been added to the ctrlPoint vector and since the NURBS will pass on the left-half side of C1, no more points have to be considered, the algorithm will return in the for loop for evaluating next obstacle intersection.

After considering the correct case, the algorithm will follow the same procedure of adding control point to the *cltrPoints* vector. After that the entry point will be considered in the NURBS generation.

freeMediumPoint1 calculation

Referring to figure 5.30, 5.29, 5.28 and 5.31, no matter in which case the NURBS is, the next point to be added is the same, in this case point P3, called *freeMediumPoint1*. For the computation of P3, points S1 and S2 are found by considering intersections between the segment *C1C2* and C1, and *C1C2* and C2 obstacles.
P is simply the medium point between S1 and S2:

$$P_x = \frac{S1_x + S2_x}{2} \quad P_y = \frac{S1_y + S2_y}{2}$$

Now point P3 is added to the control points following the update NURBS procedure. If P3 were to be found within an obstacle, means no space between two obstacles is present, the algorithm through an exception, since can pass between them.

Evaluation if it is a group of two or three obstacles

Two more control points are decided to be added based if the NURBS is the 2^{th} or 3^{rd} case of table 5.27, basically when three nearby obstacles are detected. The condition that verifies the previous cases is:

```
if(cnt==2 || flagSecondMediumPoint);
```

Basically, when *cnt* is equal 2 or the *flagSecondMediumPoint* is true, the algorithm needs to add two more points; the barycenter B and the freeMediumPoint2 P4 or P6.

Barycenter calculation

Barycenter B between the three centers is calculated using geometry formula:

$$B_x = \frac{C1_x + C2_x + C3_x}{3} \quad B_y = \frac{C1_y + C2_y + C3_y}{3}$$

This procedure works always because the *flagSecondMediumPoint* is set only in particular condition and only when three near obstacles are close each other.
The point B is then added to the control point vector and the update procedure of the NURBS starts.

freeMediumPoint2 calculation

The *freeMediumPoint2* corresponds to point P4 or P6 depending on which cases the NURBS enters. The above points are calculated taking into consideration the correct pair, C1-C3 for P4 and C2-C3 for P6. This point is calculated in the same way as *freeMediumPoint1* by taking the medium of $\overline{F1F3}$ or $\overline{F2S3}$.

Addition of Exit Point

Finally, the last control point is added to the control points vector. The last point represents the exit point found in the passages above from the two external intersection. Update procedure for adding a control point is executed.

State 1 ends here.

5.3.6 State 2

State 2 represents the state when the NURBS intersect an already visited obstacle. A NURBS can intersect many times the same obstacle for different reason. It can happen that the *freeControlPoint* is not enough to bring the NURBS away from the obstacle.
The general flowchart, in figure 5.32, shows the passages that are executed in this state. Below, in figure 5.33, is illustrated the procedure of adding Q1 and Q2 as new control points.

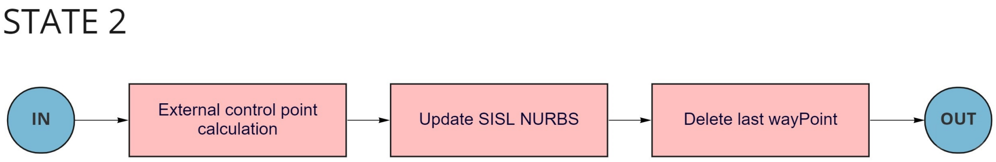


Figure 5.32: State 2, already visited obstacle.

The construction of these point is very easy. P1 is the *firstIntersectionPoint* for the grey straight NURBS, and P2 the *freeControlPoint*. Both are added to the control points vector, after that the orange NURBS is obtained. Iteration and intersections are performed and K1 if found as *firstIntersectionPoint* for the orange curve. A straight line starting from C that pass through K1 and intersects the external circumference in Q1 is created. Q1 is now simply added to the control points vector and after that the update procedure of NURBS starts. Finally, the previous Waypoint is deleted since the NURBS was not able to pass the obstacle.

The passages above, of adding a point Q1, are continuously done at each iteration if an already visited obstacle is intersected. For examples, in figure 5.33, if the red NURBS created by adding the Q1 control points still intersects the obstacle, another point Q2 is added with the same passages.

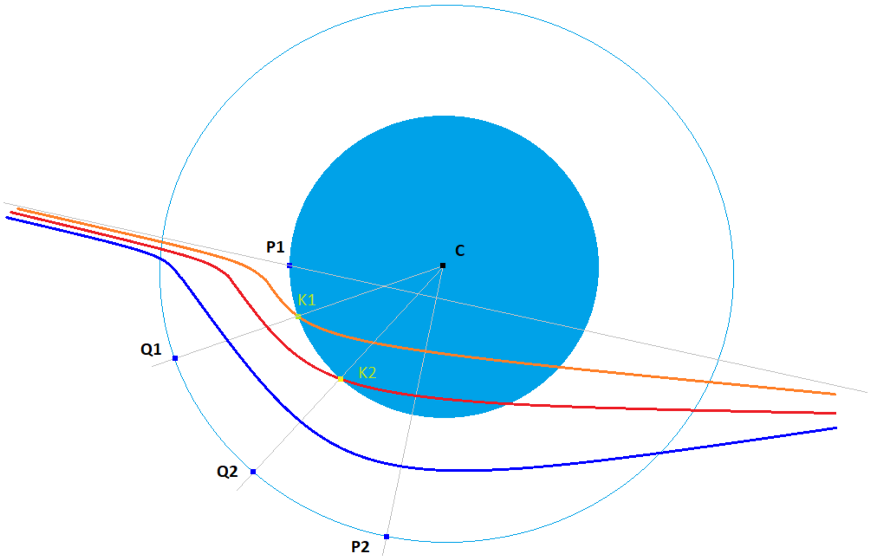


Figure 5.33: Intersections in the same obstacle.

Chapter 6

Test and Simulations

In this section, it will be presented some images of the path planning with NURBS, directly taken from the OpenGL render. Next, a comparison with A* is done, and after a dynamic simulation of the path, with a re-planning in the Waypoints is shown. Finally a real-case simulation using Moon and Mars surfaces is done.

6.1 Simulation of NURBS path

Some examples of NURBS paths are presented.
Figure 6.1, shows the iterations that the NURBS algorithm does to avoid the obstacles, with the final NURBS from start to goal.

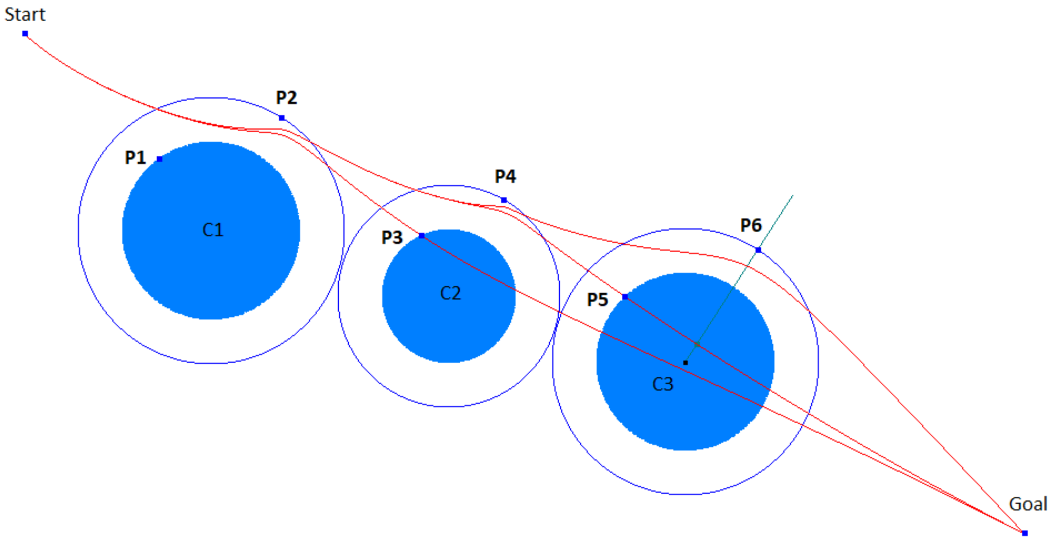


Figure 6.1: NURBS iterations

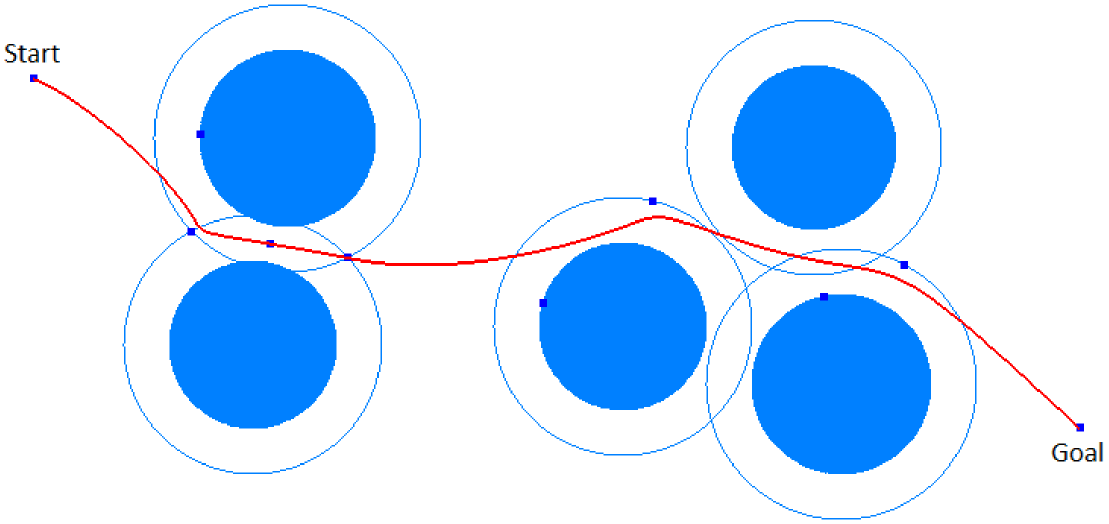


Figure 6.2: NURBS trajectory

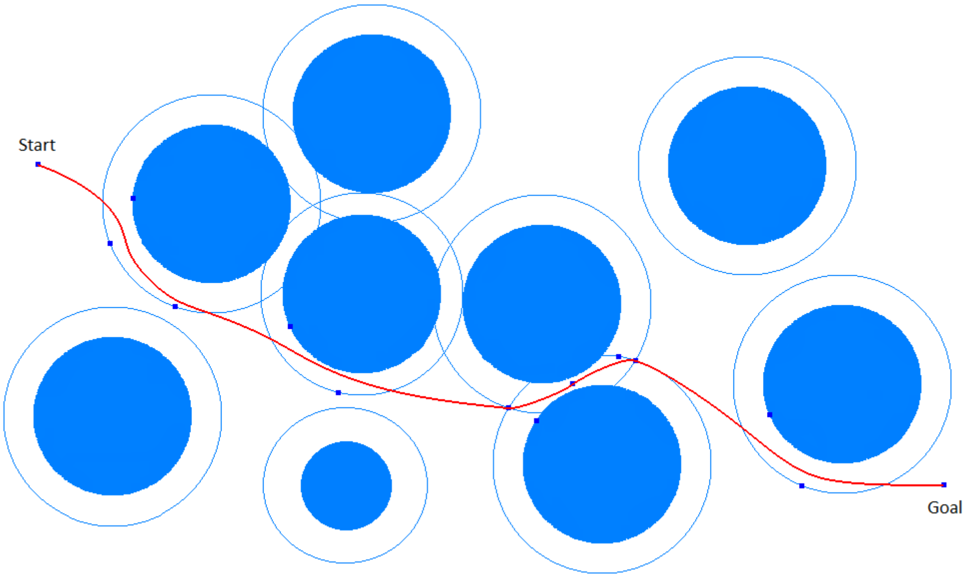


Figure 6.3: NURBS trajectory

Figure 6.2 and 6.3 show two examples of NURBS path generation.

6.2 Comparison between NURBS and A*

Two examples of NURBS and A* path are presented.

Example 1

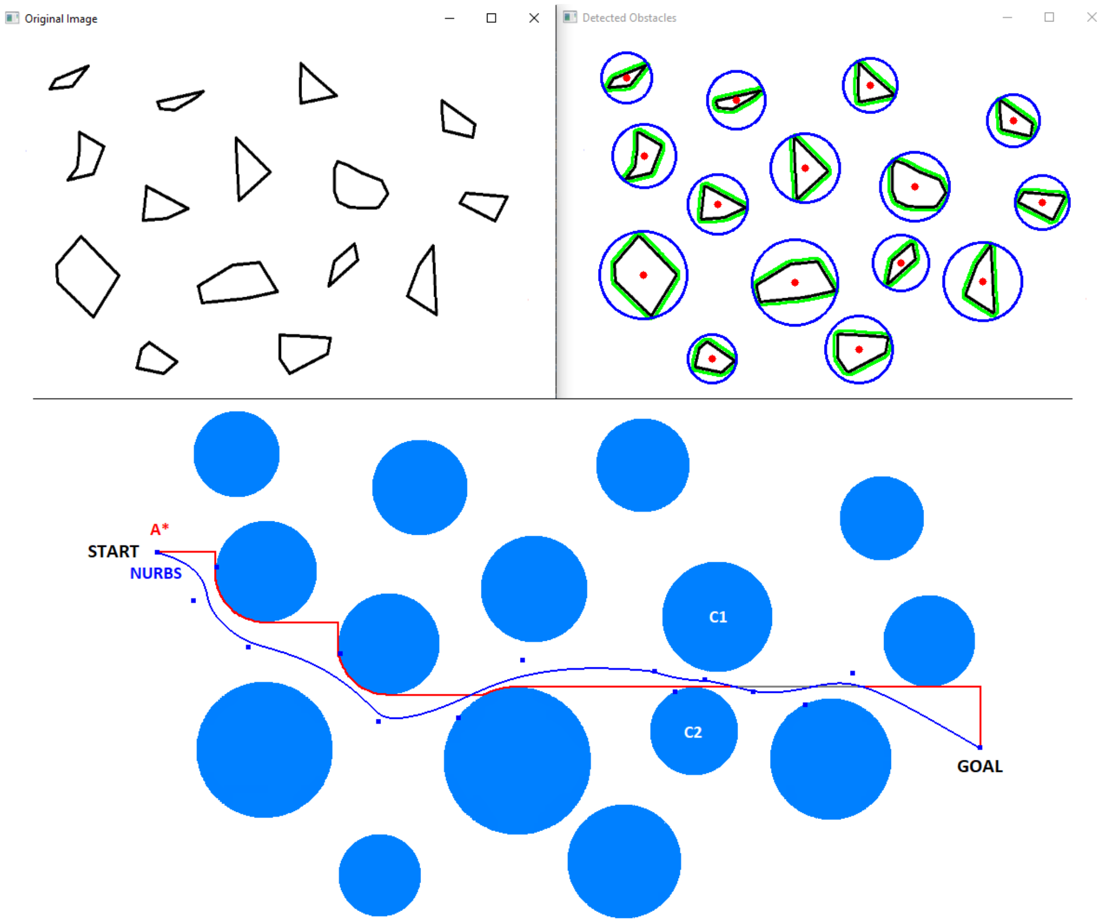


Figure 6.4: Detected obstacles and NURBS/A* trajectory

Figure 6.4 shows the obstacles detected using the *Obstacle Detection Algorithm* seen in section 5.1, chapter 5, based on Computer Vision. The NURBS path in blue shows a very smooth flow respect to the A* path, in red. The blue squares dots represent the control point of the NURBS, and only fifteen points are needed to describe the path. In the other hand, the A* is the closest possible path from start to goal but passes near obstacles. A totally different behaviour is present on the NURBS, it tends to stay equally spaced from the obstacle, in particular when near obstacles are present, such as C1 and C2.

Example 2

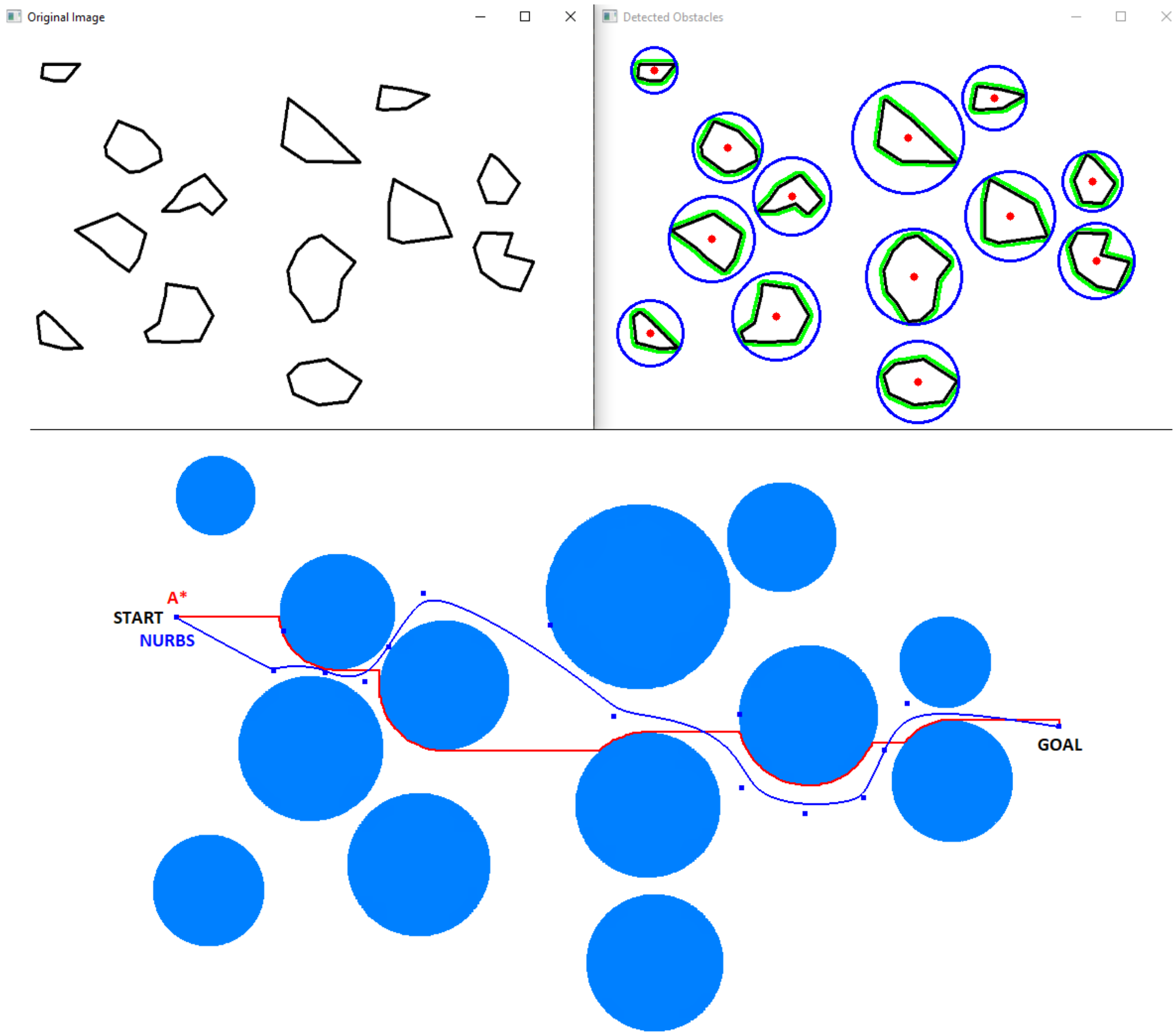


Figure 6.5: Detected oobstacle and NURBS/A* trajectory

In figure 6.5 another example is presented. Similar consideration can be made between the blue NURBS and the red A*.

Using the theoretical formula 4.15, to complete compute the NURBS show in the previous examples, only the control points and the degree of the curve are needed. Storing these variables, the NURBS can be recomputed when needed in the machine.

Memory usage comparison

A* and NURBS show a different behaviour from a point of view of the memory usage.

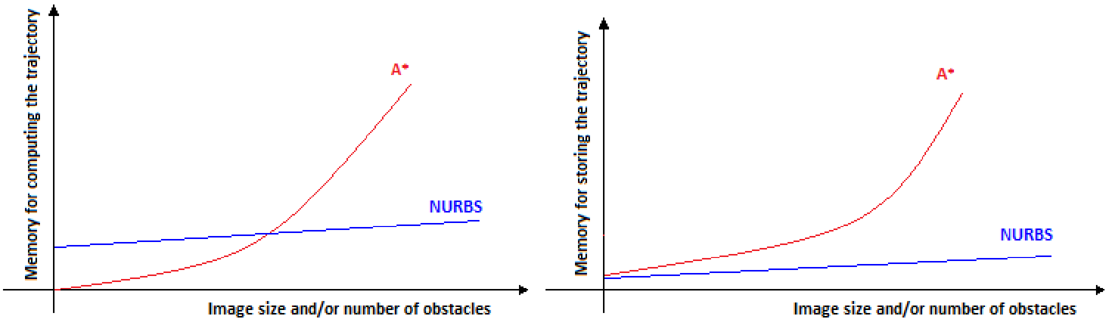


Figure 6.6: Memory usage between A* and NURBS

Figure 6.6 left, shows that the runtime memory needed to compute a NURBS trajectory is greater that A* but almost constant when the image size or the number of obstacles increases. The memory for computing a NURBS trajectory becomes independent from the environment complexity. Despite A* usage memory depends of how big is the generated graph from the input image, bigger and high resolute image may take up more memory that NURBS.

In the other hand, figure 6.6 right, shows the memory needed to store the generated trajectory. Saving the control points, weights, and the degree of the NURBS it is possible to store the trajectory. Despite, the A* algorithm needs to save each "pixel" or "grid node", with the relative predecessor. It is understandable that having a bigger graph, more nodes have to be store and the memory usage becomes dependent from the image size and obstacles, different from NURBS that is independent.

6.3 Dynamic path re-calculation

In this section, a dynamic re-calculation of the path in the Waypoints is executed. Basically the rover moves along the trajectory and when a Waypoint is reached, the rover will re-executed the path calculation based on another acquired image, if new obstacles are detected the rover will follow the new path, if not it will proceed with the previous one.

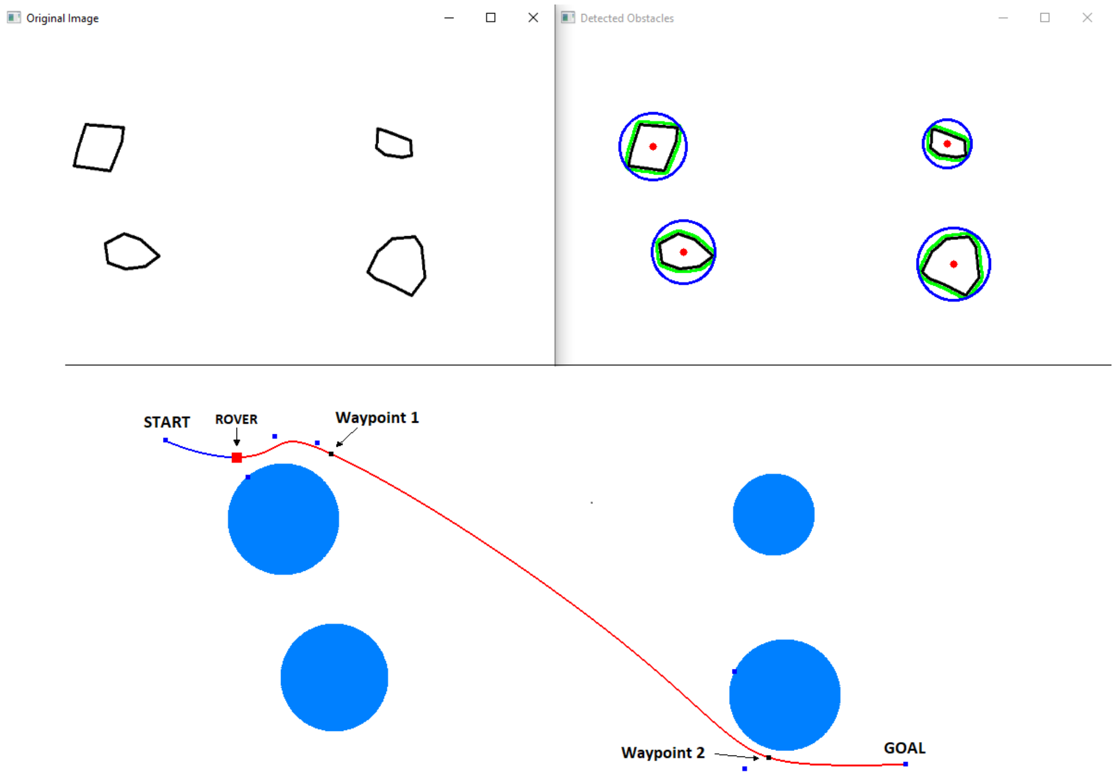


Figure 6.7: Dynamic path step 1

Figure 6.7 shows the obstacles detected and the path generated. The NURBS path is shown in red, while the blue curve shows the piece of the route already made from the rover, the red square. The black squares along the path represents the Waypoints in which a re-calculation of the path is executed if new obstacles are detected.
Figure 6.8 shows two re-calculations of the paths based on the new detected obstacles, first C1 and next C2-C3.

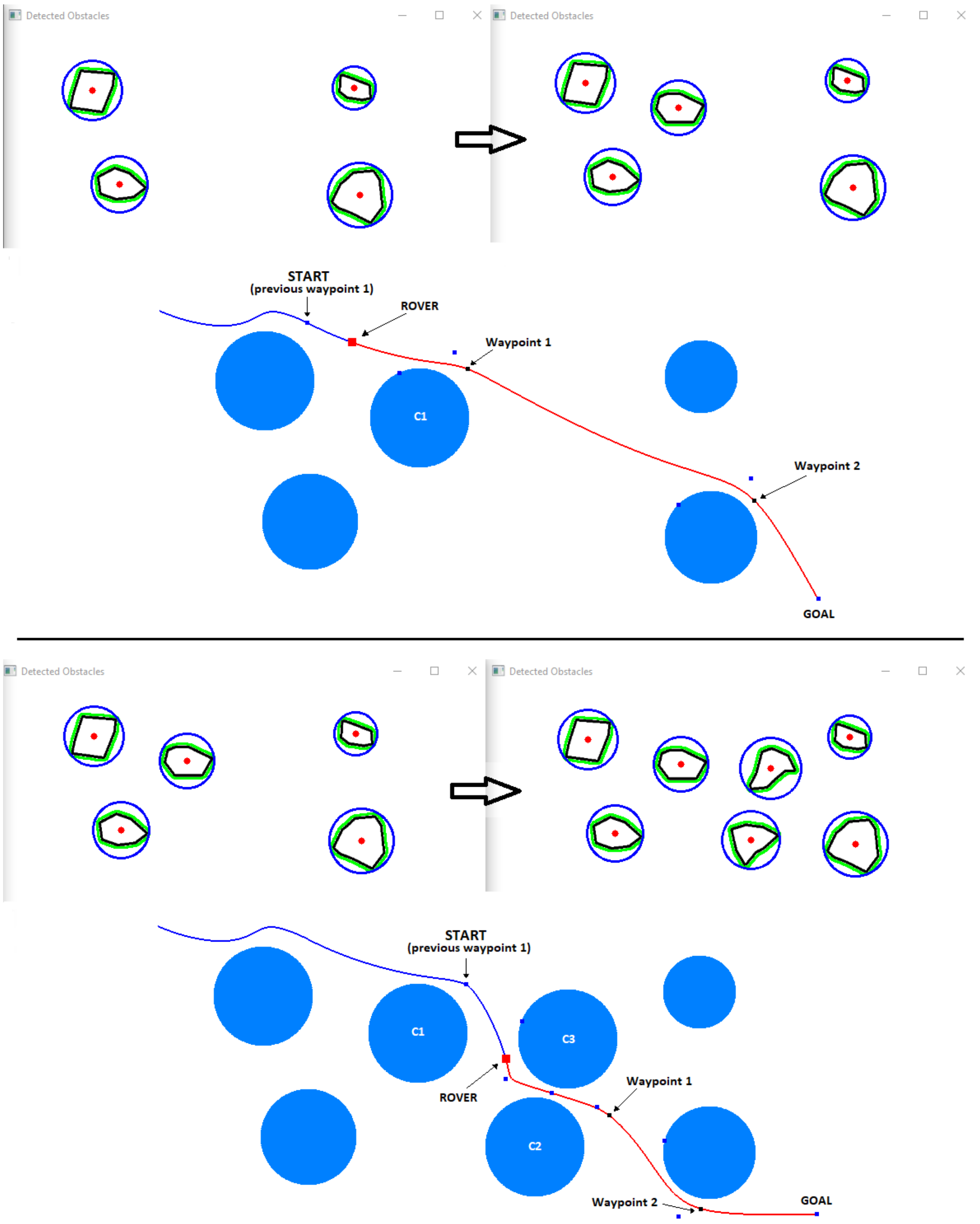


Figure 6.8: Dynamic path step 2

No new obstacles are detected and the path will remain the same till the goal.

6.4 Real surface planetary paths

In this section, the algorithms are tested on real surface images of Mars and Moon.

6.4.1 Moon surface test

Figure 6.9 shows the obstacles detected in a moon surface. The environment is very complex and it is a very challenging case for the NURBS path planner.

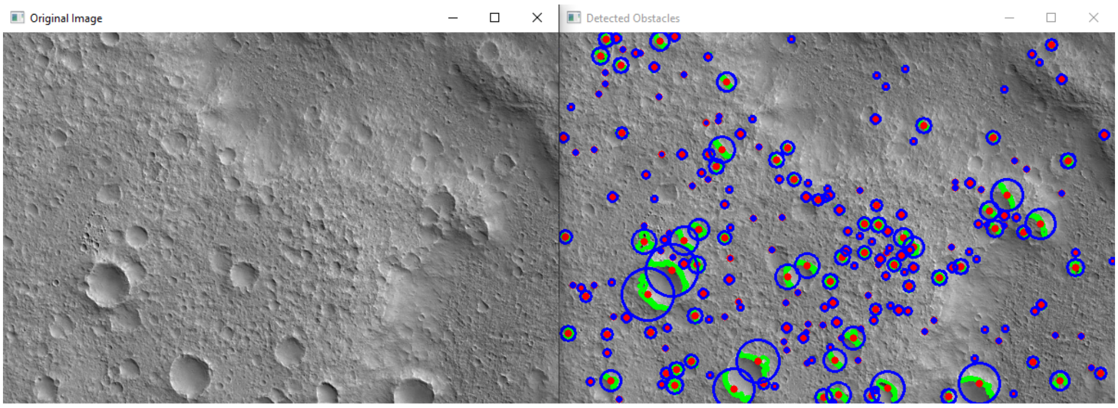


Figure 6.9: Moon surface and obstacles

Some examples of trajectory generation is shown in the figures below.

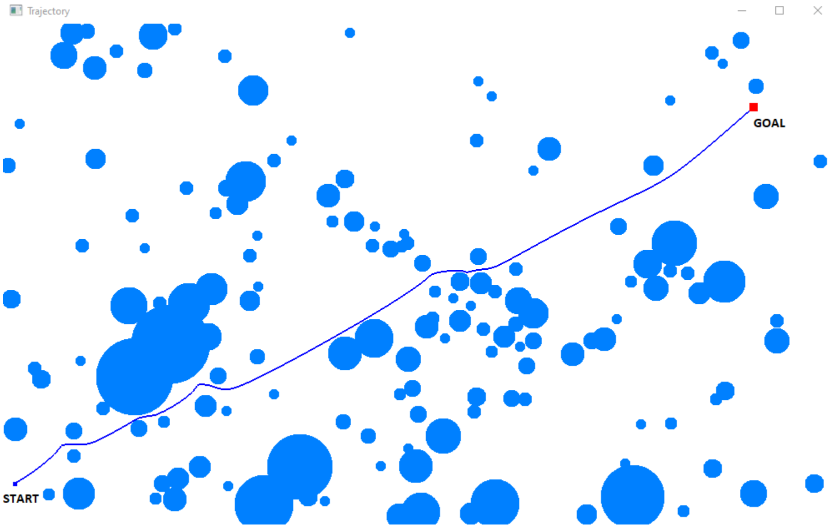


Figure 6.10: Trajectory on Moon surface

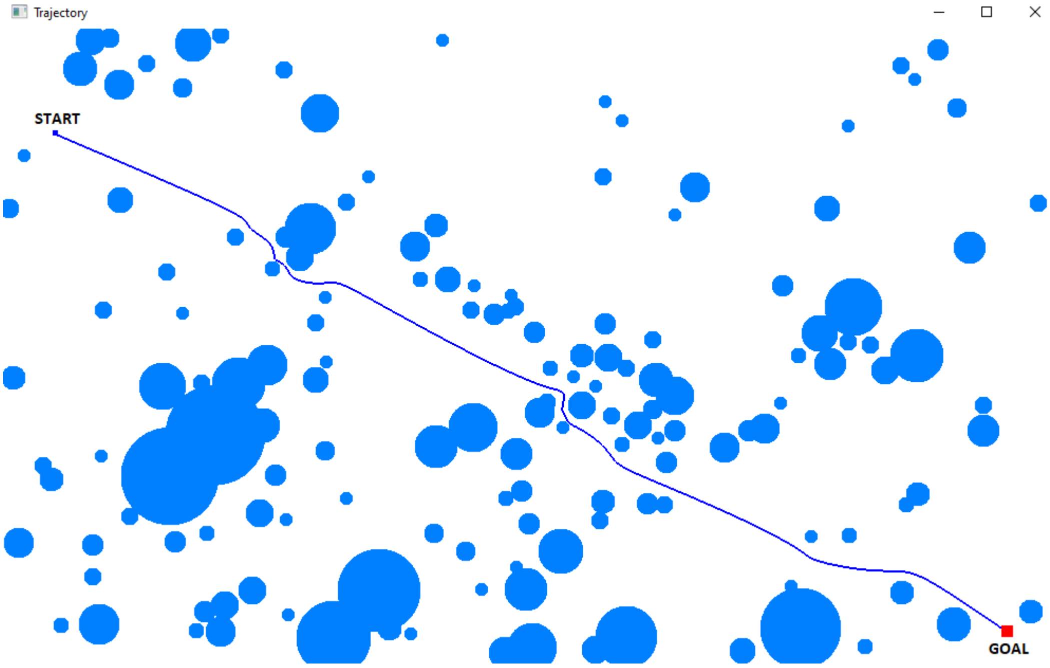


Figure 6.11: Trajectory on Moon surface

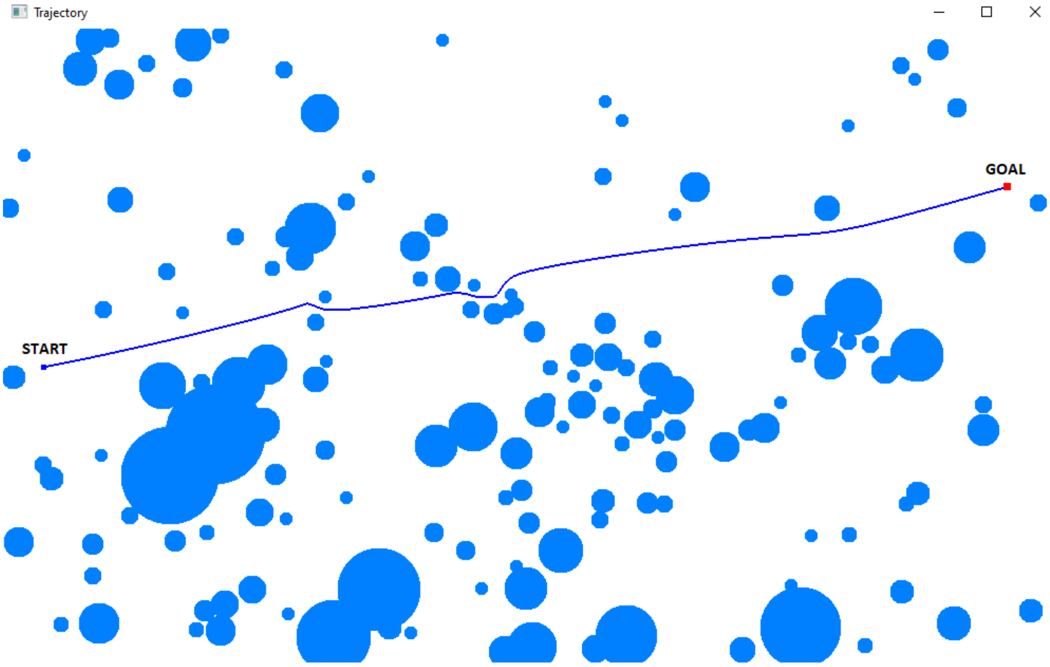


Figure 6.12: Trajectory on Moon surface

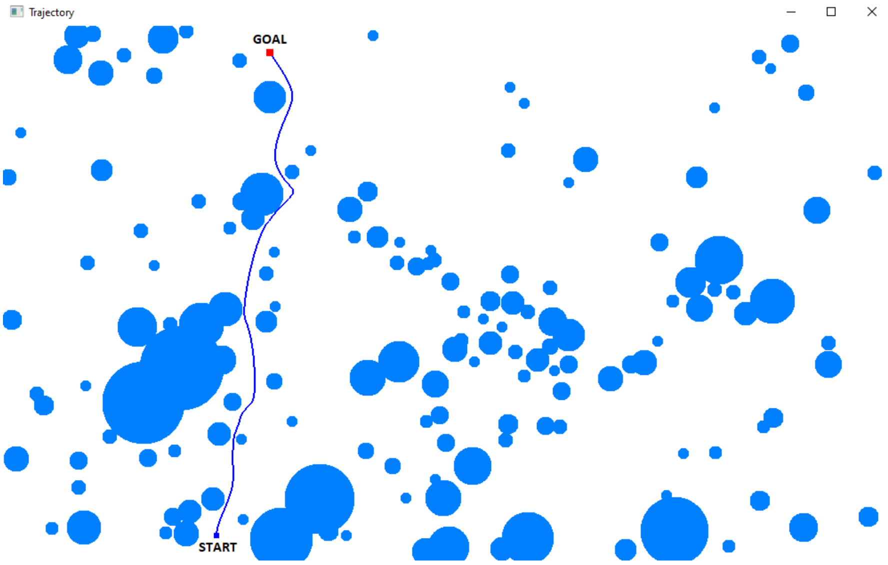


Figure 6.13: Trajectory on Moon surface

The above images show the trajectories generated using different start and goal positions. The external circumference and the control points are not printed to make the pictures more clean.

6.4.2 Mars surface test

Now, a mars surface image is used to test the two algorithms. Results can be seen below.

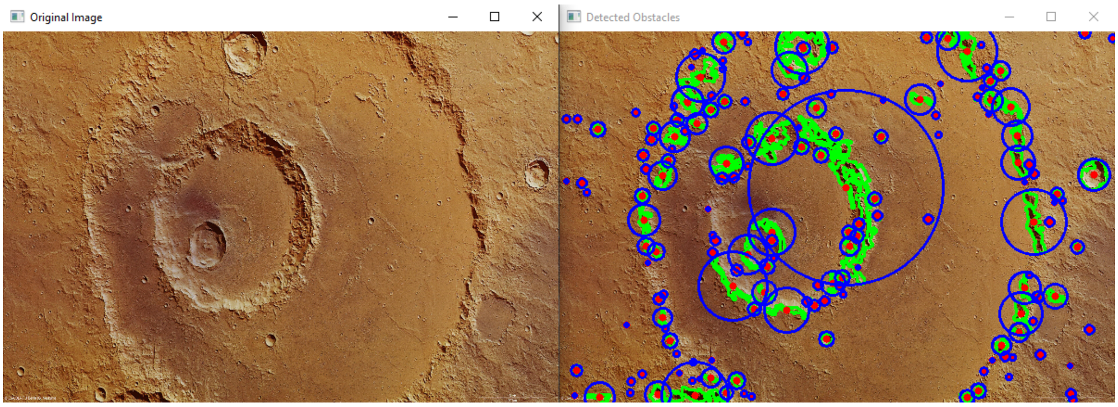


Figure 6.14: Mars surface and obstacles (Hardley Crater).

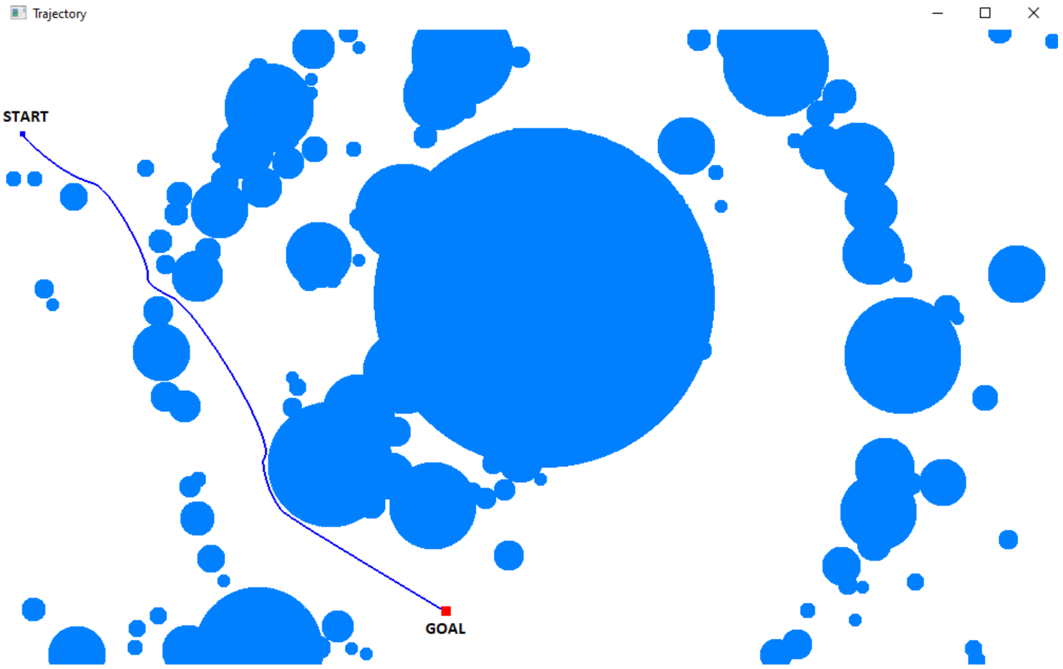


Figure 6.15: Trajectory on Mars surface

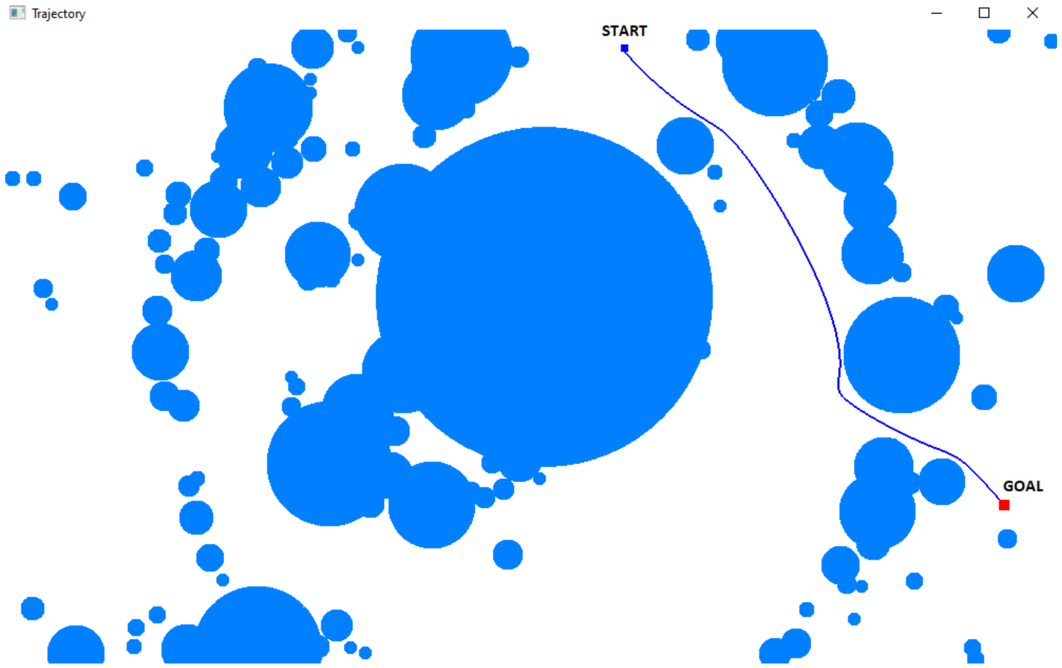


Figure 6.16: Trajectory on Mars surface

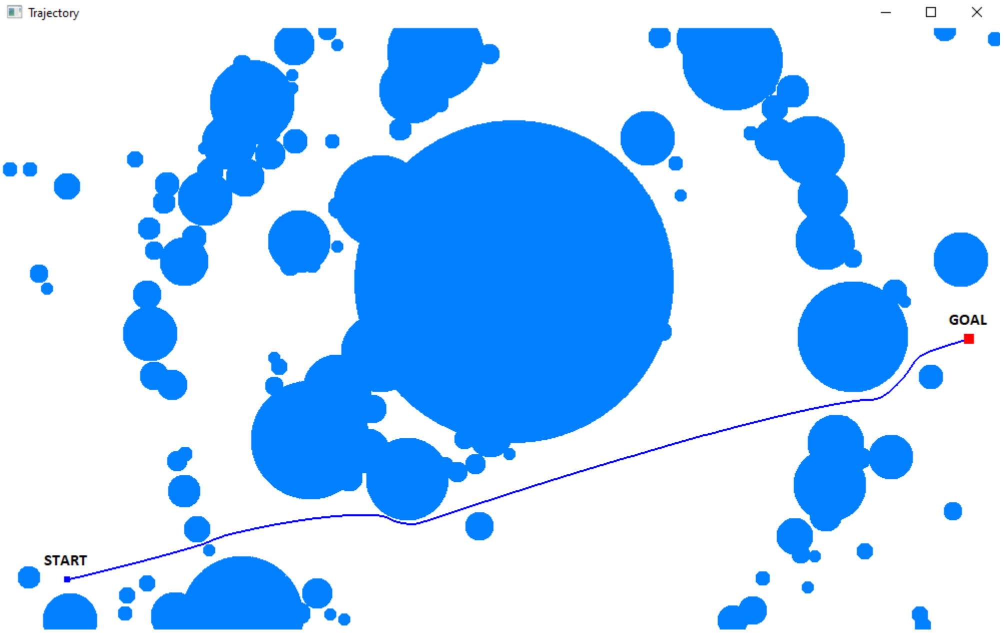


Figure 6.17: Trajectory on Mars surface

Chapter 7

Conclusions

In this thesis, an alternative path planner method for mobile robots and rovers has been created and analysed in detail. An algorithm that implements such path planner is applied in the context of planetary explorations. The algorithm, written in C++, uses Computer Vision to detect the obstacles and NURBS theory to trace the trajectory. The algorithm is developed starting from SISL library, developed by a researcher group at SINTEF in Oslo, and SISL toolbox library, created by Antonino Bongiovanni.

NURBS path planner algorithm has been compared with a A* algorithm using the same environment and same obstacles; the results show that NURBS trajectories achieve very smooth paths, no sharp turns are present and no point-turn maneuvers are present, despite A* shows an opposite behaviour, it finds the closest path, but 90 degrees angles are present in the trajectories. In the other hand, NURBS generates a curve of degree three independently from the environment and from the number of obstacles, it is fixed a priori. Very smooth trajectories, no point-turn maneuvers, low degree complexity and constant memory usage independent from the environment make this kind of method a very alternative in path planning for planetary explorations.

A* analyses each pixel of the image, or each grid that composed it, and so the computational complexity depends on how much resolute the image is and so how big would be the generated graph. The complexity is proportional to the sum of the nodes and branches. For big resolute image, the A* becomes heavily and memory consumed. The complexity of NURBS algorithm depends on how many times the NURBS intersects the obstacles, it is totally independent from the resolution or from the grid image since it does not use a graph as input.

Few variables are needed to save a NURBS curve, degree, and control points;

using these variables a NURBS curve can be calculated in any time. More memory is needed in real-time processing of the NURBS, since temporary variable are needed to evaluate geometry properties for generating the NURBS. It is important to remark that the total amount of memory needed during the NURBS path generation is almost fixed, independent from the cases in which the path must be generated.

Future Works

- Optimization of the algorithm, from a point of view of C++ programming techniques, control points positioning and from a usage total memory.
- ROS implementation with NAV2 plugin for real simulation in the rover to be used and test in ROXY facility at TASI (Thales Alenia Space Italia).

Bibliography

[1] Sasiadek, J. (2014). Space robotics - Present and past challenges. 2014 19th International Conference on Methods and Models in Automation and Robotics, MMAR 2014. 926-929. 10.1109/MMAR.2014.6957481.

[2] Sasiadek, J.Z., 1992 and Sasiadek, J.Z. 1994.

[3] Matijevic, J., 1998, "Autonomous Navigation and the Sojourner Microrover".

[4] "Mars Science Laboratory Rover in the JPL Mars Yard". NASA/JPL.

[5] NASA site

[6] "The ExoMars Programme 2016–2018". European Space Agency (ESA). 2015. Retrieved 16 March 2016.

[7] "ExoMars Rover Vehicle Perception System Architecture and Test Results" McManamon, K, ASTRA 2013.

[8] Salichs, Miguel. (2001). NAVIGATION OF MOBILE ROBOTS: LEARNING FROM HUMAN BEINGS.

[9] Thomas Bräunl. EMBEDDED ROBOTICS.

[10] Lee, Gim & Jr, Marcelo. (2008). Mobile Robots Navigation, Mapping, and Localization Part I. 10.4018/9781599048499.ch158.

[11] Bora, Leanardo & Lancaster, Richard & Nye, Ben & Barclay, Chris & Rubio, Sergio & Winter, Matthias. ExoMars rover control, localisation and path planning in a hazardous and high disturbace environment.

[12] Van Pham, Bach & Maligo, Artur & Lacroix, Simon. Absolute map-based

localization for planetary rover.

[13] Lourakis, Manolis & Chliveros, Georgios & Xenophon, Zabulis. Autonomous visual navigation for planetary exploration rovers.

[14] Robert D. Christ, Robert L. Wernli,Chapter 17 - Navigational Sensors, Editor(s): Robert D. Christ, Robert L. Wernli, The ROV Manual (Second Edition), Butterworth-Heinemann, 2014, Pages 453-475.

[15] POTENTIAL FIELD METHODS AND THEIR INHERENT PPROACHES FOR PATH PLANNING. Sabudin E. N, Omar. R and Che Ku Melor C. K. A. N. H.

[16] H. Seraji, "Traversability index: a new concept for planetary rovers," Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C), 1999, pp. 2006-2013 vol.3, doi: 10.1109/ROBOT.1999.770402.

[17] A. Saffiotti: "The use of fuzzy logic in autonomous robot navigation", Journal of Soft Computing, vol. 1, no. 4, 1997.

[18] W. Blochl: "Fuzzy control in real-time for vision quided autonomous mobile robots", Proc. Austrian Conf. on Fuzzy Logic in AI, pp. 114-125, 1993.

[19] D. B. Gennery: "Traversability analysis and path planning for a planetary rover", To appear in Journal of Autonomous Robots, 1999.

[20] H. Seraji, "Traversability indices for muti-scale terrain assessment".

[21] LS. Kwoen: "Extracting topographic terrain features from elevation maps", CGGIP: Image Understanding, vol 59, no. 2, pp.171-182, 1994.

[22] Lumelsky, Stepanov 1986.

[23] Kamon, Rivlin 1997.

[24] Ng. Bräunl 2007.

[25] B. Zhao, P. Ai and J. Han, "Study on the control method of NURBS curve quality for computer aided industrial design," 2012 7th International Conference on Computer Science & Education (ICCSE), 2012, pp. 658-661,

doi: 10.1109/ICCSE.2012.6295160.

[26] X. Liu, "Geometric Features Modification of NURBS Curves via Energy Optimization," 2009 First International Workshop on Education Technology and Computer Science, 2009, pp. 929-932, doi: 10.1109/ETCS.2009.211.

[27] H. Chen, C. Guo, Z. Wang, T. Wen, Z. Zeng and Z. Lin, "The trajectory planning system for spraying robot based on k-means clustering and NURBS curve optimization," IECON 2020 The 46th Annual Conference of the IEEE Industrial Electronics Society, 2020, pp. 5356-5361, doi: 10.1109/IECON43393.2020.9255172.

[28] Piegl L, Tiller W. The NURBS book[M]. 2nd, Springer, 1997.

[29] “Computer Graphics with Open GL”, Hearn Baker Carithers, Fourth Edition, 2014

Appendix A

Obstacle Detection Algorithm

Link to my GitHub account: GitHub account
Link to the NURBS path planner on GitHub: NURBS path planner algorithm

Code A.1: Obstacle Detection Algorithm

```
// *****************************
//
// Obstacle detection algorithm (C++)
//
// @ Copyright 2022, Gabriel Lucian Palcau
//
// *****************************

#include <opencv2/core/core.hpp>
#include <opencv2/highgui/highgui.hpp>
#include <opencv2/imgproc.hpp>
#include <iostream>
#include <fstream>

using namespace cv;
using namespace std;

const string pathFile[] = {"C:/Users/palca/Desktop/FileProgrammiTesi/
                Obstacle_Position.txt"};
const string path = "C:/Users/palca/Desktop/FileProgrammiTesi/Ostacoli1.bmp
                ";

int heightInitial = 0;
int widthInitial = 0;
const int heightTarget = 600;
const int widthTarget = 1000;
```

```cpp

void getContours(const Mat& imgCanny, const Mat& img, const Mat& img_copy);

int main()
{
    Mat img, imgCanny, imgDil, imgMod, imgGrid;
    img = imread(path, IMREAD_COLOR);

    // Check for failure
    if (img.empty())
    {
        cout << "Could not open or find the image" << endl;
        cin.get(); //wait for any key press
        return -1;
    }
    resize(img, img, Size(600,400), INTER_LINEAR);
    heightInitial = img.rows;
    widthInitial = img.cols;

    imshow("Original Image", img);

    img.copyTo(imgGrid);
    cvtColor(img, imgMod, COLOR_BGR2GRAY);

    GaussianBlur(imgMod, imgMod, Size(5, 5), 25, 25);
    Canny(imgMod, imgCanny, 50, 160);

    Mat kernel = getStructuringElement(MORPH_RECT, Size(3, 3));
    dilate(imgCanny, imgDil, kernel);

    getContours(imgDil, img, imgGrid);

    imwrite("C:/Users/palca/Desktop/FileImmaginiTesi/immagineOBS.png", img);

    imshow("Detected Obstacles", img);
    imshow("GaussianBlur Image", imgMod);
    imshow("Canny Detection", imgCanny);
    imshow("DilateCanny", imgDil);
    imshow("GridMap 0/1 Obstacle/Non-Obstacle", imgGrid);
    waitKey(0);

    return 0;
}

void getContours(const Mat& imgCanny, const Mat& img, const Mat& img_copy) {

    vector<vector<Point>> contours;
    vector<Vec4i> hierarchy;

    findContours(imgCanny, contours, hierarchy, RETR_EXTERNAL,
                 CHAIN_APPROX_NONE);
    drawContours(img, contours, -1, Scalar(0, 255, 0), 2);

    ofstream File;
    File.open(pathFile[0], ios::out);

    //Fill contours
    vector< vector<Point> > hull(contours.size());

    for (int i = 0; i < contours.size(); i++)
        convexHull(Mat(contours[i]), hull[i], false);

    fillPoly(img_copy, hull, Scalar(0, 0, 0));

    //Center computation
    vector<Point> centers;
    for (int i = 0; i < contours.size(); i++) {
        Moments M = moments(contours[i]);
        Point center(M.m10 / M.m00, M.m01 / M.m00);
        centers.push_back(center);
        circle(img, centers[i], 1, CV_RGB(255, 0, 0), 5);
    }

    //Max distance from center
    float max_dist = 0.0;
    float max_dist1 = 0.0;
    float dist = 0.0;
    float dist1 = 0.0;

    vector<Point> perimeter;
    for (int i = 0; i < contours.size(); i++) {
        perimeter = contours[i];
        int x = 0;
        int y = 0;
        max_dist = 0.0;

        for (int j = 0; j < size(perimeter); j++) {
            x = perimeter[j].x;
            y = perimeter[j].y;

            dist = sqrt(pow(x - centers[i].x, 2) + pow(y - centers[i].y, 2)
                 );
            dist1 = sqrt(pow(x * widthTarget / widthInitial - centers[i].x
                 * widthTarget / widthInitial, 2) + pow(y * heightTarget /
                 heightInitial - centers[i].y * heightTarget / heightInitial
                 , 2));
            if (dist > max_dist) {
                max_dist = dist;
                max_dist1 = dist1;
            }
        }

        circle(img, centers[i], max_dist, CV_RGB(0, 0, 255), 2);
        File << (int)ceil(centers[i].x * widthTarget / widthInitial) << " "
                  << (int)ceil((heightInitial - centers[i].y) * heightTarget
                  / heightInitial) << " " << (int)ceil(max_dist1) << endl;

    }

    File.close();
}
```

Appendix B

NURBS path planner Algorithm

Link to my GitHub account: GitHub account
Link to the NURBS path planner on GitHub: NURBS path planner algorithm

Code B.1: NURBS path planner Algorithm

```
// *****************************
//
// NURBS path planner algorithm (C++)
//
// @ Copyright 2022, Gabriel Lucian Palcau
//
// *****************************

#include <iostream>
#include "generic_curve.hpp"
#include "curve.hpp"
#include "circular_arc.hpp"
#include <vector>
#include "sisl.h"
#include "straight_line.hpp"
#include <fstream>

#ifdef _WIN32
#include <GL/glut.h>
#elif defined(_APPLE_)
#endif

#define CURVE_EVALUATIONS 500
using namespace std;

bool flag3d = false;
```

```
static int wayPointsFlag = 0;
int cntFile = 0;

// -------------------Global Variable -------------------

shared_ptr<GenericCurve> genericCurve = nullptr;
vector<Eigen::Vector3d> ctrlPoints{};

vector<shared_ptr<CircularArc>> obs;
vector<shared_ptr<CircularArc>> obs_external;
vector<Eigen::Vector3d> obs_points{};

vector<Eigen::Vector3d> pathRobot{};
vector<Eigen::Vector3d> wayPoints{};

vector<Eigen::Vector3d> pathRobotGlobal{};

int RobotPosition = 0;
int initSizePulse = 5;
int statePulse = 1;

static Eigen::Vector3d START{0,0,0};
static Eigen::Vector3d GOAL{0,0,0};

// -- OpenGL drawing prototype functions --
void DisplayCtrPoints();
void init2d();
void DrawSinglePoints(const vector<Eigen::Vector3d>& points);
void DrawObs(const shared_ptr<CircularArc>& ob, const bool flag);
void Draw2dNURBS();
void DrawpathRobotGlobal();

// -- Prototype Function for Manimulation of the SISL curve --

void UpdateNurbs(int& degree, vector<double>& knots, const vector<double>&
                                 weights);
Eigen::Vector4d firstIntersection(vector<Eigen::Vector3d>& intersections,
                                 const shared_ptr<GenericCurve>& nurbs,
                                 const bool flag);
int FindObstacleFromPoint(const shared_ptr<CircularArc>& pc1, const Eigen::
                                 Vector3d& point);
int UpdateParameterOrder(vector<double>& paramOrder);
double UpdateParameterDistance(const Eigen::Vector3d& Point, const
                                 shared_ptr<GenericCurve>& nurbs);
Eigen::Vector3d FindCenterMedian(const shared_ptr<CircularArc>& C1, const
                                 shared_ptr<CircularArc>& C2);
```

```
vector<Eigen::Vector3d> ExternalIntersection(const int pos_first, const
                                    vector<int>& VecCenterIntersection, const
                                    Eigen::Vector3d& Last, const Eigen::
                                    Vector3d& LastL, const int cnt, const int
                                    Median, const bool Median1);
void printNurbsInformation(const vector<double>& knots);
void pathRobotCalculation();
int EvaluatePointInObstacle(const Eigen::Vector3d& point);
void ExternalRadiusOpt(vector<int>& radius);
void CalculateWayPoints(vector<Eigen::Vector3d>& wayPointsClosest);
void timer(int);
void timer1(int);
void timer2(int);
void display();
int pathgeneration();





// --------------- MAIN ------------------

int main(int argc, char** argv) {

    //Acquiring Start and Goal position
    cout << "******* Insert Start and Goal position *******" << endl;
    cout << "Insert Start position, x and y: " << endl;
    cin >> START.x();
    cin >> START.y();
    cout << "Insert Goal position, x and y: " << endl;
    cin >> GOAL.x();
    cin >> GOAL.y();

    //Trajectory printed on OpenGL
    int x = pathgeneration();
    if (x == -10){
        cerr << "Error, too complex environment";
        return -1;
    }

    glutInit(&argc, argv);
    glutInitWindowSize(1000, 600);
    glutInitWindowPosition(175, 40);
    glutCreateWindow("Trajectory");
    init2d();

    glutDisplayFunc(display);
    glutTimerFunc(0,timer,0);
    glutTimerFunc(0,timer1,0);
    glutTimerFunc(0,timer2,0);
    glDisable(GL_TEXTURE_2D);
    glutMainLoop();

    return 0;
}

// ------ OpenGL drawing functions -------

void DisplayCtrPoints()
{
    glPointSize(5.0);
    glColor3f(0.0f, 0.0f, 1.0f);

    glBegin(GL_POINTS);
    for (const auto& coord : ctrlPoints)
        glVertex2f((GLfloat)coord[0], (GLfloat)coord[1]);
    glEnd();
}

void init2d()
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 1000, 0.0, 600);
    glClearColor(1.0f, 1.0f, 1.0f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);
}

void DrawSinglePoints(const vector<Eigen::Vector3d>& points)
{
    glPointSize(5.0);
    glColor3f(0.0f, 0.0f, 0.0f);

    glBegin(GL_POINTS);
    for (const auto& i : points)
        glVertex2f((float)i[0], (float)i[1]);
    glEnd();
}

void DrawObs(const shared_ptr<CircularArc>& ob, const bool flag)
{
    int left = 0;
    auto* discrete = new double[3 * CURVE_EVALUATIONS];

    for (int j = 0; j < CURVE_EVALUATIONS; j++)
    {
        double t = ob->StartParameter_s() + (ob->EndParameter_s() - ob->
                                 StartParameter_s()) * j / (
                                 CURVE_EVALUATIONS - 1.0);
        int stat;
        s1227(ob->CurvePtr(), 0, t, &left, discrete + 3 * j, &stat);

        if (stat != 0)
            cerr << "s1227 returned status: " << stat << endl;
    }

    if (flag3d) {
        glLineWidth(2.0);
        glBegin(GL_LINE_STRIP);
        for (int j = 0; j < CURVE_EVALUATIONS; j++) {
            glColor3f(0.0f, 0.5f, 1.0f);
            glVertex3f((float)discrete[0 + (3 * j)] * 10 / 1000, (float)
                                 discrete[1 + (3 * j)] * 6 / 600,
                (float)discrete[2 + (3 * j)] * 3 / 100);
        }
        glEnd();
        glFlush();
    }
    else {
        glLineWidth(1.0);
        glBegin(GL_LINE_STRIP);
        for (int j = 0; j < CURVE_EVALUATIONS; j++) {
            glColor3f(0.0f, 0.5f, 1.0f);

            if (flag)
                glVertex2f((float)ob->CentrePoint().x(), (float)ob->
                                 CentrePoint().y());
            glVertex2f((float)discrete[0 + (3 * j)], (float)discrete[1 + (3
                                  * j)]);
        }
        glEnd();
    }
    delete[] discrete;
}

void Draw2dNURBS()
{
    glLineWidth(2.0);
    glBegin(GL_LINE_STRIP);
    glColor3f(1.0f, 0.0f, 0.0f);
    for (const auto& j : pathRobot)
        glVertex2f((float)j.x(), (float)j.y());

    glEnd();
}
```

```

void DrawpathRobotGlobal()
{
    glLineWidth(2.0);
    glColor3f(0.0f, 0.0f, 1.0f);
    glBegin(GL_LINE_STRIP);
    for (const auto& ele : pathRobotGlobal)
        glVertex2f((GLfloat)ele.x(), (GLfloat)ele.y());

    glEnd();
}

// ----- Function for Manimulation of the SISL curve -----

void UpdateNurbs(int& degree, vector<double>& knots, const vector<double>& weights)
{
    unsigned int nCtrl = ctrlPoints.size();
    if (ctrlPoints.size() == 3)
        degree = 2;
    else if (ctrlPoints.size() >= 4)
        degree = 3;

    unsigned int n_knots = degree + 1 + nCtrl; //Mathematical Formula for NURBS

    int value = 0;
    knots.clear();
    for (int i = 0; i < n_knots; i++) {
        if (i < degree + 1) {
            knots.push_back(value);
        }
        else {
            if ((n_knots - i) > degree + 1) {
                value++;
                knots.push_back(value);
            }
            else {
                value++;
                knots.push_back(value);
                value--;
            }
        }
    }
}

Eigen::Vector4d firstIntersection(vector<Eigen::Vector3d>& intersections, const shared_ptr<GenericCurve>& nurbs, const bool flag)
```

```
{
    double min = 9999;
    double eps = 5;
    Eigen::Vector3d temp;
    Eigen::Vector4d temp1 = { 0,0,0,0 };
    shared_ptr<StraightLine> ptr_Line = nullptr;

    for (int j = 0; j < 10 * CURVE_EVALUATIONS; j++) {

        double t = nurbs->StartParameter_s() + (nurbs->EndParameter_s() -
                                 nurbs->StartParameter_s()) * j / (10 *
                                 CURVE_EVALUATIONS - 1.0);
        nurbs->FromAbsSislToPos(t, temp);

        if (!flag) {
            for (const auto& ele : intersections) {
                ptr_Line = make_shared<StraightLine>(temp, ele, 3, 2);
                if (ptr_Line->Length() <= eps) {
                    auto elef = ele;
                    auto it = remove(intersections.begin(), intersections.
                                 end(), ele);
                    intersections.erase(intersections.end());
                    return { elef.x(), elef.y(), elef.z(), t };
                }
            }
        }
        else {

            tuple<double, double> closest = nurbs->FindClosestPoint(
                                 intersections[0]);
            auto abscissa_m1 = (double)(get<0>(closest));
            auto ClosestPoint = nurbs->At(double(abscissa_m1));
            ptr_Line = make_shared<StraightLine>(ClosestPoint, temp, 3, 2);

            if (ptr_Line->Length() <= min) {
                min = ptr_Line->Length();
                temp1 = { ClosestPoint.x(),ClosestPoint.y(),ClosestPoint.z
                                 (),t };
            }
        }
    }
    return temp1;
}

int FindObstacleFromPoint(const shared_ptr<CircularArc>& pc1, const Eigen::
                                 Vector3d& point)
{
    bool inside = false;
    Eigen::Vector3d temp = { point.x(),point.y(),point.z() };
    shared_ptr<StraightLine> line = nullptr;
    line = make_shared<StraightLine>(pc1->CentrePoint(), temp, 3, 2);
    shared_ptr<StraightLine> line1 = nullptr;
    line1 = make_shared<StraightLine>(pc1->CentrePoint(), pc1->StartPoint()
                                      , 3, 2);

    if (line->Length() <= (line1->Length() + 1))
        inside = true;
    return inside;
}

int UpdateParameterOrder(vector<double>& paramOrder)
{
    auto itP = paramOrder.begin();
    bool flagP = false;
    int index = -1;
    double x = -1;

    if (paramOrder.size() == 1)
        index = 1;
    else {
        x = paramOrder.back();
        paramOrder.pop_back();
        for (int p = 0; p < paramOrder.size(); p++) {
            if (x <= paramOrder.at(p)) {
                flagP = true;
                index = p + 1;
                paramOrder.insert(itP + p, x);
                break;
            }
        }
        if (!flagP) {
            paramOrder.push_back(x);
            index = (int)paramOrder.size();
        }
    }
    return index;
}

double UpdateParameterDistance(const Eigen::Vector3d& Point, const
                                      shared_ptr<GenericCurve>& nurbs)
{
    float eps = 1;
    Eigen::Vector3d temp;
    Eigen::Vector3d cPoint = Point;
    shared_ptr<StraightLine> Line = nullptr;

    tuple<double, double> closest = nurbs->FindClosestPoint(cPoint);
    auto abscissa_m1 = (double)(get<0>(closest));
    auto ClosestPoint = nurbs->At(double(abscissa_m1));

    for (int j = 0; j < 5 * CURVE_EVALUATIONS; j++) {

        double t = nurbs->StartParameter_s() + (nurbs->EndParameter_s() -
                                 nurbs->StartParameter_s()) * j / (5 *
                                 CURVE_EVALUATIONS - 1.0);
        nurbs->FromAbsSislToPos(t, temp);
        Line = make_shared<StraightLine>(ClosestPoint, temp, 3, 2);

        if (Line->Length() <= eps)
            return t;
    }
    return -1;
}

Eigen::Vector3d FindCenterMedian(const shared_ptr<CircularArc>& C1, const
                                 shared_ptr<CircularArc>& C2)
{
    shared_ptr<StraightLine> ptr_Line = nullptr;
    ptr_Line = make_shared<StraightLine>(C1->CentrePoint(), C2->CentrePoint
                                 (), 3, 2);

    vector<Eigen::Vector3d> temp{};
    vector<Eigen::Vector3d> intersections{};

    temp = ptr_Line->Intersection(C1);
    intersections.push_back(temp[0]);

    temp = ptr_Line->Intersection(C2);
    intersections.push_back(temp[0]);

    Eigen::Vector3d Medium{};
    Medium.x() = (intersections[0].x() + intersections[1].x()) / 2;
    Medium.y() = (intersections[0].y() + intersections[1].y()) / 2;
    Medium.z() = 0;
    return Medium;
}

vector<Eigen::Vector3d> ExternalIntersection(const int pos_first, const
                                 vector<int>& VecCenterIntersection, const
                                 Eigen::Vector3d& Last, const Eigen::
                                 Vector3d& LastL, const int cnt, const int
                                 Median, const bool Median1)
{
    const vector<shared_ptr<CircularArc>>& circle = obs_external;
    vector<Eigen::Vector3d> Intersections{};

    if (cnt == 2 && !Median1) {
        for (const int& ele : VecCenterIntersection) {
            auto IntersectionsTemp = circle.at(pos_first)->Intersection(circle.at(ele));
            for (const auto& ele1 : IntersectionsTemp)
                Intersections.push_back(ele1);
        }
    }
    else if (cnt == 1 && Median1) {

        auto IntersectionsTemp = circle.at(pos_first)->Intersection(circle.at(Median));
        for (const auto& ele1 : IntersectionsTemp)
            Intersections.push_back(ele1);
        IntersectionsTemp = circle.at(VecCenterIntersection.at(0))->Intersection(circle.at(VecCenterIntersection.at(1)));
        for (const auto& ele1 : IntersectionsTemp)
            Intersections.push_back(ele1);
    }
    else {
        Intersections = circle[pos_first]->Intersection(circle[Median]);
    }

    shared_ptr<StraightLine> Line = nullptr;
    double min = 9999;
    double max = 9999;
    Eigen::Vector3d MinPoint;
    Eigen::Vector3d MaxPoint;

    for (const auto& ele : Intersections) {
        Line = make_shared<StraightLine>(Last, ele, 3, 2);
        if (Line->Length() <= min) {
            min = Line->Length();
            MinPoint = ele;
        }

        Line = make_shared<StraightLine>(LastL, ele, 3, 2);
        if (Line->Length() <= max) {
            max = Line->Length();
            MaxPoint = ele;
        }
    }

    vector<Eigen::Vector3d> TwoPoints{};
    TwoPoints.push_back(MinPoint);
    TwoPoints.push_back(MaxPoint);
    return TwoPoints;
}

```

```
void printNurbsInformation(const vector<double>& knots)
{
    cout << "Generated NURBS:" << endl;
    cout << "----- Degree: " << genericCurve->Degree() << endl;
    cout << "----- Control Points: " << endl;
    cout << "----- ----- [ ";
    for (const auto& ele : ctrlPoints)
        cout << ele.x() << " ";
    cout << endl << "              ";
    for (const auto& ele : ctrlPoints)
        cout << ele.y() << " ";
    cout << "]" << endl;
    cout << "----- Knot Vector: " << endl;
    cout << "----- ----- [ ";
    for (const auto& ele : knots)
        cout << ele << " ";
    cout << "]" << endl;
}

void pathRobotCalculation()
{
    pathRobot.clear();

    int left = 0;
    auto* discrete = new double[3 * 10 * CURVE_EVALUATIONS];
    double len = 0.0;
    double lenTemp = 0.0;
    shared_ptr<StraightLine> line = nullptr;
    Eigen::Vector3d Apoint;
    Eigen::Vector3d Bpoint;

    for (int j = 0; j < 10 * CURVE_EVALUATIONS; j++) {
        double t = genericCurve->StartParameter_s() +
            (genericCurve->EndParameter_s() - genericCurve->
                                 StartParameter_s()) * j / (10 *
                                 CURVE_EVALUATIONS - 1.0);

        int stat;
        s1227(genericCurve->CurvePtr(), 0, t, &left, discrete + 3 * j, &
                                 stat);
        if (stat != 0)
            cerr << "s1227 returned status: " << stat << endl;

        if (j == 1) {
            Bpoint = { discrete[0 + (3 * j)], discrete[1 + (3 * j)], 0.0 };
            Apoint = { discrete[0 + (3 * (j - 1))], discrete[1 + (3 * (j -
                                 1))], 0.0 };
            pathRobot.push_back(Apoint);
            pathRobot.push_back(Bpoint);
            len = 1;
        }

        if (j > 1)
        {
            Bpoint = { discrete[0 + (3 * j)], discrete[1 + (3 * j)], 0.0 };
            line = make_shared<StraightLine>(pathRobot.back(), Bpoint, 3,
                                 2);
            lenTemp = line->Length();
            if (lenTemp >= (len - len / 8)) {
                pathRobot.push_back(Bpoint);
            }
        }
    }
}

int EvaluatePointInObstacle(const Eigen::Vector3d& point)
{
    int ObsToPoint = 0;
    for (int z = 0; z < obs.size(); z++) {
        if (FindObstacleFromPoint(obs.at(z), point)) {
            ObsToPoint = z;
            break;
        }
    }
}

void ExternalRadiusOpt(vector<int>& radius)
{
    vector<Eigen::Vector3d> intersections{};
    int times = 0;
    Eigen::Vector3d axis = { 0,0,1 };
    bool limit = false;

    for (int i = 0; i < obs_external.size(); i++) {
        times = 0;
        limit = false;
        for (const auto& ele : obs) {
            intersections = obs_external.at(i)->Intersection(ele);
            if (!intersections.empty())
                times++;
        }
        if (times > 2) {
            radius.at(i) = radius.at(i) - 1;
            if (radius.at(i) <= (obs.at(i)->StartPoint().x() - obs.at(i)->
                                 CentrePoint().x()) + 3) {
                radius.at(i) = (obs.at(i)->StartPoint().x() - obs.at(i)->
                                 CentrePoint().x()) + 5;
                limit = true;
            }
            obs_external.at(i) = make_shared<CircularArc>(2 * 3.1415926,
                                 axis, obs_points.at(i) + Eigen::Vector3d{ (
                                 double)radius.at(i), 0.0, 0.0 }, obs_points
                                 .at(i), 3, 3);
            if (!limit)
                i--;
        }
    }
}

void CalculateWayPoints(vector<Eigen::Vector3d>& wayPointsClosest)
{
    shared_ptr<StraightLine> line = nullptr;
    wayPoints.clear();

    for (auto& j : wayPointsClosest) {
        for (int i = 0; i < pathRobot.size(); i++) {
            line = make_shared<StraightLine>(j, pathRobot.at(i), 3, 2);
            if (line->Length() < 10) {
                wayPoints.push_back(pathRobot.at(i + 30));
                break;
            }
        }
    }
}

void timer(int)
{
    glutPostRedisplay();
    glutTimerFunc(1000 / 200, timer, 0);

    if (wayPointsFlag == 0) {
        for (const auto& ele : wayPoints) {
            if (ele == pathRobot[RobotPosition]) {
                wayPointsFlag = 1;
                pathRobotGlobal.push_back(pathRobot.at(RobotPosition));
            }
        }
    }

    if (wayPointsFlag == 0) {
        pathRobotGlobal.push_back(pathRobot.at(RobotPosition));
        RobotPosition++;
    }

    if (RobotPosition >= pathRobot.size())
        RobotPosition = (int)pathRobot.size() - 1;
}
```

```

void timer1(int)
{
    glutPostRedisplay();
    glutTimerFunc(1000 / 10, timer1, 0);
    switch (statePulse)
    {
    case 1:
        if (initSizePulse <= 10)
            initSizePulse++;
        else
            statePulse = 2;
        break;
    case 2:
        if (initSizePulse >= 5)
            initSizePulse--;
        else
            statePulse = 1;
        break;
    default:
        cerr << "Error in switching state Pulse of the robot position" << endl;
    }
}

void timer2(int)
{
    glutTimerFunc(3000, timer2, 0);

    if (wayPointsFlag == 1)
    {
        int x = pathgeneration();
        wayPointsFlag = 0;
        if (x == -10) {
            cerr << "Error too complex environment";
        }
    }
}

void display()
{
    glClear(GL_COLOR_BUFFER_BIT);

    for (const auto& ob : obs) {
        DrawObs(ob, true);
    }
    for (const auto& ob : obs_external) {
        DrawObs(ob, false);
    }

    Draw2dNURBS();
    DisplayCtrPoints();
    DrawSinglePoints(wayPoints);
    DrawpathRobotGlobal();

    glPointSize((float)initSizePulse);
    glColor3f(1.0f, 0.0f, 0.0f);
    glBegin(GL_POINTS);
    glVertex2f((GLfloat)pathRobot[RobotPosition].x(), (GLfloat)pathRobot[RobotPosition].y());
    glEnd();

    glutSwapBuffers();
}

int pathgeneration()
{

    cntFile++;

    // Defining the path of the obstacles txt
    string pathObstacleTxT;
    try {
        if (cntFile >= 1)
            pathObstacleTxT = "C:/Users/palca/Desktop/FileProgrammiTesi/Obstacle_Position.txt";
        else if (cntFile > 1)
            pathObstacleTxT = "C:/Users/palca/Desktop/FileProgrammiTesi/Obstacle_Position1.txt";
    }
    catch (...) {
        cerr << "Error while trying to read the path where the obstacle position are written!" << endl;
        return -1;
    }

    //Pre-Processing for saving the obstacle position from file (control Points)
    static vector<int> obs_radius{};
    static vector<Eigen::Vector3d> ctrlPoints_All;

    if (cntFile == 1)
        ctrlPoints_All.emplace_back(START);

    bool flagDynamicPath = false;
    bool flagSameObs = false;
    int x, y, r;
    Eigen::Vector3d ctrlTemp{};
    stringstream ss;
    string line;
    ifstream dataIN;

    dataIN.open(pathObstacleTxT);
    if (dataIN.is_open())
    {
        while (getline(dataIN, line))
        {
            flagSameObs = false;
            ss.str(line);
            ss >> x;
            ss >> y;
            ss >> r;

            if (r > 5) {

                if (cntFile == 1) {
                    ctrlPoints_All.emplace_back(x, y, 0);
                    obs_radius.push_back(r);
                }
                else {
                    for (const auto& ele : ctrlPoints_All) {
                        if ((x == (int)ele.x()) && (y == (int)ele.y())) {
                            flagSameObs = true;
                        }
                    }
                    if (!flagSameObs) {
                        flagDynamicPath = true;
                        ctrlPoints_All.insert(ctrlPoints_All.begin() + 1,
                                 Eigen::Vector3d{ (double)x,(double)y,0 });
                        obs_radius.insert(obs_radius.begin() + 1, r);
                    }
                }
            }
            ss.clear();
        }
        dataIN.close();
    }
    else {
        cerr << "Unable to open file txt where the position of the
                                 obstacles are saved!" << endl;
        return -2;
    }

    if (cntFile == 1)
        ctrlPoints_All.emplace_back(GOAL);

    // If new obstacles are not detected, return to the original path
    if (!flagDynamicPath && cntFile > 1) {
        wayPoints.erase(wayPoints.begin());
        return 0;
    }

    // If new obstacles are found, go to dynamic path
    if (wayPointsFlag == 1 && cntFile > 1)
    {
        ctrlPoints_All.at(0) = wayPoints.at(0);
        wayPoints.erase(wayPoints.begin());
        RobotPosition = 0;

        obs.clear();
        obs_external.clear();
        obs_points.clear();
    }

    const int nAll = (int)ctrlPoints_All.size();
    const int nObs = nAll - 2;
    auto itAll = ctrlPoints_All.begin();

    // Defining the FISRT parameters for the NURBS, from start to goal, it will be replaced later
    const int dimension = 3;
    int degree = 1;

    ctrlPoints.clear();
    ctrlPoints.push_back(*itAll);
    ctrlPoints.push_back(*(ctrlPoints_All.end() - 1));
    auto it = ctrlPoints.begin();

    vector<double> weights{};
    weights.push_back(1.0);
    weights.push_back(1.0);
    auto it1 = weights.begin();

    vector<double> knots{};
    const vector<double> coeff{};

    // ---- Defining only the obstacles control points ----

    for (int i = 1; i < ctrlPoints_All.size() - 1; i++)
        obs_points.emplace_back(ctrlPoints_All.at(i));

    // ---- Defining the SISL curve for obstacle and external_obstacle circle ----

    const int safetyDim = 15;
    vector<int> obs_ext_radius{};

    for (int i = 0; i < nObs; i++)
        obs_ext_radius.push_back(obs_radius.at(i) + safetyDim);

    const int dimensionObs = 3;
    const int orderObs = 3;
    const Eigen::Vector3d axis = { 0.0, 0.0, 1.0 };

    for (int i = 0; i < nObs; i++) {
        obs.push_back(make_shared<CircularArc>(2 * 3.1415926, axis, obs_points.at(i) + Eigen::Vector3d{ (double)obs_radius.at(i), 0.0, 0.0 }, obs_points.at(i), dimensionObs, orderObs));
        obs_external.push_back(make_shared<CircularArc>(2 * 3.1415926, axis, obs_points.at(i) + Eigen::Vector3d{ (double)obs_ext_radius.at(i), 0.0, 0.0 }, obs_points.at(i), dimensionObs, orderObs));
    }

    // Function for External radius optimization, maximum three obstacles intersections
    ExternalRadiusOpt(obs_ext_radius);


    // ----Definition of variables ----

    bool flagEnd = false;

    vector<Eigen::Vector3d> intersections_All = {};
    vector<Eigen::Vector3d> intersections{};
    vector<int> position{};
    vector<double> paramOrder;
    vector<int> visited;
    vector<Eigen::Vector3d> freeControlObs;

    Eigen::Vector3d ClosestPoint = {};
    Eigen::Vector3d second{};
    Eigen::Vector3d first{};

    int c = 0;
    int pos_first = -1;
    int previous = -2;
    int index = -1;
    int VectorAdder = 0;

    shared_ptr<StraightLine> ptr_straightLine = nullptr;
    vector<Eigen::Vector3d> wayPointsClosest{};

    // Cicle for the NURBS path generation

    for (int k = 0; k < 20; k++) {

        cout << endl << endl;
        cout << "Evaluating intersection of obstacle: iteration n." << k <<
                                  endl;

        UpdateNurbs(degree, knots, weights);
        it = ctrlPoints.begin();
        it1 = weights.begin();

        genericCurve = make_shared<GenericCurve>(degree, knots, ctrlPoints,
                                  weights, coeff, dimension, degree + 1);

        intersections_All.clear();
        position.clear();
        pos_first = -1;

        //---- Calculating all the NURBS points intersection ----
        for (int j = 0; j < obs_points.size(); j++) {

            intersections.clear();
            intersections = genericCurve->Intersection(obs[j]);

            if (!intersections.empty())
                position.push_back(j);

            for (const auto& ele : intersections)
                intersections_All.push_back(ele);
        }

        if (!intersections_All.empty()) {

            //Find the first intersection, the obstacle, and the exit point
                                  from the obstacle
            Eigen::Vector4d temp1 = firstIntersection(intersections_All,
                                 genericCurve, false);
            first = { temp1.x(),temp1.y(), temp1.z() };
            paramOrder.push_back(temp1.w());
            auto itP = paramOrder.begin();
            index = UpdateParameterOrder(paramOrder);

            for (int w = 0; w < obs_points.size(); w++) {
                if (FindObstacleFromPoint(obs.at(w), first)) {
                    pos_first = w;
                    break;
                }
            }

            temp1 = firstIntersection(intersections_All, genericCurve,
                                     false);
            second = { temp1.x(),temp1.y(), temp1.z() };

            if (!FindObstacleFromPoint(obs.at(pos_first), second)) {
                second = {};
                second = first;
            }

            // Evaluate if the first obstacle (external circle) intersects
                                     others obstacles
            auto itV = find(visited.begin(), visited.end(), pos_first);
            vector<int> VecCenterIntersection{};
            for (int q = 0; q < obs_points.size(); q++) {
                if (pos_first != q) {
                    auto CenterIntersection = obs_external[pos_first]->
                                     Intersection(obs[q]);
                    if (!CenterIntersection.empty())
                        VecCenterIntersection.push_back(q);
                }
            }

            if (itV != visited.end()) {

                // --- We already visited this obstacle ---
                cout << "Obstacle already visited.." << endl;

                ptr_straightLine = nullptr;

                // Calculation of the point to be added on the external
                                     radius for avoiding the obstacle
                double x_abs = abs(first.x() - obs_points[pos_first].x());
                double y_abs = abs(first.y() - obs_points[pos_first].y());
                double kk = obs_ext_radius[pos_first] / min(x_abs, y_abs);

                if (first.y() <= obs_points[pos_first].y()) {
                    if (first.x() <= obs_points[pos_first].x()) {
                        ptr_straightLine = make_shared<StraightLine>(
                                     obs_points[pos_first], obs_points[pos_first
                                     ] + Eigen::Vector3d{ -kk * x_abs, -kk *
                                     y_abs, 0.0 }, 3, 2);
                    }
                    else {
                        ptr_straightLine = make_shared<StraightLine>(
                                     obs_points[pos_first], obs_points[pos_first
                                     ] + Eigen::Vector3d{ +kk * x_abs, -kk *
                                     y_abs, 0.0 }, 3, 2);
                    }
                }
                else {
                    if (first.x() <= obs_points[pos_first].x()) {
                        ptr_straightLine = make_shared<StraightLine>(obs_points[pos_first], obs_points[pos_first] + Eigen::Vector3d{ -kk * x_abs,kk * y_abs,0.0 }, 3, 2);
                    }
                    else {
                        ptr_straightLine = make_shared<StraightLine>(obs_points[pos_first], obs_points[pos_first] + Eigen::Vector3d{ kk * x_abs,kk * y_abs,0.0 }, 3, 2);
                    }
                }

                intersections.clear();
                intersections = ptr_straightLine->Intersection(obs_external[pos_first]);

                ctrlPoints.insert(it + index, intersections.at(0));
                weights.insert(it1 + index, 1.4);
                UpdateNurbs(degree, knots, weights);
                genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff, dimension, degree + 1);
                for (int s = 0; s < paramOrder.size(); s++)
                    paramOrder[s] = UpdateParameterDistance(ctrlPoints[s + 1], genericCurve);

                wayPointsClosest.pop_back();
                tuple<double, double> closest = genericCurve->FindClosestPoint(ctrlPoints[ctrlPoints.size() - 2]);
                auto abscissa_m1 = (double)(get<0>(closest));
                ClosestPoint = genericCurve->At(double(abscissa_m1));
                wayPointsClosest.push_back(ClosestPoint);

                printNurbsInformation(knots);
                continue;

            }
            else {

                //---- First time we visit this obstacle ----
                cout << "Obstacle visited for the first time.. " << endl;

                visited.push_back(pos_first);

                ctrlPoints.insert(it + index, first);
                weights.insert(it1 + index, 0.8);
                UpdateNurbs(degree, knots, weights);
                genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff, dimension, degree + 1);
                for (int s = 0; s < paramOrder.size(); s++)
                    paramOrder[s] = UpdateParameterDistance(ctrlPoints[s + 1], genericCurve);

                it = ctrlPoints.begin();
                it1 = weights.begin();

                // Calculate external point to be addend to bring the NURBS away from the obstacle
                tuple<double, double> closest = genericCurve->FindClosestPoint(obs_points[pos_first]);
                auto abscissa_m1 = (double)(get<0>(closest));
                ClosestPoint = genericCurve->At(double(abscissa_m1));
                double x_abs = abs(ClosestPoint.x() - obs_points[pos_first].x());
                double y_abs = abs(ClosestPoint.y() - obs_points[pos_first].y());
                double kk = obs_ext_radius[pos_first] / min(x_abs, y_abs);
                kk = kk + 1;

                Eigen::Vector3d finalPoint1;

                if (x_abs < 0.001 || y_abs < 0.001) {
                    if (x_abs < 0.001) {
                        if (ClosestPoint.y() <= obs_points[pos_first].y())
                            finalPoint1 = obs_points[pos_first] + Eigen::Vector3d{ 0.0, -(double)obs_ext_radius.at(pos_first) - 10, 0.0 };
                        else
                            finalPoint1 = obs_points[pos_first] + Eigen::Vector3d{ 0.0, +(double)obs_ext_radius.at(pos_first) + 10, 0.0 };
                    }
                    if (y_abs < 0.001) {
                        if (ClosestPoint.x() <= obs_points[pos_first].x())
                            finalPoint1 =
                            obs_points[pos_first] + Eigen::Vector3d{ -(double)obs_ext_radius.at(pos_first) - 10, 0.0, 0.0 };
                        else
                            finalPoint1 =
                            obs_points[pos_first] + Eigen::Vector3d{ +(double)obs_ext_radius.at(pos_first) + 10, 0.0, 0.0 };
                    }
                }
                else {
                    if (ClosestPoint.y() <= obs_points[pos_first].y()) {
                        if (ClosestPoint.x() <= obs_points[pos_first].x())
                                 {
                            finalPoint1 = obs_points[pos_first] + Eigen::
                                 Vector3d{ -kk * x_abs, -kk * y_abs, 0.0 };
                        }
                        else {
                            finalPoint1 = obs_points[pos_first] + Eigen::
                                 Vector3d{ +kk * x_abs, -kk * y_abs, 0.0 };
                        }
                    }
                    else {
                        if (ClosestPoint.x() <= obs_points[pos_first].x())
                                 {
                            finalPoint1 = obs_points[pos_first] + Eigen::
                                 Vector3d{ -kk * x_abs, +kk * y_abs, 0.0 };
                        }
                        else {
                            finalPoint1 = obs_points[pos_first] + Eigen::
                                 Vector3d{ +kk * x_abs, +kk * y_abs, 0.0 };
                        }
                    }
                }

                ptr_straightLine = nullptr;
                ptr_straightLine = make_shared<StraightLine>(obs_points[
                                 pos_first], finalPoint1, 3, 2);
                freeControlObs = ptr_straightLine->Intersection(
                                 obs_external[pos_first]);

                // Evaluating in the freeControlObs is inside another
                                 obstacle
                if (EvaluatePointInObstacle(freeControlObs.at(0)) == 0) {

                    intersections.clear();
                    intersections.push_back(freeControlObs[0]);
                    temp1 = firstIntersection(intersections, genericCurve,
                                 true);
                    paramOrder.push_back(temp1.w());
                    intersections.clear();

                    index = UpdateParameterOrder(paramOrder);
                    ctrlPoints.insert(it + index, freeControlObs[0]);
                    weights.insert(it1 + index, 2);
                    UpdateNurbs(degree, knots, weights);
                    genericCurve = std::make_shared<GenericCurve>(degree,
                                 knots, ctrlPoints, weights, coeff,
                                 dimension, degree + 1);
                    for (int s = 0; s < paramOrder.size(); s++)
                        paramOrder[s] = UpdateParameterDistance(ctrlPoints[
                                 s + 1], genericCurve);

                    it = ctrlPoints.begin();
                    it1 = weights.begin();
                }

                // Evaluating if we have 2 or 3 near obstacles
                Eigen::Vector3d freeMediumPoint = {};
                int Median = -1;
                int cnt = 0;
                bool flagFreeMediumPoint = false;
                bool Median1 = false;

                if (!VecCenterIntersection.empty()) {
                    ptr_straightLine = nullptr;
                    double minParMedian = 999;
                    Median = -1;
                    for (const auto& ele : VecCenterIntersection) {
                        ptr_straightLine = make_shared<StraightLine>(
                                 obs_points[pos_first], obs_points[ele], 3,
                                 2);
                        intersections.clear();
                        intersections = genericCurve->Intersection(
                                 ptr_straightLine);
                        if (!intersections.empty()) {
                            cnt++;
                            flagFreeMediumPoint = true;
                            temp1 = firstIntersection(intersections,
                                 genericCurve, false);
                            if (temp1.w() <= minParMedian) {
                                Median = ele;
                                minParMedian = temp1.w();
                            }
                        }
                    }

                    if (VecCenterIntersection.size() > 1) {
                        ptr_straightLine = make_shared<StraightLine>(
                                 obs_points[VecCenterIntersection.at(0)],
                            obs_points[VecCenterIntersection.at(1)], 3, 2);
                        intersections.clear();
                        intersections = genericCurve->Intersection(
                                 ptr_straightLine);
                        temp1 = firstIntersection(intersections, genericCurve, false);
                        intersections.emplace_back(temp1.x(), temp1.y(), temp1.z());
                        auto intersections1to2 = obs.at(VecCenterIntersection.at(0))->Intersection(obs_external.at(VecCenterIntersection.at(1)));
                        auto intersections2to1 = obs_external.at(VecCenterIntersection.at(1))->Intersection(obs.at(VecCenterIntersection.at(0)));

                        if (!intersections.empty() && (temp1.w() >= minParMedian) && (!intersections1to2.empty() || !intersections2to1.empty()))
                            Median1 = true;
                    }
                }

                // Evaluating what control point need to be added in the case of 2 or 3 near obstacles
                if (flagFreeMediumPoint) {

                    freeMediumPoint = FindCenterMedian(obs[pos_first], obs[Median]);

                    if (EvaluatePointInObstacle(freeMediumPoint) != 0)
                        return -10;

                    Eigen::Vector3d Last{};
                    if (EvaluatePointInObstacle(freeControlObs[0]) == 0)
                        Last = ctrlPoints[ctrlPoints.size() - 3];
                    else
                        Last = ctrlPoints[ctrlPoints.size() - 2];

                    vector<Eigen::Vector3d> TwoPointsExternal{};
                    TwoPointsExternal = ExternalIntersection(pos_first, VecCenterIntersection, Last, ctrlPoints[ctrlPoints.size() - 1], cnt, Median, Median1);

                    vector<Eigen::Vector3d> TempVector{};
                    TempVector.push_back(TwoPointsExternal[0]);

                    shared_ptr<StraightLine> LineTemp = nullptr;
                    LineTemp = make_shared<StraightLine>(TwoPointsExternal[0], ctrlPoints[ctrlPoints.size() - 2], 3, 2);

                    if (EvaluatePointInObstacle(TwoPointsExternal[0]) == 0) {
                        if (LineTemp->Length() < 15) {
                            ctrlPoints.at(index) = TwoPointsExternal[0];
                            weights.at(index) = 6;
                            UpdateNurbs(degree, knots, weights);
                            genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff,
                                dimension, degree + 1);
                            it = ctrlPoints.begin();
                            it1 = weights.begin();
                        }
                        else {
                            auto temp4 = firstIntersection(TempVector, genericCurve, true);
                            paramOrder.push_back(temp4.w());

                            index = UpdateParameterOrder(paramOrder);
                            ctrlPoints.insert(it + index, TwoPointsExternal[0]);
                            weights.insert(it1 + index, 10);
                            UpdateNurbs(degree, knots, weights);
                            genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff,
                                dimension, degree + 1);
                            it = ctrlPoints.begin();
                            it1 = weights.begin();
                        }
                    }

                    for (int s = 0; s < paramOrder.size(); s++)
                        paramOrder[s] = UpdateParameterDistance(ctrlPoints[s + 1], genericCurve);

                    TempVector.clear();
                    TempVector.push_back(freeMediumPoint);
                    auto temp4 = firstIntersection(TempVector, genericCurve, true);
                    paramOrder.push_back(temp4.w());

                    index = UpdateParameterOrder(paramOrder);
                    ctrlPoints.insert(it + index, freeMediumPoint);
                    weights.insert(it1 + index, 6);
                    UpdateNurbs(degree, knots, weights);
                    genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff,
                                dimension, degree + 1);
                    it = ctrlPoints.begin();
                    it1 = weights.begin();

                    for (int s = 0; s < paramOrder.size(); s++)
                        paramOrder[s] = UpdateParameterDistance(ctrlPoints[
                                 s + 1], genericCurve);

                    if ((VecCenterIntersection.size() == 2) && (Median !=
                                 -1)) {
                        auto itR = remove(VecCenterIntersection.begin(),
                                 VecCenterIntersection.end(), Median);
                        VecCenterIntersection.pop_back();
                    }

                    // Evaluating if we have 3 near obstacle
                    if (cnt == 2 || Median1) {

                        Eigen::Vector3d Baricenter;
                        Baricenter.x() = (obs_points[pos_first].x() +
                                 obs_points[Median].x() + obs_points[
                                 VecCenterIntersection[0]].x()) / 3;
                        Baricenter.y() = (obs_points[pos_first].y() +
                                 obs_points[Median].y() + obs_points[
                                 VecCenterIntersection[0]].y()) / 3;
                        Baricenter.z() = 0;

                        TempVector.clear();
                        TempVector.push_back(Baricenter);
                        temp4 = firstIntersection(TempVector, genericCurve,
                                  true);
                        paramOrder.push_back(temp4.w());

                        index = UpdateParameterOrder(paramOrder);
                        ctrlPoints.insert(it + index, Baricenter);
                        weights.insert(it1 + index, 5);
                        UpdateNurbs(degree, knots, weights);
                        genericCurve = std::make_shared<GenericCurve>(
                                 degree, knots, ctrlPoints, weights, coeff,
                                 dimension, degree + 1);
                        it = ctrlPoints.begin();
                        it1 = weights.begin();

                        for (int s = 0; s < paramOrder.size(); s++)
                            paramOrder[s] = UpdateParameterDistance(
                                 ctrlPoints[s + 1], genericCurve);

                        //Evaluation path between near obstacles
                        int from = -1;
                        if (Median1)
                            from = Median;
                        if (cnt == 2)
```

```
                            from = pos_first;

                        freeMediumPoint = FindCenterMedian(obs[from], obs[VecCenterIntersection[0]]);
                        TempVector.clear();
                        TempVector.push_back(freeMediumPoint);
                        temp4 = firstIntersection(TempVector, genericCurve, true);
                        paramOrder.push_back(temp4.w());

                        index = UpdateParameterOrder(paramOrder);
                        ctrlPoints.insert(it + index, freeMediumPoint);
                        weights.insert(it1 + index, 5);
                        UpdateNurbs(degree, knots, weights);
                        genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff, dimension, degree + 1);
                        it = ctrlPoints.begin();
                        it1 = weights.begin();

                        for (int s = 0; s < paramOrder.size(); s++)
                            paramOrder[s] = UpdateParameterDistance(ctrlPoints[s + 1], genericCurve);

                        TempVector.clear();

                        if (EvaluatePointInObstacle(TwoPointsExternal[1]) == 0) {

                            TempVector.push_back(TwoPointsExternal[1]);
                            temp4 = firstIntersection(TempVector, genericCurve, true);
                            paramOrder.push_back(temp4.w());

                            index = UpdateParameterOrder(paramOrder);
                            ctrlPoints.insert(it + index, TwoPointsExternal[1]);
                            weights.insert(it1 + index, 5);
                            UpdateNurbs(degree, knots, weights);
                            genericCurve = std::make_shared<GenericCurve>(degree, knots, ctrlPoints, weights, coeff,
                                dimension, degree + 1);
                            it = ctrlPoints.begin();
                            it1 = weights.begin();

                            for (int s = 0; s < paramOrder.size(); s++)
                                paramOrder[s] = UpdateParameterDistance(ctrlPoints[s + 1], genericCurve);
                        }
```

```
                    }
                    else {

                        // We have only two near obstacles
                        TempVector.clear();
                        TempVector.push_back(TwoPointsExternal[1]);
                        temp4 = firstIntersection(TempVector, genericCurve,
                                  true);
                        paramOrder.push_back(temp4.w());

                        index = UpdateParameterOrder(paramOrder);
                        ctrlPoints.insert(it + index, TwoPointsExternal[1])
                                 ;
                        weights.insert(it1 + index, 4);
                        UpdateNurbs(degree, knots, weights);
                        genericCurve = std::make_shared<GenericCurve>(
                                 degree, knots, ctrlPoints, weights, coeff,
                                 dimension, degree + 1);
                        it = ctrlPoints.begin();
                        it1 = weights.begin();

                        for (int s = 0; s < paramOrder.size(); s++)
                            paramOrder[s] = UpdateParameterDistance(
                                 ctrlPoints[s + 1], genericCurve);
                    }
                }
            }

            tuple<double, double> closest = genericCurve->FindClosestPoint(
                                 ctrlPoints[ctrlPoints.size() - 2]);
            auto abscissa_m1 = (double)(get<0>(closest));
            ClosestPoint = genericCurve->At(double(abscissa_m1));
            wayPointsClosest.push_back(ClosestPoint);

            printNurbsInformation(knots);

        }
        else
            break;
    }
    pathRobotCalculation();
    CalculateWayPoints(wayPointsClosest);
}
```