Path planning for Lunar rovers

A lunar surface path finding and obstacle avoidance algorithm

L.I.A. Gelling

Delft University of Technology

TU Delft

Path planning for Lunar rovers

A lunar surface path finding and obstacle avoidance algorithm

by

L.I.A. Gelling

To obtain the degree of Master of Science Embedded Systems
at the Delft University of Technology,
to be defended publicly on Tuesday January 31, 2023

Author:	L.I.A. Gelling	
Student number:	4590600	
Project duration:	April 15, 2022 – Januari 31, 2023	
Thesis committee:	Dr. ir. C.J.M. Verhoeven	TU Delft, Chair
	Dr. ir. R.T. Rajan	TU Delft, Daily Supervisor
	Prof. Dr. G.C.H.E de Croon	TU Delft
	Dr. ir. A. Noroozi	TU Delft

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft

Abstract

The unique six-legged swarming rover Lunar Zebro is designed and produced by students from the Delft University of Technology. The objective of the rover is to accomplish an autonomous mission on the Lunar surface by 2024. This thesis evaluates a path planning algorithm that is designed for autonomous navigation in the Lunar environment. The thesis studies existing path planning algorithms and determines the essential functionalities of the algorithm and the unique requirements of Lunar Zebro. It is found that an Artificial Potential Field based path planning algorithm accommodates the determined needs and requirements. With the help of the Artificial Potential Field path planning algorithm and the unique requirements, a vector field based algorithm is developed. The algorithm uses an attractive vector field to attract the rover to the determined target. Meanwhile, obstacles or other obstructions are denoted by a repulsive rotational vector field around the edge of the obstacles. This rotational repulsive force ensures obstacle avoidance and prevention of the local minimum trap, which often occurs in Artificial Potential Field path planning. Improvements are suggested to increase reachability and decrease path length and planning time of the rotational vector field algorithm. In the Python developed simulation, the improved algorithm accomplishes a 62% reduction in planning time compared with the original Artificial Potential Field algorithm and achieves similar path length results. Moreover, the proposed algorithm has a reachability of 90% where the Artificial Potential Field algorithm just reaches a success rate of 55%. The thesis concludes with the future work recommendations for a low level implementation in C or either C++ to facilitate the deployment in a microcontroller.

Keywords: Lunar navigation, Path finding, Obstacle avoidance, Lunar Zebro, Swarming, Vector field path planning, Rotational Vector Field.

Preface

In 1969 Neil Armstrong was the first person ever to set foot on the moon surface. When touching the moon surface he spoke the famous words: “one small step for man, one giant leap for mankind”. Ever since this first moon landing, exploration has been a trending topic. Plenty of research is done towards the expansion of the physical exploration of the moon. Both humans and rovers have explored the Lunar surface multiple times after the first moon landing and it is for sure that this will continue in the future. This thesis is presenting a path planning algorithm that is able to navigate a Lunar rover through the Lunar environment. With the presentation of this thesis, mankind will again be one small step closer to the autonomous exploration of the moon.

Acknowledgements
This work would not have been possible without the support of many people. I am grateful to all of those with whom I have had the pleasure to work with during this project. In particular I would like to thank Dr. ir. Raj T. Rajan and Dr. ir. Chris J.M. Verhoeven who supervised me during the intensive period of nine months which resulted in this thesis. Secondly I want to thank my friends and family, especially my parents and two younger sisters who always supported my choices that have led me to the finishing of this thesis.

Contents

List of Figures

List of Tables

List of Algorithms

Acronyms

AI Artificial Intelligence . . . 35
ANN Artificial Neural Network . . . 13
APF Artificial Potential Field . . . 20
DE Differential Evolution . . . 68
ES Evolution Strategies . . . 68
GA Genetic Algorithm . . . 67
GPS Global Positioning System . . . 1
IDE Integrated Development Environment . . . 25
IMU Inertial measurement unit . . . 4
LIDAR Light Detection and Ranging . . . 4
LZ Lunar Zebro . . . 1
OPAL Stereo Vision Obstacle Processing ALgorithm . . . 4
PSO Particle Swarm Optimization . . . 68
RRT Rapidly exploring Random Trees . . . 19
RVF Rotational Vector Field . . . 44
SLAM Simultaneous Localization And Mapping . . . 13
VSC Visual Studio Code . . . 25
ZPU Zebro Processing Unit . . . 3

1 Introduction

In July 1969 the Apollo 11 spacecraft landed the first humans on the moon for an exploring mission of the Lunar surface. Shortly after this moon mission, the first Lunar rover was deployed. In 1970 the Lunokhod 1(Russian for ”Moonrover 1”) touched the Lunar surface for further research. The rover had a length of 2.3 metres and a mass of 756 kilograms. Several other Lunar rovers have landed on the moon surface for exploration and research since 1970 [7]. Each rover of similar size and weight as the first one that was ever landed on the moon. Positioning and navigating of these rovers have been a challenge since the early exploration missions [31]. The non-existence of a Global Positioning System (GPS) or any other form of global positioning, makes localization and navigating a challenging task for Lunar rovers. Several studies have been proposing different techniques that could be applied to solve these problems [22]. Extensive research, testing and moon missions are needed to validate a concept that enables the possibilities of exploring the full Lunar surface. The Lunar Zebro project, carried out by students on the Delft University of Technology, develops a rover that overcomes these challenges and continues where other rovers stop.

Lunar Zebro mission

Lunar Zebro is a unique six-legged nano-rover that is build for a mission on the moon. The unique properties of the rover can be found in the size, walking behaviour and swarming capabilities. The rover is a so called nano-rover the typical length, width and height which are less than 400mm by 300mm by 200mm respectively. The rover is equipped with six legs instead of wheels, that enables the rover to move forward and backward. The legs ensure a robust but slow walking behaviour. The rover is initially designed to operate in a swarm with multiple rovers. A swarm is said to be the cooperation of multiple rovers with the same objective, but individual behaviour. The size, walking behaviour and swarm capabilities make the Lunar Zebro a unique moon exploration rover with objectives that have never been achieved before.

The objective of the first mission will be a single rover on the moon to explore the possibilities and validate the concept of the rover. The focus will be on traveling 200 meters in one lunar day, which is roughly 29.5 earth days. During this mission the rover must successfully survive the moon conditions and transmit data with the lander. Moving on the Lunar surface must be executed autonomously. In other words, the rover must navigate and operate without the interference of external inputs. Achieving autonomous navigation of the rover will open up the possibilities for operation in a swarm.

Thesis objective

Positioning and navigating is not only a challenge for past lunar exploration rovers, but also challenges the Lunar Zebro. Navigating is said to be the guidance on the lunar surface, where possible obstacles must be avoided and set target locations must be reached. At this moment in time the Lunar Zebro (LZ) navigation problem is not yet solved and obstructs the possibility to execute the mission. The thesis will address this problem and proposes a suitable implementation for navigating the rover through the Lunar environment.

Thesis synopsis
The remainder of this thesis is divided in seven chapters. The first two Chapters (Ch.2 and Ch.3) are focused on gathering all existing information on LZ and on state of the art algorithms for path planning in robots respectively. Chapter 4 is defining the problem that can be deducted from the LZ requirements and state of the art path planning algorithms. This is followed by Chapter 5 that is giving a detailed analysis on a suitable basic planning algorithm for swarming moon rovers. Chapter 6 proposes a relevant improved algorithm that uses the concepts of the basic algorithm explained in Ch.5. The long term goal is an LZ implemented path planning algorithm, Chapter 7 therefore discusses the implementation of the proposed algorithm in a low level programming language that can run on the hardware of LZ. Finally the conclusion is drawn and further work is proposed in Chapter 8.

2

Lunar Zebro

One of the remaining challenges in the LZ project is the navigation of the rover through the moon environment. As this problem is to be tackled in this thesis, the focus will be on the latter and the adjacent issues that are required to identify the problem. Before a problem can be clearly defined, the functional state of LZ must be determined. The functional state can be described by a hardware state and a software state. The hardware and software states give an indication on the detailed issues that still need to be solved to obtain a solution for the problem. This chapter will first describe these states in detail. An even more important aspect of the LZ project are the unique objectives. These objectives make the project one of a kind. The corresponding unique features will be discussed after the functional state description. The chapter will conclude with LZ requirements and a detailed problem definition.

2.1. Hardware state

As hardware is a widespread concept, for the sake of simplicity, it will be defined as the physical components that must interact with software to obtain full functioning. The hardware state of LZ is therefore limited to the processor, actuators and sensors that are part of the rover. The software for the navigation problem is running on the processor and only interacting with actuators and sensors and not with other hardware like the chassis or the payload. Within the LZ project there are two types of rovers. One that is actually prepared for launching to the moon(moon rover) and one that is used for testing on earth(terrestrial rover). The one for the moon is build with strictly essential hardware as everything adds up to the weight and energy consumption. The rover for testing on earth has more freedom in processing power and gives the possibility to add some hardware for testing.

2.1.1. Processor

The processor is quite important regarding the software algorithm possibilities. Simple rule-based algorithms or deterministic algorithms use less computational power, while machine learning algorithms have a relatively high computational demand. The capabilities of the processor are therefore of great importance in the software decision process.

The Terrestrial Rover is equipped with a Raspberry Pi as processor. When using a machine learning based algorithm, the training of the algorithm will be done on external hardware. In other words the training will not be done on the processor in LZ. A Raspberry Pi is perfectly able to execute an externally trained machine learning algorithm and is therefore also suitable for simple Rule-based or deterministic algorithms. Implementing state of the art algorithms is therefore not considered to be a problem.

The Moon Rover is equipped with the CP400.85 and is called the Zebro Processing Unit (ZPU). This processor has less computational power compared to a Raspberry Pi. It is hard to say how much process power is exactly available as multiple programs and/or algorithms are running on this processor

during operation. However, it can be said that the lack of computational power should be taken into account during the design of the algorithm. Computational power will most likely be a limiting factor.

2.1.2. Sensors

Currently there are multiple sensors available on the rover that could or must be used for the execution of a navigation algorithm. However, this is based on the implementation of the current rovers. It is not yet clear if these will change in upcoming versions. It might be able to add or replace sensors in the upcoming versions if this is strictly necessary for the optimal operation of a navigation algorithm.

The Terrestrial Rover is equipped with 2 camera's for stereo vision. Stereo vision enables the possibility to estimate dept and therefore to estimate the distance to objects in the field of view. The software that processes the data from this camera's and gives the possibility for dept analysis is also known as Stereo Vision Obstacle Processing ALgorithm (OPAL) [29]. Furthermore it is equipped with a Light Detection and Ranging (LIDAR) that is not working due to the lack of supporting software. Finally the earth rover has two times an Inertial measurement unit (IMU) and an ultra sonic sensor to determine distance between the rover and objects.

The Moon Rover has a similar sensor configuration but is missing the LIDAR and the ultrasonic sensor. The ultrasonic sensor is not able to perform on the moon environment as it has a different atmosphere composition. In addition, the moon rover has a solar sensor to determine the perfect angle of the solar panel to yield maximum solar energy.

2.1.3. Actuators

The only actuators that are influenced by the navigation algorithm are the motors that enable LZ to walk. The rover has six separately controller legs, each directly connected to an individual motor. This means the systems controls six motors individually to ensure the movement of the rover. Each motor is connected to a motor driver and each motor driver is controlled by software. The software to control these drivers is called the locomotion algorithm [23]. The actuators and their control are identical for the terrestrial rover and the moon rover.

2.2. Software state

The second researched subject is the available software for LZ and the level of completion. The level of completion is characterized by the deployment of software that is written and tested. To obtain a detailed problem formulation, an overview of all software modules in LZ is given and an extensive explanation is given on the software modules that are involved in the navigation of LZ.

2.2.1. Software overview

The software of LZ is divided in modules which communicate with the "brain" of LZ. This brain, or master control, is called TRON. Both TRON and all adjacent sub-modules are running on the ZPU. All sub-modules are connected via TRON and can only communicate with TRON and not with the other sub-modules. Fig.2.1 shows all connected modules and building blocks of TRON. The modules involved in navigation are shown in red. This includes the OPAL software module for processing camera data and the locomotion software with the leg modules for movement of the robot. There are no other software modules that are related to the navigation of LZ. This means that these two modules form the available LZ software on the navigation problem and taking these modules into account can be defined as the complete search area for a detailed thesis problem.

2.2.2. Navigation software

Navigation is still a relatively broad concept that can be divided in multiple smaller problems. The problem can be divided in three parts: Obstacle detection, obstacle avoidance and path planning. This is based on two assumption, the first one being that a rover is facing a known direction and is given a target location(goal) that needs to be reached by the rover. The second being that the generated path is executed by one or multiple algorithms that ensures the correct motion of the rover. Looking from this perspective the steps that are involved in making a rover move, are shown in Fig.2.2. It starts with a

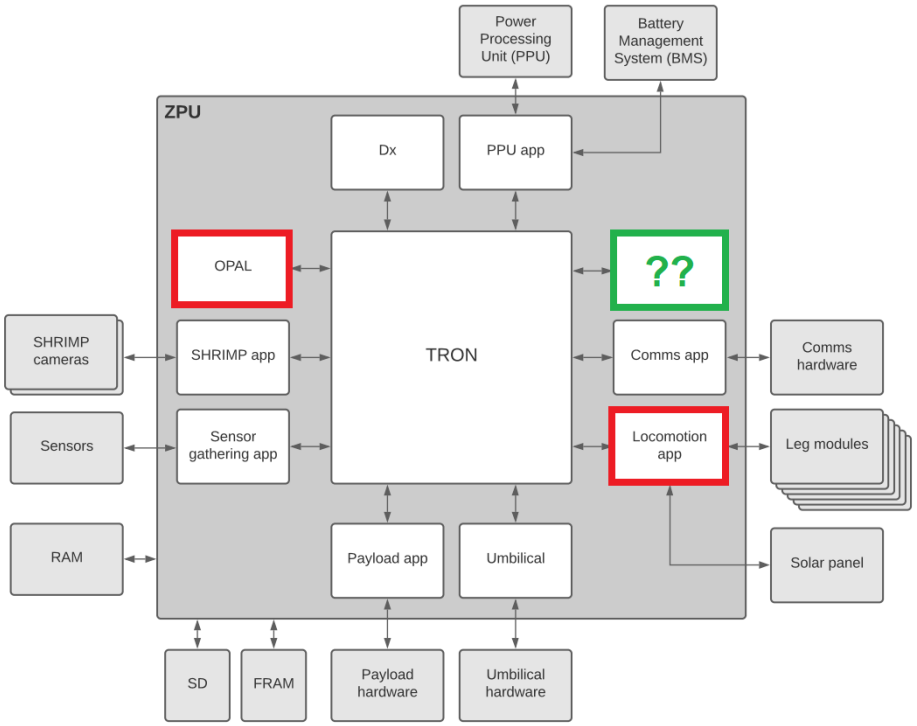


Figure 2.1: Overview of software modules in the ZPU

direction or target from the starting position. A path is formed between the start and the target, which is typically a straight path. When the rover moves to the target, it encounters detected obstacles. When an obstacle is detected it should be avoided and a new path should be created with the knowledge of this newly detected obstacle(s). Finally the rover should move over the generated path. The algorithm that ensures the movement of the rover is divided in a locomotion algorithm and the individual motor control algorithm as the rover consists over multiple motors.

For the LZ project it holds that the yellow block shows the part that still needs to be determined or defined. Green blocks show the modules that are (almost fully) completed or are currently build. Excluding the yellow block from the problem, as this is determined later in the project, the problem is defined between the existing(green) blocks. These can therefore be seen as the boundaries in the search toward a detailed thesis problem proposal. The red blocks, ”Obstacle avoidance” and ”Path Planning”, form the yet undefined problem. Before taking a closer look on the definition of this problem, the functioning and the level of implementation of the remaining blocks must be defined.

Obstacle detection

The obstacle detection module can again be divided in smaller sub-modules to simplify the concept and implementation of the complete obstacle detection module. The first sub-module enables stereo vision such that depth estimation is possible. The second sub-module is the detection of multiple objects in the field of view instead of just one. The last sub-module ensures the estimation on the distance to the objects. Together these sub-modules forms the obstacle detection, also referred to as the OPAL software. Tab.2.1 shows these sub-modules and shows which parts have been written and tested. Note that the software is coded in a low level language (C++) and is tested in a high level language(Python). This is done because C++ is extremely powerful and efficient on processor level to ensure optimal usage of the processor on the moon rover. Python on the other hand is very powerful when used for data processing while training and testing machine learning algorithms. For training and testing of the OPAL software, pictures from the actual moon are used. Such pictures are shown in Fig.2.3. The ”Stereo vision” and ”Multi object detection” is working and tested. The algorithm placed boxes around (multiple) detected rocks in the shown pictures. Currently the distance to obstacles part in the code is written but not yet tested. This means that the true distance to the obstacles can not be determined.

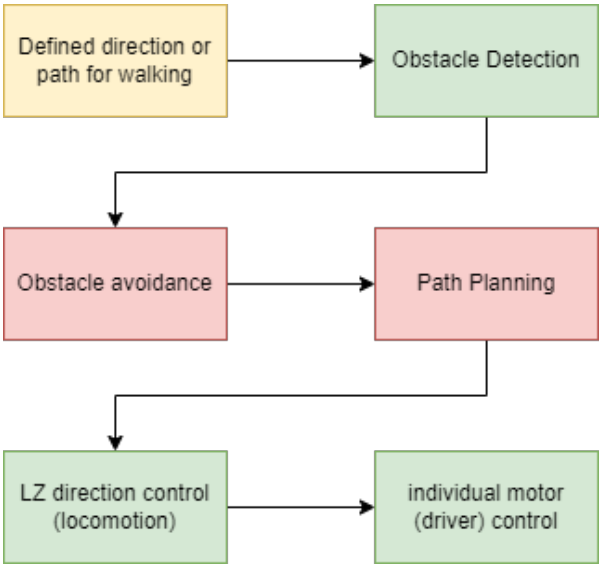


Figure 2.2: Sequential software building blocks needed for the movement of a rover

This cannot be determined as the dept projection of the pictures is not verified to be equal to the true distance between the object and the rover. Remaining steps would be to test the distance calculations to objects and to train and test the algorithm while the robot is in motion.

In short, a preliminary conclusion shows that when the obstacle detection algorithm is finished, the robot should be able to detect obstacles and to determine the distance to the edge of the obstacle. The actual size or radius and the middle point of the obstacle are not yet known. It requires additional software to transform the known information from the obstacle detection to relevant information for a navigation algorithm. To clarify the problem Fig.2.4 shows the top view of LZ in front of a rock(obstacle). LZ is drawn in the 2D XY-plane where the positive Z axis comes out of the paper. Navigation will be done in the 2D XY-plane and the rock will have a midpoint (blue dot) and a radius around this midpoint, which denotes the ”forbidden” area for the rover. However, right now the rock is detected only in the 2D YZ-plane (red line) where the midpoint is also defined in the 2D YZ-plane only (green dot on red line). Therefore to make the obstacle information relevant for a navigation algorithm it should be mapped form the YZ-plane to information on the XY-plane. The software to map this information to the relevant plane is still to be made. However, it is assumed that the output of this software would be a midpoint and the size of an object in the 2D XY-plane. At this point an object that is detected denotes an obstacle for LZ and the corresponding midpoint and size information is therefore considered as the input for the navigation algorithm.

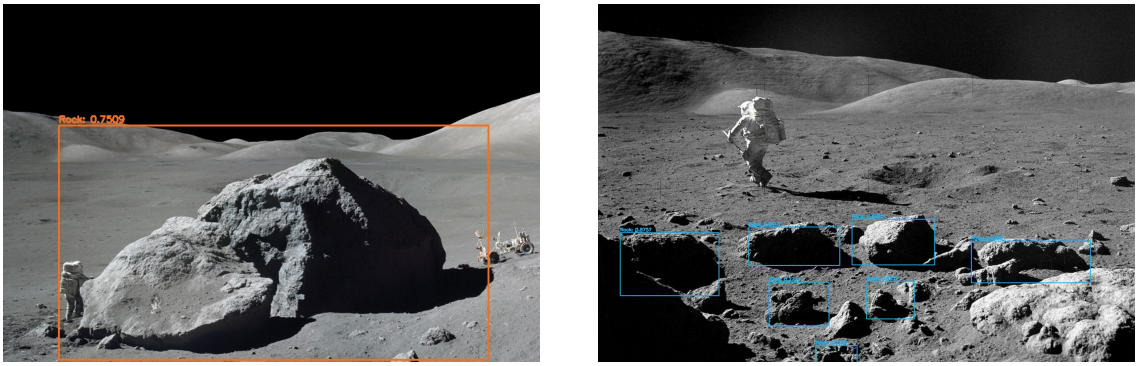
Table 2.1: Sub-modules from obstacle detection written and tested

Part	Written(C++)	Tested(Python)
Stereo Vision	x	x
Multi object detection	x	x
Distance to objects	x	

Locomotion

The Lunar Zebro direction control, called the locomotion module, is focused on controlling the motion of LZ as a complete entity. This module is responsible for walking a trajectory in a given field. The inputs of the locomotion algorithm are the outputs of the problem that is defined by the red blocks in Fig.2.2. At the time of writing the locomotion module is not yet completely finished, which therefore requires a clear definition of the boundaries between these modules.

Currently the locomotion algorithm is able to move the rover forward and backward and statically change directions. This results in the rover not being able to walk curved paths but only change directions while standing still. This characteristic can collide with a path smoothing algorithm. Going



(a) Single Rock detection

(b) Multiple Rocks detection

Figure 2.3: Rock detection on pictures from the moon

Figure 2.4: LZ in front of a detected object(rock)

only forward, backwards, left and right would bring an unusual complexity and inefficiency to the path planning as this is against the function of most optimal path planning algorithms. It is therefore chosen that statically changing directions could be done in degrees left or right instead of only going left(-90°) or right(90°). This ensures the possibility of an optimal path and the correct functioning of LZ. Beside direction changing, the amount of steps, standard deviation and speed are dependent on the settings and hardware of the individual rover. For these problems a more detailed analysis on the actual LZ rovers is needed and currently unavailable. As the solution for these problems can be generated with a clear defined path and the individual rover parameters, it is chosen that a generated path would be a sufficient input for the locomotion module. This path would be generated from the starting position and would be generated again when an obstacle is detected. When the rover detects an obstacle and a new path is generated, the new starting position will be the place of the rover at the time of detecting the obstacle. This path would be the output of the path planning module. By outputting a path, the locomotion module could generate a distance(steps), a direction and if desired a speed to achieve the goal. Possible inaccuracy or slippage of the rover could therefore be taken into account in the locomotion algorithm.

The output of the locomotion module is defined as the input of the individual motor control module, where each leg is controlled. Each individual leg motion contributes to the movements of the rover as a complete entity. This results in the locomotion module being one control level above the individual motor driver control module.

Individual motor driver control

The other end of the navigation problem is the low level control of the legs. Each of the six legs are connected to a brushless DC motor. This motor is driven by a motor driver. The low level controls for these drivers are written and give the opportunity to control each leg separately. The input of these

drivers is the output of the locomotion module. The drivers can be given a position of the brushless motor and a δt to move to the desired position. In other words the speed and position of the leg can be controlled. When requested, the drivers can give their temperature, the leg position and a fault status.

2.2.3. Obstacle avoidance and path planning

From Sec.2.2.2 it can be concluded that a relevant and academic problem is designing and testing the obstacle avoidance and path planning software modules. The stated navigation problem can be scaled down to the design of these two software modules. Looking at Fig.2.1, it shows the modules that are involved in the navigation surrounded by red blocks. The actual path planning and obstacle avoidance algorithm is not particularly defined in either of these modules. It can therefore be said that either the definition of the modules need to be rephrased or an additional module for navigation should be introduced as shown in green. In the past, this module was already defined, but it was taken out to keep the first models of LZ simple. To keep the original software and modules structured and accessible by the current software architecture, it is chosen to add a new module instead of rephrasing the existing modules. This additional module was and will be defined as ”Navigation”. It will host the algorithm that at least defines a path from start to goal while avoiding obstacles. As input this module will use data from the OPAL module and information on the goal that is set for the rover.

2.3. Unique features

As stated in the beginning of this chapter, Lunar Zebro is a one-of-a-kind project that comes with several unique objectives. These unique objectives result in requirements for the final navigation algorithm. Lunar Zebro has three unique features that will be discussed in this section; ability to operate in the moon environment, swarming capabilities and the walking behaviour.

2.3.1. Moon environment

As stated, the main objective of LZ is to perform missions and tasks on the moon. The moon environment is much more challenging than the well known earth environment. This results in some strict hardware requirements to ensure operation on the moon. Beside hardware requirements it also impacts the software requirements. When the rover is on the moon it is almost impossible to tweak, reset or help the rover in any way. For this reason the software must be build in a robust way such that software module failure will not automatically lead to mission failure.

2.3.2. Swarming behaviour

Looking from the bigger picture, the rover is built with the objective of swarming. In other words the rover operates in a group of rovers that perform swarm behaviour regarding the set target or goal of the mission. There will not be a central agent which controls the swarm. Each rover must therefore have autonomous capabilities to navigate, walk and perform other tasks. Moreover, it must also be able to perform swarm tasks. The navigation algorithm that will be deployed in the LZ rover must be capable of performing swarm tasks and autonomous rover tasks. Swarming can add additional boundaries to the design but can also result in less weight on an individual rover requirement. As a result of operating in a swarm, the impact of an event on individual rovers could be reduced. This relaxation of requirements duo to swarm capabilities will be further explained in Ch.4.

2.3.3. Walking behaviour

The walking behaviour of LZ brings in some relevant designing aspects. LZ has legs instead of wheels, these legs control the motion of LZ. Two important behaviours are the result of using legs. The first behaviour is the relatively slow movement. Second, the completion of a step is not similar to wheel behaviour.

Slow movement

Moving with legs results in a relatively slow but robust movement as it is simple to control slow movements. Compensation for the desired path is relatively small as the deviation of the path is happening slowly. However, the locomotion algorithm will still not be able to walk the path with a 100% accuracy. Naturally there are small deviations that need to be compensated. This impacts the free boundary that has to be formed around LZ while navigating.

For the navigation of LZ a certain distance between the obstacle and the center of LZ should be considered. Naturally the first assumption would be that this distance should be half the length and width of LZ. However, because of the inaccuracy of the locomotion algorithm the maximum path deviation should be added to the clearance area around an obstacle. As this deviation is yet unknown and depending on individual rover specification, it is assumed to be an input parameter of the navigation algorithm.

Step movement
Wheels can be very accurate as you could rotate them on every position. To fulfil a step with LZ, one rotation of the legs must be completed. This means the steps of LZ can have a maximum accuracy that is equal and thus limited to the step size. The navigation algorithm must be able to supply a path that can be as accurate as the step size of the rover. However, an accuracy that is better than the step size has no additional benefits as it is limited by the hardware. [2] proposes a formula to calculate the step size of LZ. The step size is not a fixed variable as it is dependent on the height of the rover. The formula to calculate the step size is shown in Eq.2.1, where s is the step size in cm and h is the height of the rover in mm.

$$s(h) = \frac{29 * cos(2.1h - 1.5)}{2} + sin(2.4h) - \frac{7 * sin(9h)}{20} \tag{2.1}$$

Taking a maximum height of 35mm results in the smallest step size of 5.723 centimeters. An accuracy of 5 centimeters would therefore be more than enough to accommodate the needs of LZ.

2.4. Problem definition and requirements

Sections 2.1, 2.2 and 2.3 give a clear insight on the unsolved issues and the unique requirements corresponding to the LZ project. A problem statement and the corresponding requirements are presented below.

2.4.1. Problem statement

Concluding on the open issues of obstacle avoiding and path planning, an algorithm has to be made that covers both. The result would form the connection between the ”Obstacle detection” and the ”Locomotion” modules. This connecting module would be named the ”Navigation” module. The ”Navigation” module hosts the algorithm that is the answer to the research question: *”Develop a 2D path finding and obstacle avoiding algorithm, which can run on a single Lunar Zebro and supports possible swarm behaviour.”*

2.4.2. MoSCoW analysis

An overview of the resulting requirements for an LZ navigation algorithm will be listed according to the MoSCoW method. The MoSCoW method is divided in several priorities that highlight the relevance or irrelevance for a final design. The different priorities are:

- Must Have (M): These requirements must be in the final design. Without these requirements the algorithm is useless in its field of application.
- Should Have (S): These requirements are important but not mandatory for delivery of the algorithm in the current delivery timeline.
- Could Have (C): These requirements are desirable but do not add to the functionality of the algorithm. It could improve user experience or freedom of implementation.
- Won’t Have (W): It is agreed by the stakeholders that this requirement will not be part of the algorithm.

Below, all requirements for the navigation algorithm are listed according to the MoSCoW method. The requirements are ordered according to their priority. In Must have, Should have and Could have the lowest number is the highest priority.

Must Have

1. Path planning from starting point to end point,
2. Ability to operate in swarming behaviour,
3. Online/Local path planning (avoiding obstacles),
4. Accuracy of the path planning algorithm should be less than the accuracy of the rover caused by the walking behaviour,
5. Inputs: Target location(with respect to itself), Distance to obstacle center and Shape of obstacles,
6. Outputs: Path points with respect to starting point,
7. Fully tested and simulated navigation algorithm,
8. Ability to run on the terrestrial rover,
9. State of the art software/algorithms to ensure academic level.

Should Have

1. Ability to run on current moon rover.

Could Have

1. Dynamic obstacle avoidance,
2. Path planning that avoids the shadow of rocks,
3. Global planning in advance.

Won't Have

1. Software or algorithms that are related to the blocks that are not coloured green or red in Fig.2.1,
2. Solar panel angle control,
3. Obstacle detection, classification and localization,
4. Low level control of the motors,
5. Locomotion algorithm, which makes the robot controllable as one entity,
6. Position Control to follow desired path,
7. Any form of control that deals with the rover inaccuracy that causes the rover to deviate from the desired path (slippage, standard direction deviation, etc),

3

State of the Art Navigation algorithms

Knowing the problem that is stated in Ch.2, applicable algorithms can be researched. With the knowledge of all available algorithms and their characteristics, it can be determined which algorithms are applicable for Lunar Zebro navigation. An overview of state of the art navigation algorithms is presented first. This will be followed by a classification of the presented algorithms. The chapter will conclude with three possible base algorithms that are chosen according to the classification and the unique LZ requirements.

3.1. State of the art algorithms

There are two types of path planning, global and local path planning. Global path planning requires the environment to be static and already known. So the controller can define the complete path before execution of the path. For local path planning, the environment is not known and is determined during the movement of the robot. The robot can therefore define a (new) path with real-time input of the environment. On the first hand it seems that global path planning is not suitable for the Lunar Rover as it has to avoid obstacles that are detected during the mission. However, this is not entirely true. Literature proposes algorithms which are a combination of global and local algorithms and will therefore work in local conditions as well. Another aspect that is involved in navigation is the environmental modeling. Before an algorithm can find an optimal path, it first has to be chosen how the environment is modeled.

3.1.1. Environment modeling

An environmental model is an abstract or formal description of the structure and function of the environmental system [17]. This model is constructed before the actual path planning and is modeling the environment in a suitable form such that the algorithm can optimise a path. Optimising the way of modeling could reduce calculations and can therefore increase the speed of the algorithm. The most common and well known method is the grid model, which makes use of an evenly distributed grid that is placed on the field. This is a discrete model where each grid point is used as a place where the rover could be located and a path point could be formed. [17] and [40] propose several other ways to model the environment. These approaches are listed below and are extensively discussed in AppA.1.

- Grid model,
- Cell tree model or decomposition approach,
- Voronoi diagram model,
- Tangent graph Method,
- Visibility graph model,
- Free space approach,
- Topological method,
- Probabilistic roadmap method.

3.1.2. Navigation Algorithms

Modeling the environment is the first step in the navigation from start to goal, but is not resulting in an actual route. For obtaining a path between the start and the goal, the points that are created in the environment model need to be connected. Connecting the points between the start and the goal is done by a navigation algorithm, which yields in a path. Each algorithm differs in their problem approach and their used environmental model. Generated paths are therefore not similar between algorithms. However, some algorithms work in a similar way or are based on the same principle. As the current availability of navigation algorithms is very broad, it is chosen to group algorithms that have similarities in their path generation characteristics. According to [17] navigation algorithms can be divided in four categories; Methods based on a geometric model search, Probabilistic sampling-based algorithms, Artificial potential field type algorithms and Bio-inspired Intelligence algorithms. A fifth category is added as some of the available algorithms are only usable in combination with other algorithms. These are named the miscellaneous algorithms and makes the classification categories complete. Unless stated otherwise, all algorithm definitions are formed on the basis of the papers [4], [17], [40] and [38]. The detailed explanation and functioning of the algorithms can be found in App.A.2.

Methods based on a geometric model search

Can be considered as the more classical path search algorithms. The map or environment of the robot is made discreet with one of the environmental models. The robot is than localized with respect to this geometric map [30]. From the localization point, the path to the goal is planned and executed by determining the next optimal grid point according to the algorithm policy. The used environmental model and accuracy of the model has a big influence on the performance of the planned path. The following methods are considered to be based on a geometric model search.

- A* Algorithm
- D* Algorithm
- Dijkstra Algorithm
- Level set Method
- Fast marching algorithm
- Boustrophedon Decomposition Algorithm
- Internal Spiral Algorithm

Probabilistic sampling-based algorithms

These algorithms are based on, as the name already suggests, random samples that are independent and identically distributed in the environment [12]. These samples, or nodes, are than used to construct a path from the start location to the goal. These methods are non-optimal regarding path length and make use of the random environment modeling technique. The two available method are shown below.

- Rapidly exploring Random Trees
- Probabilistic Roadmap Method

Artificial potential field type algorithms

A potential field is constructed, which is used as a map for path planning. The field in the neighborhood of obstacles is repulsive and the field close to the target point is attractive[28]. The potentials ensure a path around the edge of an obstacle, while being attracted to the actual target. Algorithms that make use of repulsive and attractive potentials are shown below.

- Artificial potential field
- Bug algorithm
- Vector-polar histogram

Bio-inspired Intelligence algorithms
This is a specific class that is not sharing a common approach but a common characteristic. These methods are mimicking human, animal or organism behaviour to learn non-linear relations. When this behaviour is either way trained or described in the algorithm, it is able to use this information to obtain a solution in a new and unknown environment. A specific type of algorithms in this class, are the swarming animal algorithms. These are deducted from swarm and colony behaviour of animals and consist in multiple types. These types are further explained in App.A.2.

- Swarming animal algorithms
- Genetic Algorithm
- Differential Evolution
- Imperial competition algorithm
- Simulated annealing algorithm
- Q-learning/dynamic programming
- Tabu Search
- Reinforcement learning
- Deep Reinforcement learning

Miscellaneous algorithms
The last class of algorithms is also a class that is not based on a common path planning approach. For full functionality they all require the use of another algorithm for either training or collaborated use. An example is the use of an Artificial Neural Network (ANN), which requires extensive training based on predefined data. This predefined training data is obtained based on paths that are created by other navigation algorithms or by the user preferences. Relevant miscellaneous algorithms are shown below.

- Artificial neural network
- Fuzzy
- Cased-Based Learning Method
- Behaviour decomposition method
- Rolling Window Algorithm
- Simultaneous Localization And Mapping (SLAM)

3.2. Considered navigation algorithms
Sec.3.1 shows all the relevant state of the art navigation algorithms and the possible environmental mapping methods. However, some of these navigation algorithms are more relevant for LZ than others and some are even irrelevant for LZ. First the detailed scope for the navigation algorithm will be determined. With the use of this scope, irrelevant algorithms can be excluded from the problem. With the reduction of possibilities, three algorithms will be chosen for further research.

3.2.1. Algorithm scope
Looking at all the presented algorithms there are two commonly made assumptions that change in the scheme of LZ. The unique requirements stress the swarm functionality and the mission objective of going to the moon. The moon environment brings requirements regarding robustness but on top of that a problem with the localization of the rover.

Navigation and swarm capabilities
The implemented navigation algorithm must not prevent the rover from functioning in a swarm. However, this requirement does not mean that the navigation algorithm has to be a swarm algorithm itself. In fact, the rover must be autonomous and must be able to navigate without the information of the swarm. [1] proposes an example of a robot with swarming capabilities but also a separation between the actual navigation and swarm algorithms. Looking at the software architecture of LZ as stated in Sec.2.2.1, it fits the way of implementation to split the navigation and swarm algorithms. The swarming algorithm can be implemented in a different software block, as long as the navigation software block keeps the ability to reset the target if this is desired by the swarming algorithm.

Localization problem
Most of the presented solutions for navigation problems make use of the fact that the position of the vehicle or robot is known somehow. On earth this can be done with GPS for example. A yet unsolved problem for a lunar rover is the unknown position because there is no GPS available on the moon. This could be solved with for example a SLAM solution. Taking a closer look at when the localization is used, it can be reduced to ensuring that the rover follows the generated path. Following the path and staying on the path is part of the locomotion problem. As stated in the requirements from Sec.2.4, any software development that is related to the locomotion software will not be part of the navigation problem. Therefore the position problem is excluded from the navigation algorithm.

For the navigation algorithm it is assumed that the starting position is known (as the lander position is known) and that the goal position is known. During the execution of the path, the rover should keep track of the steps that are taken. This ensures the correct replanning starting point when an obstacle is detected.

3.3. Consideration of algorithms
Sec.3.1 shows all the state of the art algorithms relevant for path planning. However, some of these algorithms are more relevant for Lunar Zebro than others and some are even irrelevant in the use case of Lunar Zebro. By looking at the algorithms functioning and the LZ requirements, the most suitable algorithms can be determined.

3.3.1. Navigation algorithms overview
Tab.3.1 shows the algorithms divided in the corresponding categories. For the classes Methods based on a geometric model search, Probabilistic sampling-based algorithms and Artificial potential field type algorithms, the algorithms use similar search strategies. Each algorithm in these classes are improvements on the others or use the same methods with emphasis on different aspects. Highlighted in green are the three algorithms that represent a basic algorithms in each class. These are simple algorithms that perform according to the path planning approach of the corresponding class and are widely researched and described in papers.

The algorithms highlighted in red are discarded from further research. This is either because they perform outside the described algorithm scope proposed in Sec.3.2.1 or exceed the basic design fundamentals as described in Sec.2.4.2. This results for example in discarding the miscellaneous algorithm class. The algorithms in this class require the cooperation with another algorithm and are therefore not suitable as stand alone algorithms. Moreover, all animal based bio-inspired intelligence algorithms are very much focused on swarm behaviour of the animal colonies. As Sec.3.2.1 explains, the navigation and swarm capabilities will not be programmed in the same algorithm. Therefore the animal based bio-inspired intelligence algorithms will also not be considered in the further search towards a navigation algorithm. The reason for discarding the other highlighted algorithms can be found in the detailed explanation of the algorithms, which is presented in App.A.2.

This leaves 16 possible algorithm for the implementation. As this is too much for further research a distinguish between them is still to be made.

3.3.2. Base algorithms
Choosing a base algorithm in each of the four left over classes reduces the amount of algorithms for further research, while still researching the navigation characteristics of each class. Furthermore, common algorithm pitfalls can be observed in the early stage of algorithm development. When a pitfall

occurs, it can be determined if either this can be fixed by improving the base algorithm or if the algorithm should be rejected for the LZ application. The classes Methods based on a geometric model search, Probabilistic sampling-based algorithms and Artificial potential field type algorithms have an obvious base algorithm which is highlighted in green. The Bio-inspired Intelligence Algorithm category is a bit different from the other three. This is less straight forward as almost all algorithms have individual approaches that are not similar to each other. Furthermore these algorithms are less commonly used and substantiated as navigation algorithms compared to the other three classes. For these reasons the choice is made to exclude these algorithms for further research as well. The following three algorithms are therefore chosen as the basic navigation algorithms that are applicable for the LZ mission and project.

- A* Algorithm,
- Rapidly Exploring Random Trees,
- Artificial Potential Field.

Both the A* and the Artificial Potential Field algorithms make use of the grid model as environmental model. This means they both work with a predefined grid and accuracy. The Rapidly Exploring Random Trees algorithm makes use of the Probabilistic Roadmap Method, where each grid point is randomly determined in the search field. The accuracy of this method can be defined as the maximum expand distance between a known point and the next defined random point. The chosen grid and/or accuracy plays an important role in the performance of all three algorithms.

The stated problem in Sec.2.4 is specifically focused on path planning and obstacle avoidance, which is a subset of the navigation problem. The proposed basic algorithms are also limited to path planning and obstacle avoidance. Obstacle avoidance can also be seen as a subset of path planning. From now on the LZ navigation problem will therefore be addressed as a path planning problem. The solution to this problem will therefore be a path planning algorithm.

Table 3.1: Classification of all considerable algorithms

Methods based on a geometric model search	**Probabilistic sampling-based algorithms**	**Artificial potential field type algorithms**	**Bio-inspired Intelligence algorithms**	**Miscellaneous algorithms**
A* Algorithm	Rapidly exploring random tree	Artificial potential field	Genetic Algorithm	Artificial Neural Network
D* Algorithm	Probabilistic Roadmap Method	Bug algorithm	Differential Evolution	Fuzzy
Dijkstra Algorithm		Vector-polar histogram	Imperial competition algorithm	Cased-Based Learning Method
Level set Method			Simulated annealing algorithm	Behaviour decomposition method
Fast marching algorithm			Q-learning/ dynamic programming	Rolling Window Algorithm
Boustrophedon Decomposition Algorithm			Tabu Search	SLAM
Internal Spiral Algorithm			Swarming animal algorithms	
			Reinforcement learning	
			Deep Reinforcement learning	

4

Basic path planning algorithms

Ch.3 states A* , Rapid Exploring Random Trees and Artificial Potential Field algorithm as the three possible basic path planning algorithms for LZ. This chapter will elaborate on the performance of these three algorithms and will conclude which of the three is the most suitable base algorithm for the LZ application. The elaboration is done by a detailed introduction to the functioning of the algorithm, an extensive analysis and a simulation based results comparison of quantitative and qualitative performance metrics. These base algorithms can and probably must be improved before a proper implementation in LZ. Therefore, last but not least, this chapter will elaborate on the potential improvements of each of the three base algorithms.

4.1. Algorithm functioning

This Section will explain in detail how each algorithm operates and which mathematical functions are used to determine the rover paths. Each algorithm is therefore written out in pseudo code(a representation of the algorithm in words) and a detailed explanation about the pseudo code is given. The pseudo code will solely describe the functioning of the algorithm and no additional code that is used for simulating the rover or simulating the algorithm. Some variables and functions have the same purpose in multiple algorithms. These common variables and functions are described in Tab.4.1 and Tab.4.2 respectively.

Table 4.1: Variables that are commonly used in the pseudo code descriptions

Variables	Description
$Node$	A point that is either on a discreet grid or randomly generated. Nodes are the only points that can be used as path points by the algorithms.
N_{Start}	The start node of the rover. This is the point where the path search is starting.
N_{Goal}	The goal node of the rover and therefore the point where the path search should stop
$ObstacleList$	List that holds all obstacles that are known or detected by the rover. It has the following structure :(X-coordinate, Y-coordinate, radius in meters)
$RoverRadius$	Radius of the rover in meters This includes the safety boundry that should be created around the rover
$AreaBounds$	The area size which contains the N_{Start} and N_{Goal} It has the following stucture[min-x, max-x, min-y, max-y]
$GridSize$	Distance between the dicreet grid points in meters. Deacreasing the $GridSize$ results in a more accurate path.
$ExtendDistance$	Maximum length the algorithm can extend with the addition of one path point. This is somewhat similar to the $GridSize$.
$MaxIterations$	Maximum number of times the algorithm may run to find a path. This is used to prevent a loop of infinity tries.

Table 4.2: Functions that are commonly used in the pseudo code descriptions

Functions	Description
$RoverMotionDirections$	The 8 possible directions the rover can go when on a node. These 8 directions are the adjacent nodes
$AddDirectionToNode$	This function defines the possible next node by adding the motions to the current node
$AddNodeToList$	This function will add the next path node to a set or list. This set or list will be used to form the final path.
$MakePathFromNodeList$	This function transforms a set or list with path points to an actual path. This path can than be used by the rover.
$CalculateDistance$	Function to calculate the Euclidean distance between two nodes.

4.1.1. A* Algorithm

The A* algorithm is presented in Alg.1 as pseudo code. The explanation of the pseudo code, and therefore the actual code, will be given in this chapter. A flowchart representation of the A* algorithm is depicted in Fig.4.1. [16] and [8] present a similar code structure for the A* algorithm, while [16] describes an A* algorithms for beginners to explain the basic principles of A*. [8] presents an improved A* algorithm in the perspective of modern computer games.

Inputs and outputs

The require line in Alg.1 shows the needed inputs for the algorithm to function properly. The $AstarNodes$ are defined as a separate class. Each $AstarNode$ object holds a X and Y position, a $Cost$ and a $ParentIndex$. The $ParentIndex$ is the previous node that is used to reach the current node. The variables in an $AstarNode$ object form the information that is needed for the A* path planning. The actual code has the ability to scale according to the $GridSize$, but to keep the pseudo code readable and structured this scaling is removed.

The output of the algorithm is a collision free path from the start node to the goal node. The path is presented as a list of X, Y coordinates from the goal node to the start node.

Map initialization

On line 1 from the A* pseudo code, the area bounds are used to initialize a matrix. This matrix is used to map and identify obstacles. First all matrix values are initialized to $False$, after which all nodes that collide with an obstacle are set to $True$. This $True$ value is determined by checking the distance between a node and all the obstacle centres. If the distance between a node and the the obstacle centre is smaller or equal to the $RoverRadius + ObstacleRadius$ than the node is in collision and set to $True$. This is shown between lines 2 and 9 of the pseudo code.

Path generation

The $LowestCostNode$ function finds the node in the list that has the lowest $cost$ value and makes this the $AstarN_{Current}$ in line 13 from the A* pseudo algorithm. The $cost$ is always calculated as shown in Eq.4.1. Where $g(n)$ is the exact cost from the $AstarN_{Start}$ to node n and $h(n)$ the estimated cost from node n to $AstarN_{Goal}$. $h(n)$ is chosen to be the Euclidean distance between node n and $AstarN_{Goal}$. The Euclidean distance is set to be the length of a straight line between the two given points. After determination of $AstarN_{Current}$, the $AstarN_{Current}$ is removed from the $OpenSet$ and added to the $ClosedSet$. If the $AstarN_{Current}$ is the same as the $AstarN_{Goal}$, the loop is terminated and a final path is generated.

$$cost(n) = g(n) + h(n) \tag{4.1}$$

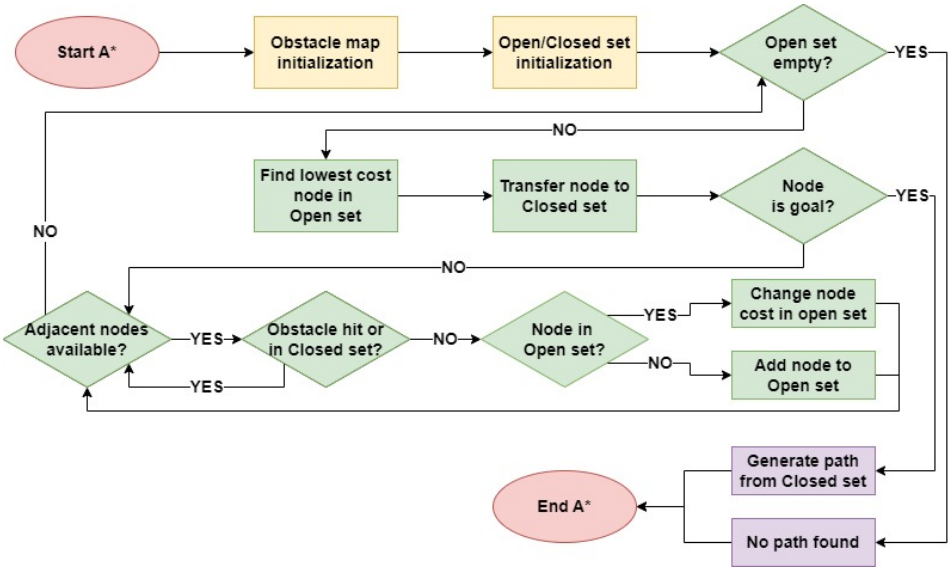


Figure 4.1: Flowchart of the A* algorithm. The initialization is denoted in yellow, path planning in green and the outputs in purple.

Algorithm 1 A* Algorithm [16]

Require: $AstarN_{Start}, AstarN_{Goal}, ObstacleList, RoverRadius, AreaBounds, GridSize$
Ensure: $Collision\ Free\ Path$
$ObstacleMap \leftarrow False \in AreaBounds_{X,Y}$
for $AstarN_{X,Y} \in ObstacleMap$ **do**
  for $Obstacle_{X,Y}, ObstacleRadius \in ObstacleList$ **do**
    if $CalculateDistance(N_{X,Y}, Obstacle_{X,Y}) \leq RoverRadius + ObstacleRadius$ **then**
      $ObstacleMap_{X,Y} \leftarrow True$
      break
    end if
  end for
end for
$OpenSet \leftarrow [AstarN_{Start}]$
$ClosedSet \leftarrow \emptyset$
while $OpenSet \neq \emptyset$ **do**
  $AstarN_{current} \leftarrow LowestCostNode(OpenSset)$
  $OpenSet \leftarrow RemoveNodeFromList(AstarN_{current}, OpenSet)$
  $ClosedSet \leftarrow AddNodeToList(AstarN_{current}, ClosedSet)$
  if $NodesAreSame(AstarN_{current}, AstarN_{goal})$ **then**
    $ClosedSet \leftarrow AddNodeToList(AstarN_{goal}, ClosedSet)$
    $Path \leftarrow MakePathFromNodeList(ClosedSet)$
    return $Path$
  end if
  for $RoverMotionDirections$ **do**
    $AstarN_{next} \leftarrow AddDirectionToNode(AstarN_{current}, RoverMotionDirection)$
    if $ObstacleMap[AstarN_{next(X,Y)}]\ is\ True$ **then**
      continue
    end if
    if $AstarN_{next} \in ClosedSet$ **then**
      continue
    end if
    if $AstarN_{next} \notin OpenSet$ **then**
      $Open\_set \leftarrow AddNodeToList(AstarN_{next}, OpenSet)$
    else
      if $NodeCost(OpenSet[AstarN_{next}]) > NodeCost(AstarN_{next})$ **then**
        $NodeCost(OpenSet[AstarN_{next}]) = NodeCost(AstarN_{next})$
      end if
    end if
  end for
end while
return $False$

4.1.2. RRT Algorithm

The Rapidly Exploring Random Trees Algorithm is presented in Alg.2 as pseudo code. Variables and functions that are shown in Tab.4.1 and Tab.4.2 are again used as common knowledge. Fig.4.2 depicts the Rapidly exploring Random Trees (RRT) algorithm in a flowchart representation. [33] presents a similar RRT base algorithm that can be used for robot path planning. In addition the paper presents a significant improvement in computation time with the so called Node-Control RRT algorithm. This improvement is neither considered in this thesis or pseudo code.

Inputs and outputs

For the RRT algorithm the $RRTNode$ class is used, which also has a X, Y and a $ParentIndex$ variable. Additional to these three variables, the $RRTNode$ class also holds the $PathX$ and $PathY$ variables. These are used when the $ExtendDistance$ is different from the desired path resolution. For example, the $ExtendDistance$ is taken as two meters but the output needs to have a resolution of one meter. In this case the $ExtendDistance$ is split up in to parts of one meter and consists of three coordinates instead of two. The three sub-nodes are now saved in $PathX, PathY$. As the other base algorithm only have the $gridsize$ to represent the resolution, the RRT algorithm will also be compared with only the $ExtendDistance$ as resolution. This insures a fair comparison between the algorithm and means that the $PathX, PathY$ variables are not used in the base algorithm.

The output is again defined as a collision free path from the start node to the goal node. The path is presented as a list of X, Y coordinates from the goal node to the start node.

Path generation

The path generation starts on line 2. The $GetRandomNode$ function creates a random $RRTNode$ within the given area bounds. The closest available node that is previously generated is determined by he $NearestNodeFromList$ function. Obviously the likelihood of a random node being within the $ExtendDistance$ is very small, therefore when the $RRTN_{rand}$ falls outside this $ExtendDistance$ the random node is used as a guidance for the direction of the new node. A new node($RRTM_{new}$) is calculated by using the $RRTN_{random}$ as direction and the $ExtendDistance$ as maximum distance from $RRTN_{nearest}$. This $RRTM_{new}$ is used for further calculations and connected to $RRTN_{nearest}$. Whenever the goal is within the $ExtendDistance$ the $Extend$ function is used to create $RRTN_{final}$, which ends the path search

Collision check

The $CollisionFree$ function checks if a node is within the obstacle bounds and is further explained on line 19 to 25 in Alg.2. Where again the Euclidean distance is used between the centre of an obstacle and the $RRTNode$. This distance should be larger than the $ObstacleRadius$ and $RoverRadius$ combined to ensure a collision free path. If there is a collision between the node and an obstacle, the node is discarded from the following path searches.

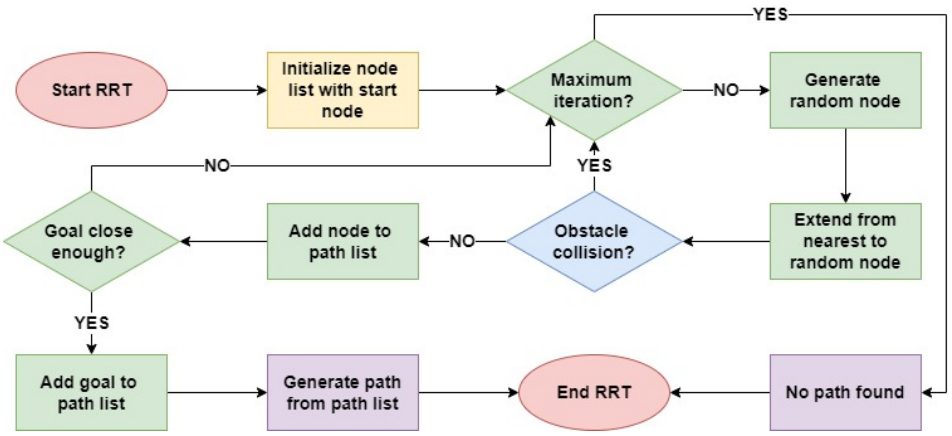


Figure 4.2: Flowchart of the RRT algorithm. The initialization is denoted in yellow, path planning in green, use of external functions in blue and the outputs in purple.

Algorithm 2 Rapidly exploring Random Trees Algorithm [33]

Require: $RRTN_{Start}, RRTN_{Goal}, ObstacleList, RoverRadius, AreaBounds, ExtendDistance, MaxIterations$
Ensure: $Collision\ Free\ Path$
1: $NodeList \leftarrow [RRTN_{start}]$
2: **for** $MaxIterations$ **do**
3: $RRTN_{rand} \leftarrow GetRandomNode(AreaBounds)$
4: $RRTN_{nearest} \leftarrow NearestNodeFromList(RRTN_{rand}, NodeList)$
5: $RRTN_{new} \leftarrow Extend(RRTN_{nearest}, RRTN_{random}, ExtendDistance)$
6: **if** $CollisionFree(RRTN_{new}, ObstacleList, RoverRadius)$ **then**
7: $AddNodeToList(RRTN_{new}, NodeList)$
8: **end if**
9: **if** $Distance(RRTN_{new}, RRTN_{Goal}) < ExtendDistance$ **then**
10: $RRTN_{final} \leftarrow Extend(NodeList[length - 1], RRTN_{goal}, ExtendDistance)$
11: **if** $CollisionFree(RRTN_{final})$ **then**
12: $Path \leftarrow MakePathFromNodeList(NodeList)$
13: **return** $Path$
14: **end if**
15: **end if**
16: **end for**
17: **return** $False$
18:
19: $CollisionFree(RRTNode, ObstacleList, RoverRadius)$:
20: **for** $Obstacle_{X,Y}, ObstacleRadius \in ObstacleList$ **do**
21: **if** $Distance(RRTNode_{X,Y}, Obstacle_{X,Y}) \leq RoverRadius + ObstacleRadius$ **then**
22: **return** $False$
23: **end if**
24: **end for**
25: **return** $True$

4.1.3. APF Algorithm

The Artificial Potential Field algorithm is presented in Alg.3 as pseudo code. Again the common variables and functions from Tab.4.1 and Tab.4.2 are used.

The first Artificial Potential Field algorithm for path finding was presented by [13]. The algorithm as described in Alg.3 has the same structure, functioning, pitfalls and benefits. [18] present the base algorithm in a similar way as Alg.3, but also present a possible improvement for robot path planning.

Inputs and outputs

The new node class is defined as $APFNode$ and holds just two variables, the X and the Y coordinates of the node. No further information is needed for proper functioning of the Artificial Potential Field (APF) algorithm. The output is again a collision free path from the $APFN_{Start}$ to $APFN_{Goal}$.

Map initialization

The potential map is initiated on line 1 with zero's and has the size that is given by the $AreaBounds$ variable. The $CalculateAtractivePotential$ calculates the attractive potential. The attractive potential is defined as zero at the goal and becomes more positive when moving away from the goal. The $CalcRepulsivePotential$ calculates the repulsive potential and is defined as becoming positive large when in and around an obstacle. Finally the $TotalPotential$ function adds these two potentials which results in a total potential. This is determined for each node in the $PotentialMap$.

Furthermore a list $Path$ is made of type $APFNode$ to hold the nodes from the final path on line 2. A second list, $PreviousNodes$, is made for the $APFNode$ type on line 3. However, this is a double-ended queue list. This means that variables in the list can be added or removed from either the beginning or the end of the list. At the end of this section the relevance of this list will be explained.

Path generation

The $RoverMotionDirections$ is similar to the one presented in Sec.4.1.1, but the cost of the nodes is not used for the APF algorithm. Instead the next node is determined by checking the potentials of all adjacent nodes. The $GetPotential$ function returns the potential value from an $APFNode$ in the $PotentialMap$. The next node is defined as the adjacent node with the lowest potential. If this node is closer to the $APFN_{Goal}$ than the $Gridsize$ the loop is terminated and the $Path$ is returned.

Local minimum detection

It can occur that the $PotentialMap$ has two or three nodes next to each other with the same potential value. In this case the algorithm will define its path back and forth between these two or three nodes and starts oscillating. In other words the algorithm is stuck in a local minimum. This can happen in front of an obstacle for example. To prevent the algorithm from crashing, a local minimum detection function is written. If a local minimum is detected the algorithm will stop planning and fails to find a path. The $NoLocalMinimum$ function is presented on line 34, where a list is kept of the three previous nodes. Every time the fourth node is added to this list, the $RemoveFirstItemFromList$ removes the first added node from the end of the list. For this reason the double ended list is used, such that a $APFNode$ can be added from one side and removed from the other side of the list. The $NodeInListIsDouble$ function checks if the three presented nodes in $PreviousNodes$ are the same. If this is the case, than a local minimum is detected and the algorithm will terminate.

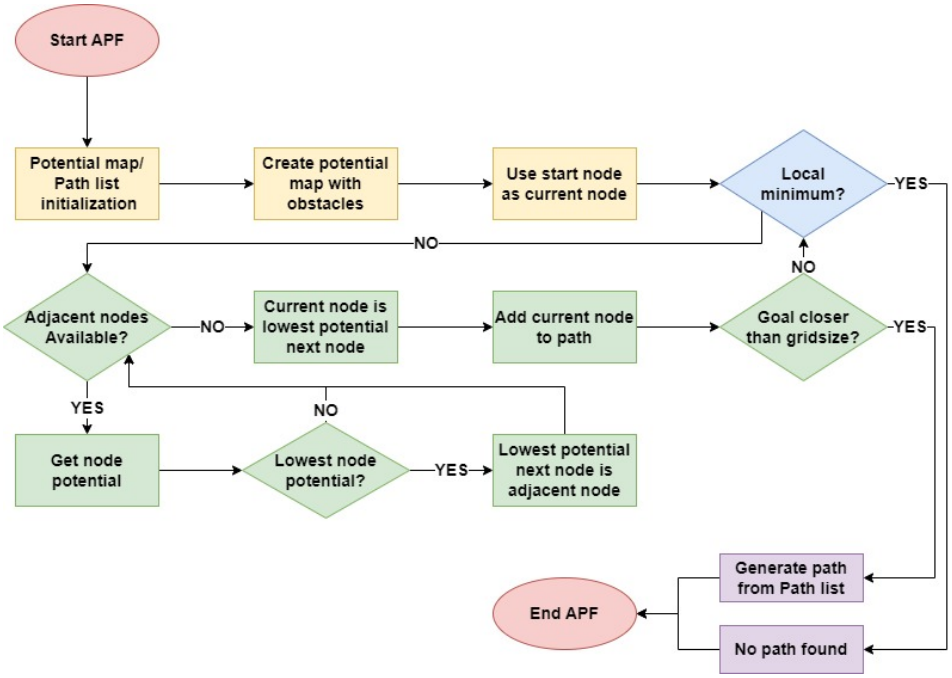


Figure 4.3: Flowchart of the APF algorithm. The initialization is denoted in yellow, path planning in green, use of external functions in blue and the outputs in purple.

Algorithm 3 Artificial Potential Field Algorithm [13]

Require: $APFN_{Start}, APFN_{Goal}, ObstacleList, RoverRadius, GridSize, AreaBounds$
Ensure: $Collision\ Free\ Path$

$PotentialMap \leftarrow 0 \in AreaBounds_{X,Y}$
$Path \leftarrow [APFN_{start}]$
$PreviousNodes \leftarrow deque()$
for $X, Y \in Area$ **do**
    $PositivePotential \leftarrow CalculateAttractivePotential(X, Y, APFN_{Goal})$
    $NegativePotential \leftarrow CalculateRepulsivePotential(X, Y, ObstacleList, RoverRadius)$
    $PotentialMap \leftarrow TotalPotential(X, Y, PostivePotential, NegativePotential)$
end for
$Distance \leftarrow CalculateDistance(APFN_{Goal}, APFN_{Start})$
$APFN_{current} \leftarrow APFN_{Start}$
while $NoLocalMinimum$ **do**
    $MinimumPotential \leftarrow \infty$
    for $RoverMotionDirections$ **do**
        $APFN_{possible\_next} \leftarrow AddDirectionToNode(APFN_{current}, RoverMotionDirection)$
        if $APFN_{possible\_next} \notin AreaBounds$ **then**
            $Potential \leftarrow \infty$
        else
            $Potential \leftarrow GetPotential(APFN_{Potential\_next}, PotentialMap)$
        end if
        if $MinimumPotential > Potential$ **then**
            $MinimumPotential = Potential$
            $APFN_{minimal\_possible\_next} = APFN_{possible\_next}$
        end if
    end for
    $APFN_{current} = APFN_{minimal\_possible\_next}$
    $Distance \leftarrow CalculateDistance(APFN_{Goal}, APFN_{current})$
    $Path \leftarrow AddNodeToList(APFN_{current}, Path))$
    if $Distance < GridSize$ **then**
        return $Path$
    end if
end while
return $None$

$NoLocalMinimum(APFN_{current})$:
    $PreviousNodes \leftarrow AddNodeToList(APFN_{current}, PreviousNodes))$
    if $length(PreviousNodes) > 3$ **then**
        $RemoveFirstItemFromList(PreviousNodes)$
    end if
    if $NodeInListIsDouble(PreviousNodes)$ **then**
        return $True$
    else
        return $False$
    end if

4.2. Algorithm analysis

To determine the most suitable algorithm for LZ, a performance comparison is completed. To compare the performance of the three base algorithms, performance metrics play a key role in defining which of the three algorithms is the most suitable. The performance of the three algorithms is dependent on the environmental factors that are present in the Lunar Zebro environment. Therefore the performance metrics and environmental factors are defined first. The impact of the environmental factors on the performance metrics for each algorithm is eventually evaluated with a simulation. This section will conclude with the explanation of the build and used simulation to obtain the results.

4.2.1. Performance metrics

A performance metrics is a way to analyze the strength and weaknesses of an algorithm. Different types of algorithms could have different types of performance metrics. For path planning algorithms in the scheme of Lunar Zebro the relevant metrics can be divided in two categories; quantitative metrics and qualitative metrics. A quantitative metric can be calculated and is defined as number and can therefore be compared directly. However, a qualitative metric is defined in a level of (im)possibility relative to other algorithms. The first three presented metrics are quantitative metrics and the succeeding metrics are qualitative metrics.

Path length

The first metric is the path length that is obtained by the algorithm. Different approaches used by algorithms have different ways to avoid obstacles and different ways to reach goals. Each approach comes with a different path length. The path length is an important performance metric as it plays a key role in the capabilities of a robotic swarm. As a rover can only travel a fixed amount of distance due to battery life or other factors, it is important this distance is used in an optimal way. During exploration missions the swarm could cover a larger area if the path length is optimized.

Planning time

The second metric is the computation time of the planning algorithm. This gives an indication on the computational complexity. The planning excludes time that is used for simulating and/or printing anything regarding the rover or the algorithm. It is strictly reduced to the time that is needed to obtain a path from the start to the goal while avoiding the known obstacles. Even if the path length is the same for different runs, it is not guaranteed that the planning time is the same for the different runs. This is due to the fact that this is highly dependent on scheduling, memory writing, interrupts, etc of the running machine or processor. This will also be the case in the Lunar Zebro microcontroller. To obtain a relevant value for this metric it is required to take a large amount of samples to get a realistic comparison between the algorithms.

Success/Fail ratio

In some cases the algorithm is not able to find a path at all. Succeeding or failing can be considered as an algorithm robustness measure. This success/fail ratio is also known as the reachability of an algorithm. The higher the success/fail ratio, the higher the reachability. Robustness or reachability is a relevant metric in the scheme of Lunar Zebro. When a rover is operating on its own, it is important that it is not failing to find a path. However, when the rover is operated in a swarm, the influence of one rover not finding a path to the goal is minor. When the majority of a swarm reaches the goal two choices could be made, either leave the stuck rovers behind or help them find the right path by telling them which path the succeeded rovers took. This way the robustness of a single rover in a swarm is less relevant compared to the swarm robustness. Looking from this perspective the algorithm should not create a situation where the rover is trapped in an unsolvable scenario.

Moving obstacles and target

Moving obstacles and targets are not relevant for LZ when on a single mission to the moon as the environment is static. However, when operating in a swarm, the swarm rovers can be seen as moving obstacles. Moreover, the swarm may be able to move the goal as the swarm can be relocating itself. This is the first qualitative metric as it depends on the algorithm how easy the implementation of moving obstacles and targets is.

Path length predictability
As path length is already one of the performance metrics this may seem less relevant. However, when the swarm is trying to cover a large area and must be able to come back to the base station, the rover batteries should allow this. If the path length is very wide spread and the return to base station rate should be 100%, than the maximum allowable distance to travel will always be on the low side of the possible path lengths. When the path length is unpredictable there is a chance that a part of the rovers are not able to find their way back to the base station. This unpredictability limits the possibilities of the swarm. A predictable path length is therefore a relevant qualitative performance metric.

Possibility to improve path by swarm behaviour
This metric is based on the fact that a part of the swarm did find the goal and a part of the swarm didn't. The rovers that succeeded can mark the path they walked such that the rovers that didn't succeed can start following this path. This is easily done by tempering the potential field in the APF algorithm but much more difficult to implement in the A* and RRT algorithms.

Swarm computation
Having multiple rovers also means the availability of multiple microcontrollers or ZPU's. Combining the computational power of the microcontrollers could make the path planning much easier for larger grids. As multiple rovers could detect obstacles it is possible for A*, RRT and APF to combine this obstacle information before planning the path and therefore share the processor workload. However, the A* and RRT algorithms need separate path planning for each rover as this is dependent on the position of the rover. The APF algorithm is making a potential map without the knowledge of the rover locations. As a result the APF navigation is done independently of where the rover is placed in the map. Therefore each rover could calculate a part of the map or improve the map. Combining these parts or improvements can result in reduced APF computation time for the complete swarm. Swarm computation is therefore beneficial for the APF algorithm but it has less impact on the A* and RRT algorithms.

4.2.2. Environmental factors
There are several factors that have influence on the above mentioned performance metrics. Again, the environmental factors with influence are depending on the environment and application. From the LZ perspective the applicable factors are discussed below.

Area size and gridsize
At first it seems that the area size and the gridsize are different factors. However, if you have a closer look at the functioning of the algorithms it becomes clear that they dependent on the amount of samples that have to be calculated. Making the gridsize two times smaller(more accurate) has the same effect on the algorithm as making the width and height of the area two times larger. This assumption comes with the minor side node that when the area size is changed, the obstacles, start and goal need to change proportional with the area size. If the gridsize becomes two times larger than the obstacle center X and Y coordinates and the obstacle size also become two times larger. Also, the start and the goal are placed next to the corners of the expended area. This ensures that the obstacle locations and sizes are relatively unchanged, such that only one factor(area size) can be analyzed. Therefore the gridsize will have a predefined and unchanged value and the obstacle centers and sizes will be scaled together with the area.

Obstacles size
The size of obstacles influences the complexity of the field that needs to be solved. The larger the obstacles, the less space for a path to be planned. Each algorithm is handling this factor different and it therefore is a relevant environmental factor.

Rover view distance
The maximal possible view distance is already constrained by the placed hardware in Lunar Zebro. However, maximal view does not necessarily means better performance. The optimal view distance has to be determined for each algorithm. In case the view distance must be decreased for better performance, it could be decreased in software.

Number of obstacles
Obviously changing the number of obstacles will change the path length and plan time. However, the relation between the obstacle scenarios and between the different algorithms is unclear. Changing the amount of obstacles shows a clear comparison between the algorithm performances in (non-)cluttered environments. The obstacles can be placed on predefined spots or can be placed randomly on the grid. When the field becomes cluttered due to the large amount of obstacles, the obstacle size should be reduced. With a cluttered environment of large obstacles the field will be fully covered, which would result in a 100% failure rate for all algorithms. In this case two factors are changed at once; size and number of obstacles. However, cluttering is an important environmental factor for comparison the algorithm performance. Therefore the choice is made to increase cluttering of obstacles and decrease the obstacle size simultaneously.

4.2.3. Simulation
To determine the affect of the environmental factors on the described performance metrics, tests need to be executed. These test could be done with the use of an actual rover that satisfies the requirements that describe LZ. However, these tests must be executed a significant amount of times to gain relevant data. As this would be extremely time consuming, it is chosen to build a simulation that executes the tests. The simulation can test all three algorithms and the affect of the environmental factors on the performance metrics. Not all performance metrics can be verified in a simulation as some of them are qualitative instead of quantitative. The qualitative performance metrics must be verified with a combination of the simulation results and the implementation possibilities.

Simulation environment
As described in Ch.2 the path finding algorithm should eventually be running on the ZPU. This requires a low level implemented coding language to ensure efficient and optimal usage of the available processing power. The implementation of the final algorithm should therefore be done in C or C++, which are both low level programming languages. The simulation is created to obtain relevant data and reduce testing time. As C or C++ are neither powerful languages for data processing it is chosen to make the simulation with a different coding language. The Python coding language on the other hand is a powerful language when it comes to data processing. For this reason it is chosen to code the simulation and the algorithms in Python. The final simulation is made in Python3 and runs on a local machine. The local machine is a $Linux$ based system that runs $Ubuntu\ 18.04.5$ with kernel version 5.4.0-120-$generic$. The used Integrated Development Environment (IDE) is Visual Studio Code (VSC) with the required Python extensions.

Simulation design
The simulation consists of three components that are shown in a simple 2D top view. This top view shows a field where the rover can move freely. Fig.4.4 shows an example of these field for all three algorithms.

The first component is the map which shows three elements; the starting point, the goal point and the obstacles. The start is presented as a black star and the goal is presented as a purple star. It is chosen that the obstacle are presented by blue circles. Despite the fact that in reality obstacles are almost never actual circles, it still satisfies its purpose. As long as the circle is larger than the actual obstacle plus $RoverRadius$, it denotes the desired safety area around an obstacle. A circle is chosen because it simplifies the representation of obstacle for both the algorithm and the simulation. A simplification for the algorithm results in a reduction of computation time and therefore computational complexity. The obstacles shown by the map are not yet known for the rover as they are not yet detected.

Second, the characteristics of the rover are shown in the simulation. The rover characteristics consist of three elements; the rover with a defined view area, the planned rover path and the actual walked path. The rover is depicted by a single dot with a view area. The view area is shown as a blue triangle in front of the rover. As soon as the blue triangle interferes with an obstacle, the obstacle becomes detected and is saved in the know obstacle list. As long as an obstacle is not detected the rover plans a path without the knowledge of this obstacle. The planned path is shown by a red line between the start and the goal. The actual walked path is represented by the yellow dots on the red lines. The reason that these two path are presented differently is due to the generation of a new path when an obstacle is detected. The new path may not be able to continue on the old planned path, as

an obstacle is blocking the way. Therefore the actual walked path can differ from the initially planned paths.

The third component is the possibility to show the functioning of the path planning algorithm. For A* this means showing the considered search points by light blue crosses. For the APF algorithm this means showing the potential field and changes in potential when an obstacle is detected. A higher potential results in a darker colour. Detected obstacles are therefore dark circles, while the goal is the lightest point in the map. The RRT algorithm shows the random search tree that is generated from the start to the goal. The search tree is shown in green.

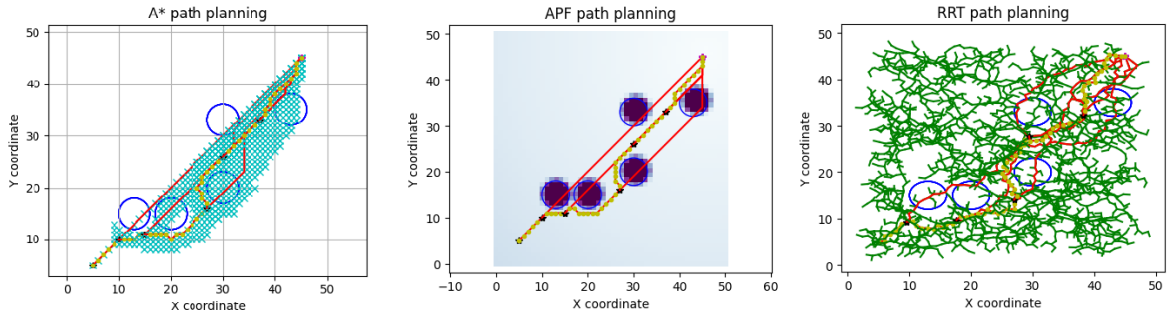


Figure 4.4: Overview of path planning simulations

standard conditions
To set a benchmark, a set of standard environmental factors is chosen such that the influence of one environmental factor can be determined easily. The following parameters are chosen to be the standard conditions:

- $Node_{Start}$: [5,5] in meters
- $Node_{Goal}$: [45,45] in meters
- $AreaBounds$: 50x50 meters
- $GridSize$: 1 meter
- $ViewDistance$: 3 meters
- $RoverRadius$: 0 meters
- $ObstacleList$(x,y,Obstacle radius[m]): (13, 15, 3), (20,15, 3),(30,20, 3),(30, 33, 3),(43, 35, 3)

Unless specified otherwise, every simulation has only one of the above factors changed. This way the influence of the individual environmental factors on the algorithm performances can be determined. The simulation is done 500 times for each algorithm and for every metric and environmental factor combination. When running the same simulation 500 times the results and distribution of results are consistent. 500 runs for each performance metric and environmental factor combination gives therefore a reliable outcome, but meanwhile has reasonable simulation times. During the experiments the $RoverRadius$ is set to zero. The radius of the rover is relatively small compared to the obstacles. Furthermore, the rover radius is just to be added to the obstacle radius as a ”no-go” area, which in practice has the same affect as changing the obstacle radius.

Random obstacles
One of the environmental factors is defined as the change of the obstacle scenario. This is the only performance factor that results in different maps for each run of the simulation. The amount of obstacles and their predefined size will be considered as the obstacle scenario. The actual placement of the obstacles will result in the obstacle map. If, for example, the simulation runs two times with five random placed obstacles the scenario is not changing over the runs. However, due to the random placement of the obstacles, the first map will be different compared to the second map. As each scenario is simulated

500 times, each scenario will also have 500 different randomly created obstacle maps. However, when the simulation is needed again for comparison of algorithms, these maps must be the same. For this specific reason the random generated location of the obstacles is saved in a file for multiple or future use.

4.3. Quantitative performance

The output of the simulation are the results for the quantitative performance metrics. This section will elaborate on all results gained from the simulation and will present an extensive Monte Carlo analysis on the individual environmental factor and performance metric combinations.

4.3.1. Quantitative table

The quantitative performance of each algorithm for all environmental factors is shown in Tab.4.3. The results are the mean values of the 500 simulations, as described in Sec.4.2.3. Notice that every first row of the environmental factors is the standard condition configuration. This is to verify the consistency of the system as the simulations are run over a longer period. Some of the simulations were not able to find a path, either for all 500 runs or a part of it. This is shown in the table as a ratio of fails from the total of 500 simulations. The higher the failure the lower the reachability of the algorithm. If a part of the simulations failed, the performance metric value is still shown. However, this value is taken as the mean of only the ”success” simulations. These results are therefore less reliable as they are deducted from fewer simulations.

4.3.2. Quantitative analyses

Some results from Tab.4.3 are further detailed down to analyse algorithm performance. The analysis will be substantiated with a Monte Carlo analyses or a boxplot representation of the dataset obtained by 500 runs. The Monte Carlo and the boxplot analysis on some specific environmental factors combined with the data from Tab.4.3, will form the conclusions on the quantitative performance of algorithms. The Monte Carlo figures and boxplot figures in this section will all have the same layout. The layout of these figures will therefore only be explained once, combined with the first introduction of the figure. The relevant results and their corresponding analysis will be discussed. A boxplot analysis of all different environmental factors is presented in App.B.1

Standard conditions

The applicable standard conditions are described in Sec.4.2.3. The corresponding performance results are shown in Fig.4.5 for all three base algorithms. The first row of Fig.4.5 shows all data corresponding with the A* algorithm. From left the right it shows, a histogram of the planning time in milliseconds, a histogram of the path length in metres and at the end a fail versus success ratio diagram. A fail means that there was no path found from start to goal and success means there was a path found from start to goal. On the second and third rows the same histograms are presented for the original APF and RRT algorithms respectively. As the x-axis are scaled to the largest measured value of all three algorithm, the histogram can become dense. For clarification an inset figure is placed at the histograms that show an extremely dense distribution.

Both for A* and APF the path length stays identical in all 500 simulation, caused by a non changing obstacle map over all 500 iterations. On every part of the path the algorithm makes the same choice. The planning time for both algorithms shows a Poisson distribution. A high density on the lower planning times and a majority of the outliers are located on the high side of the planning time. The histograms show that A* converges faster in a solution than APF. The RRT algorithm has a random path planning element, this results in different path lengths for different runs. This random path behaviour also results in an almost normal distributed planning time. Both the A* and APF algorithms have a planning time distribution that is not exceeding the first quartile of the normal distributed RRT planning time. This shows that even when considering the outliers, A* and APF have much better planning time results. Last but not least all algorithms have a success rate of 100% in the standard conditions.

Table 4.3: Base algorithm results for every performance metric and environmental factor combination.

Environment Factor		Path length [m]			Planning time [ms]		
Factor	**Value**	**A* algorithm [16]**	**RRT algorithm [33]**	**APF algorithm [13]**	**A* algorithm [16]**	**RRT algorithm [33]**	**APF algorithm [13]**
Area	**50x50**	63.25	76.19	64.67	91	234	142
	25x25	30.87	40.51	34.63	12	90	38
L · W	**100x100**	131.38	164.13	133.14	201	1748	441
[m]	**200x200**	275.14	338.28	278.66	1562	8930	1202
object	***3***	63.25	76.46	64.67	89	240	141
size	***1***	56.57	71.99	57.15	18	199	75
	5	67.6	83.69	71.60	108	284	108
radius	***8***	156.61	176.57	500/500 Fails	194	467	500/500 Fails
[m]	***10***	500/500 Fails	500/500 Fails	500/500 Fails	500/500 Fails	500/500 Fails	500/500 Fails
	3	63.25	75.70	64.67	89	238	139
View	***1***	67.15	76.04	72.08	73	197	105
distance	***5***	62.43	76.09	64.67	96	278	139
[m]	***8***	61.84	75.32	64.67	122	297	143
	10	61.25	76.03	64.67	142	323	147
Number	***Original size = 3***	63.25	76.08	64.67	90	245	142
of	***0***	56.57	70.09	56.57	5	55	8
obstacles	***5 random size =3***	58.57	74.47	127/500 Fails 59.66	30	187	127/500 Fails 59
size: radius [m]	***10 random size = 3***	61.07	79.44	280/500 Fails 60.95	68	325	280/500 Fails 107
	15 random size = 3	64.66	84.80	376/500 Fails 63.04	118	469	376/500 Fails 187
	15 random size = 1	57.31	74.27	98/500 Fails 59.15	38	335	98/500 Fails 132
	30 random size = 1	57.88	77.6	250/500 Fails 61.2	88	616	250/500 Fails 317
	60 random size = 1	59.3	1/500 Fails 83.02	466/500 Fails 63.98	241	1/500 Fails 1197	466/500 Fails 805

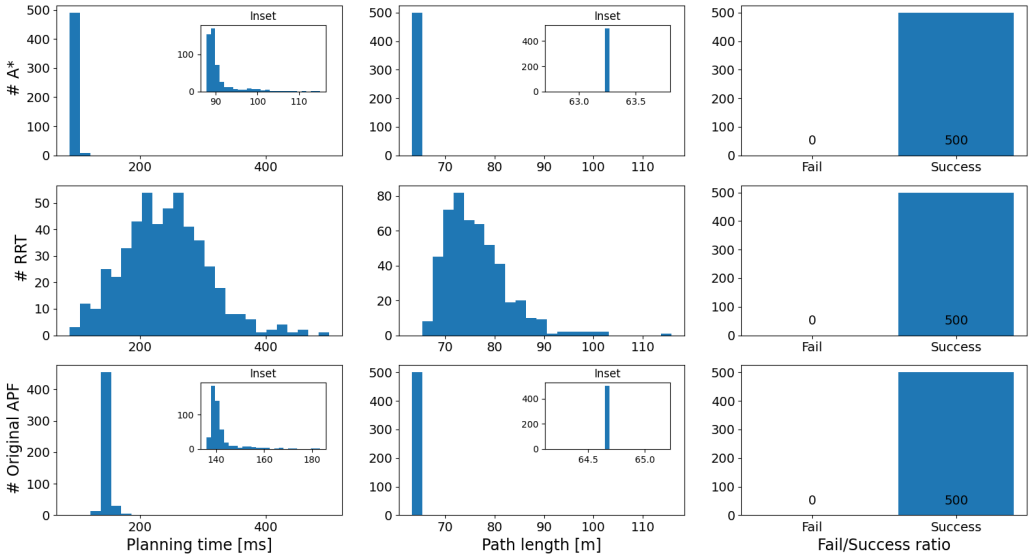


Figure 4.5: Algorithm performance in standard conditions(500 runs)

30 random obstacles

The second Monte Carlo analysis is made with the data of one of the obstacle scenario's. Placing random obstacles has a significant influence on the performance of the three algorithms. The performance table shows that the APF algorithm does not have a 100% success rate when the obstacles are placed randomly. When increasing the amount of obstacles, also the RRT algorithm fails to find a path in some of the simulations. As the obstacle maps are changing during the runs, path lengths of the A* and APF algorithms are also changing. However, the path lengths of A* and APF algorithms are lower and more predictable. The APF planning time becomes more distributed when the complexity of the obstacle map is increased.

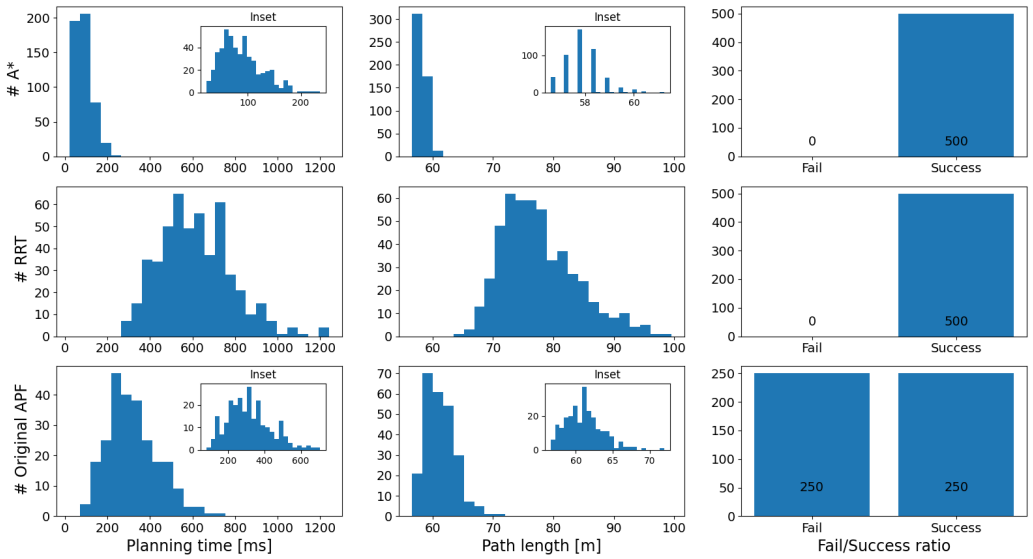


Figure 4.6: Algorithm performance on obstacle scenario with 30 random placed obstacles(500 runs)

Area 200x200
The successive relevant results are obtained with the area of 200x200 simulation. These results are shown with the help of boxplots and bar plots. Fig.4.7 shows the results for an area with size of 200x200 grid points. From left to right the figures present the planning time boxplots, path length boxplots and success/fail ratio bar plots for the three base algorithms. All three algorithm have more difficulties to form a path from start to goal. The planning time is much longer than the standard conditions planning time. Another interesting observation is the lower planning time of APF compared to the planning time of A*. A larger, and therefore more difficult area, results in APF being less computational heavy than the A* algorithm. With smaller and simpler area's, the A* algorithm outperforms the original APF algorithm. The RRT algorithm has the worst performance with extreme outliers in both planning time and path length. Again all algorithms have a success rate of 100%.

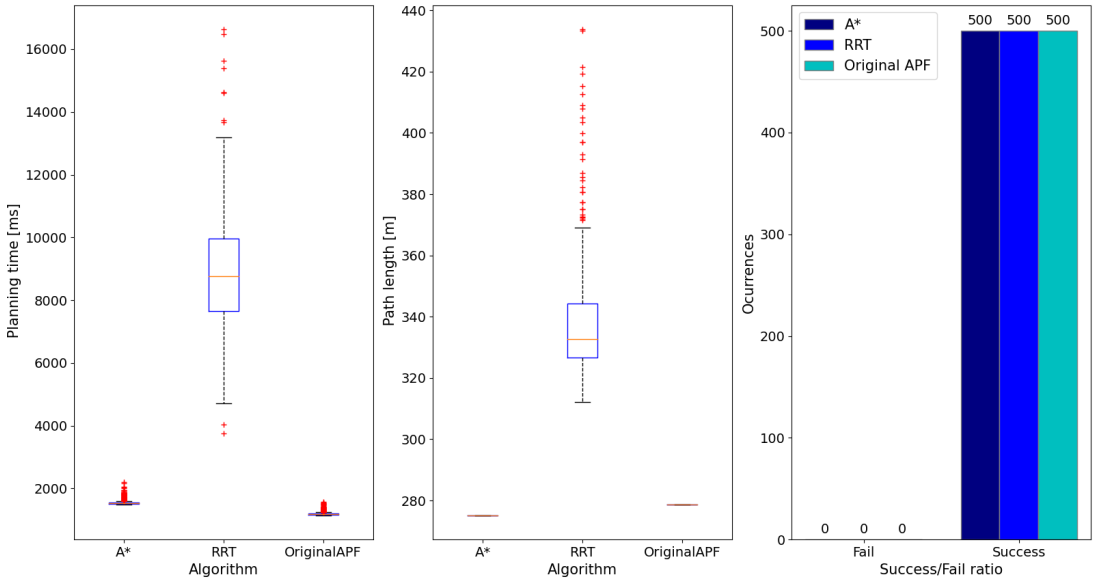


Figure 4.7: Algorithm performance with area of 200x200

Object radius 8 meters
From Tab.4.3 it becomes clear that increasing the obstacle size also increases the path length and the planning time for all algorithms. Increasing the object size results in an increasing complexity of the maps. When the object radius is set to five meters, the original APF algorithm performs similar to the A* algorithm regarding planning time. It again holds that in more difficult environment, the original APF algorithm shows more efficient results. However, an obstacle radius of eight meters becomes too difficult, the APF algorithm is not able to find any path. Fig.4.8 shows the results of the algorithm performance for an object radius of eight meters. The path length boxplot of the RRT algorithm shows a small amount of paths that are clearly shorter than the rest. This is caused by the random choice of the path, which is by coincidence going the right way around the large obstacles. This is not seen in the A* results, which still have one single path length.

Viewdistance 8 meters
Tab.4.3 shows that increasing the view distance does not automatically imply that the performance of the algorithms increase. The planning time of the algorithm even increases with large view distances. From a view distance of five meters, the path length is not improving significantly for either of the three algorithms. A view distance of one meter, on the other hand, results in path lengths that are unnecessary long. The view distance of eight meters is further displayed in Fig.4.9. The boxplots show that this relative large view distance is creating high outliers in the planning time, while the path length is barely improved. A view distance of three meters is therefore optimal for all three algorithms.

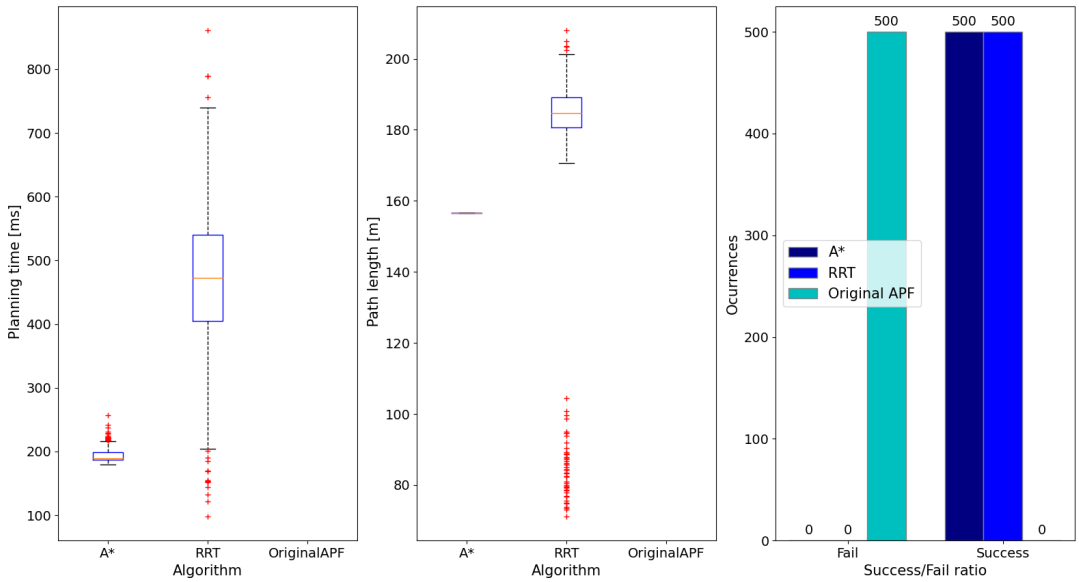


Figure 4.8: Algorithm performance with a object radius of 8 meters

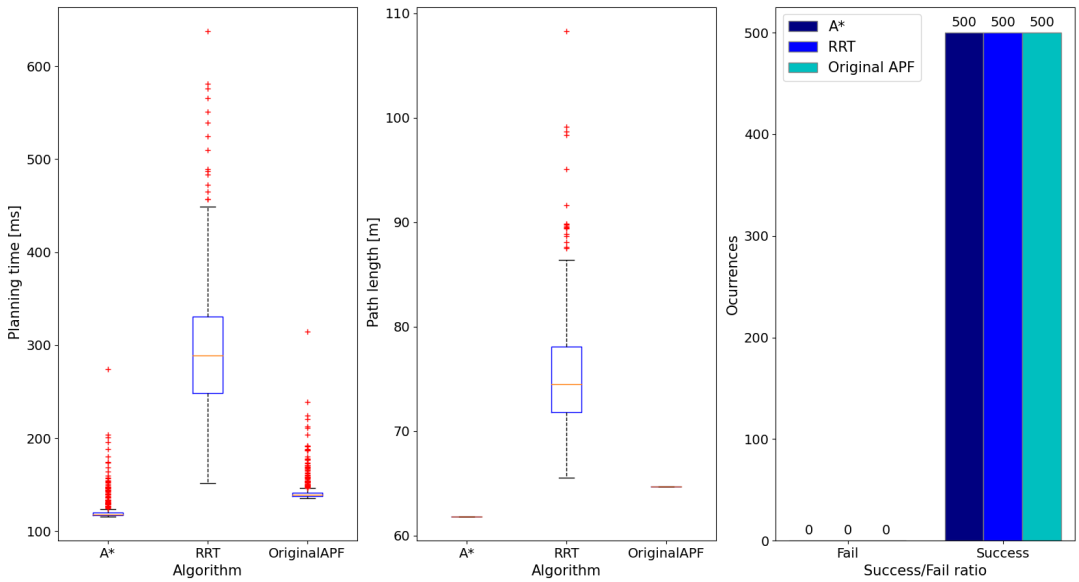


Figure 4.9: Algorithm performance with a view distance of 8 meters

4.4. Algorithm improvements

Choosing the base algorithm does not imply the algorithm is optimized for the Lunar Zebro application. This section will elaborate on the possible improvements that could be made on the path finding algorithms. The improvements are based on the results in Sec.4.2 and the requirements for LZ shown in Sec.2.4. Identifying the improvements gives the possibility to define the qualitative performance metrics of the algorithms.

4.4.1. A* path planning improvements for Lunar Zebro

To fit in the Lunar Zebro specifications, several improvements could be made on the A* algorithm. Some of these improvements are already defined in literature(Ch.3) and are proposes as new algorithms.

D* lite and Field D*

D* lite and Field D* are two improved algorithms based on the classic A* algorithm. These two are mainly used for robotic path planning in unknown environments. They are optimized on planning time and efficiency and outperform A* in unknown environments. This could be a relevant improvement looking at the environment conditions for Lunar Zebro. Extensive literature research towards their application is needed to determine which of the two would match with the LZ requirements.

Gridsize adaptability

Knowing that Lunar Zebro will have to travel at least 200 meters on the moon and that the step size accuracy needs to be around 5 centimeter (Sec.2.3.3), the grid becomes relatively large. Planning the whole rover path with an accuracy of 5 centimeters, while the path is most likely to change due to unknown obstacles, is quite a waste of computational power. Looking at the Cell Decomposition Approach as described in Sec.3.1.1, it could be possible to only split up a specified area around the rover in a grid with an accuracy of five centimeters. Outside this specified area the accuracy could be smaller, which results in less computation time.

Outlier prevention

Looking at the Monte Carlo analysis done on the A* simulations in Sec.4.3.2, the computational planning time has some extreme outliers which would ideally be prevented. This should be done by tracking the exact reason for this outliers and use prevention measures to this rare cases.

Weighted grid points

In some cases the algorithm can go right or left from an obstacle without a big difference in path length or planning time. From the MoSCoW analysis presented in Sec.2.4.2, it becomes clear that shadow avoidance for optimal solar energy yield would be a ”could have”. This can be achieved with weighted grid points that give a negative weight to shadowed area’s. Furthermore the weighted grid point could be used to give a negative weight on grid points located in difficult terrain. This ensures the avoidance of difficult terrain when possible.

Moving obstacles and goal

Looking at the swarm capabilities of Lunar Zebro it could be beneficial to have the possibility of moving obstacles and a moving goal. The moving obstacles could be other rovers in the swarm and the goal could be set and changed by the swarm. This way the swarm could constantly update the goal without the necessity of replanning the complete rover path.

4.4.2. APF path planning improvements for Lunar Zebro

Also the Artificial Potential Field algorithm comes in several variants that could benefit Lunar Zebro. Each variant is addressing different problems and improvements. The relevant improvements that can be achieved are summed up below.

Escape local minima

The success/fail ratio in some cases is very low compared to the other algorithms. This is caused by the algorithm that finds itself being stuck in a local minimum. This is causing a lot of path fails, while the solution could be as simple as: always go left or right when stuck at a local minimum. There are more advanced ways to escape these local minima that are straight forward to implement. This is a

necessary but also relatively easy improvement to be a sufficient path planning algorithm for Lunar Zebro. Implementing this improvement could improve the reachability and therefore the robustness significantly.

Dynamic APF
An extensively researched improvement on the standard APF algorithm is the dynamic variant. This variant has the ability to prevent collisions with dynamic obstacles(other swarm rovers) and the ability to navigate on a moving target.

Gridsize adaptability
Also for the APF algorithm it could be very beneficial to adapt the grid according the place where Lunar Zebro is located. This would have a similar structure as described in Sec.4.4.1. However, the classical APF algorithm already outperforms the classical A* algorithm on larger grids. As a result of this, the gridsize adaptability for an APF algorithm results in less reduction of the planning time compared to an implementation on the A* algorithm.

Outlier prevention
Similar to the the A* algorithm, the classical APF has some path planning time outliers. These would ideally be prevented to ensure a predictable and reliable operation of the rover.

4.4.3. RRT path planning improvements for Lunar Zebro

It becomes clear that the A* and APF algorithms are very close to each other regarding performance for a Lunar Zebro path planning algorithm. Rapid exploring Random Trees is not able to keep up with this performance regarding planning time, path length and path predictability. Some improvements are listed that have the potential to optimise the performance metrics to the same level as the A* and APF algorithms. However, this improvements ensure only similar results as A* and APF and not better. The gained results for RRT are simply less compared to the gained results in A* and APF.

Path smoothing
The path length performance metric results of RRT are relatively bad compared to the A* and APF results. This is obviously caused by the randomness of the points taken as nodes for the path. An improvement would be to make the path more smooth. This can be done by making a straight lines between non adjacent nodes. As long as the new lines are not conflicting with obstacles, they could be used as a path. The previously formed nodes between the two new connected nodes can be skipped. This reduces the path length but it will still not be optimal.

RRT variants
There are many variants on the RRT algorithm(Ch.3) that could be implemented to improve the algorithm. Most of the variants are focused on improving just one aspect. These variants improve path length or planning time by for example starting the search from two sides (start and goal). Another possibility would be to sample the goal more often when determining the random nodes. This ensures a less expanding tree and lower path length. However, this introduces larger planning times.

Weighted grid points
The RRT algorithm can avoid obstacles by creating random nodes around an obstacle. However, the algorithm can also create weighted nodes and paths around an obstacle. In case weighted areas are introduced, the RRT algorithm could take the path with the lowest weight.

Moving goal and objects
Similar as for the other algorithms, the moving goal and obstacles can play a relevant role in swarming. With the correct modification RRT is able to overcome the challenge of a moving goal and moving obstacles. However, these modification need to be identified first in the proposed literature that is described in Ch.3.

4.5. Qualitative performance

From the results shown in Tab.4.3, the extensive analysis in Sec.4.3.2 and the possible improvements presented in Sec.4.4, qualitative conclusions can be drawn for each performance metric. Based on these conclusions and their analysis the decision on the base algorithm will be made.

4.5.1. Qualitative table

The qualitative performance results are shown in Tab.4.4. Each algorithm is graded against the performance metrics for Lunar Zebro. Both the quantitative and qualitative performance metrics are shown. However, the quantitative metrics are shown as qualitative metrics to ensure a valid comparison between the algorithms. Swarming metrics are specifically highlighted as the weight of these are substantial for a path planning application in LZ.

Table 4.4: Qualitative performance results with the required Lunar Zebro performance metrics

Performance metric		A* Algorithm [16]	RRT Algorithm [33]	APF Algorithm [13]
Path length		+	-	+
Robustness		++	+/-	-
Computation Power large (complex) grid		+/-	–	+
Computational Power small (simple) grid		+	-	+/-
Swarming capabilities	**Moving Obstacles (Other rovers)**	+ (With improvements)	+ (With improvements)	+ (With improvements)
	Moving target (set by swarm)	+ (With improvements)	+ (With improvements)	+ (With improvements)
	Path length predictability	+	-	+
	Possibility to improve path by swarm behaviour	-	-	++
	Swarm computation	+/-	+/-	+

4.5.2. Qualitative analysis

Looking at Tab.4.4 a few conclusions can be drawn. Both A* and APF are performing in a similar way but have their strengths and weaknesses on different metrics. The RRT algorithm, on the other hand, is clearly outperformed by the other two algorithms. This means therefore that the RRT algorithm needs individual improvements to gain the same performance as the other algorithms. The RRT algorithms is therefore not favorable as an implemented path finding algorithm in LZ. To distinguish between A* and APF, the value of the metrics must be determined. Looking at the swarming scheme of LZ it becomes clear that individual rover robustness is less relevant when the rover is operating in a swarm. As explained in Sec.4.2.1, the robustness of failing to find a path can be partly covered by the behaviour of the swarm. This automatically implies that the swarming capabilities must be a higher valued metric compared to the value of individual rover robustness. Looking at the swarming capabilities, the APF algorithm is performing better than the A* and RRT algorithms.

4.6. APF type base algorithm

As already concluded, the RRT algorithm is not beneficial for Lunar Zebro compared with the other two base algorithms. Giving more value to the swarming capabilities of the algorithm, APF outperforms the A* algorithm. As the APF robustness can be simply improved by variations on the algorithm, the Artificial Potential Field algorithm is chosen to be the most suitable base algorithm for Lunar Zebro. The typical local minimum problem will be addressed first as the reachability of this algorithm is the biggest pitfall. Improvements on the local minimum problem will therefore be the starting point for an improved Artificial Potential Field type of path planning algorithm.

5

The APF local minimum problem

Despite some common pitfalls, the Artificial Potential Field type of algorithm is the most suitable for a Lunar Zebro application. Ch.4 states that a part of these APF pitfalls can be solved by the capabilities of the swarm. Swarm robustness is therefore higher valued than the individual rover robustness. However, the rover must still be able to complete a mission individually. From the extensive analysis on this algorithm, it became clear that being trapped in a local minimum is the major cause for a low reachability. This chapter will go in further detail on this local minimum problem and will present two solutions. Each solution will again be tested with the help of the quantitative performance metrics. The chapter will present the best improvement as a conclusion.

5.1. Escape the local minimum problem

To gain a better understanding of the problem, a detailed analysis will be given. Furthermore a state of the art research will show some possible solutions to the described problem. The two implemented and tested solution will be presented in Sec.5.1.2 and Sec.5.1.3. Each solutions will be substantiated with the corresponding advantages, disadvantages and an algorithm implementation.

5.1.1. Problem analysis

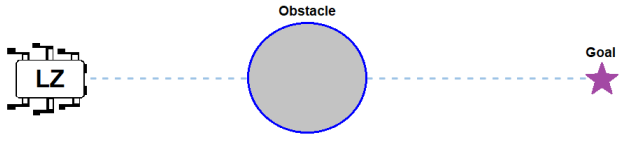
Before going into the possible solutions to escape local minima, the common local minimum scenario's will be identified first. There are three common local minimum cases that occur often when running the rover simulation. Those three cases are listed below and shown in Fig.5.1.

- Obstacle directly between goal and rover(Fig.5.1a),
- Goal right in front of obstacle(Fig.5.1b),
- Goal is in front of the rover and there are two obstacles left and right forming a narrow passage(Fig.5.1c).

State of the art solutions

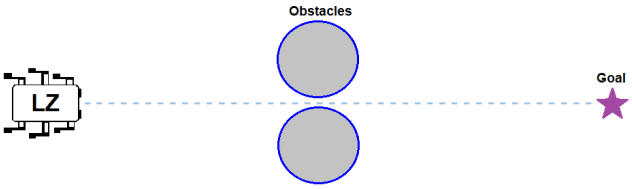
The avoidance of these local minima has been researched extensively. Literature proposes a lot of possible improvements which are able to, but not limited to escaping local minima. It can be concluded that there are many ways of solving the local minima problem. However, the use case of Lunar Zebro will determine what kind of solution would be most suitable. The requirements set in Sec.2.4 call for a minimization in computational complexity where possible. This results in looking further in the more simple solutions to avoid getting stuck in local minima. It can be the case that these relative simple solutions are not as successful as for example Artificial Intelligence (AI) algorithms. However, this does not automatically mean that the simple solutions are not fitting LZ. Below a few solutions proposed in literature are presented.

A common way of solving the problem is by placing virtual attractive and/or repellent forces on the route when encountering a local minimum. [14] proposes the implementation of multiple virtual



(a) Local minimum when obstacle directly between goal and rover

(b) Local minimum when goal right in front of obstacle



(c) Local minimum when two obstacles are forming a narrow passage

Figure 5.1: Artificial Potential Field common local minima cases

goals along the route to guide the robot along the desired route. Where [25] propose the use of virtual obstacles to create repellent forces to avoid getting stuck in local minima.

A second regular occurring method which has become more popular in recent years is the use of bio-inspired intelligence algorithms to avoid local minima. [36] proposes an improvement on the classical APF by looking from a reinforcement learning perspective. Furthermore, [11] shows an improvement by making use of Fuzzy decision trees and [43] shows improvements by using Simulated Annealing to escape local minima.

A third way, which is recently researched, is proposing a technique to avoid local minima in cluttered environments. [9] proposes a solution that keeps track of the walked path and returns on this path when a local minimum is detected. When returned, the algorithm starts finding a new path with knowledge about the local minimum location.

A fourth way of solving the local minima problem is found in more simple solutions such as [10], which is proposing a predefined shape and direction, hexagons, to avoid local minima. [37] proposes simple wall-following algorithm when stuck in a local minimum, which is very similar to the traditional Bug algorithm(Sec.A.2). This last category of solutions are the most attractive category as these simple implementations are low on computational power.

5.1.2. Forced direction when stuck in local minima

One of the simplest solutions to this problem can be found in just moving away from the local minimum. In practice this results in always walking a forced path when the rover gets stuck in a local minimum. As already said, [10] uses a hexagon shape to determine this path. A more simple approach would be to just go a certain amount of steps into one direction(”single line approach”) instead of following a defined shape. After this movement is finished, the original APF algorithm can take over again and can restart the search towards the goal. The simplicity of the forced direction approach comes with several advantages and disadvantages.

Advantages

It is clear that this approach will reduce the chance of being stuck in a local minimum. Besides this clear improvement, this improvement also comes with two other advantages:

1. Can be coded in a simple, fast and robust way,
2. Simple implementation and therefore low on computation power.

Disadvantages
A predefined or forced approach for the local minimum problem sounds like a simple and good solution. However, predefined solutions also come with several downsides which are listed below.

1. Forced steps can result in deviation from the optimal path,
2. If the forced steps end in an obstacle the algorithm will fail immediately, even though other paths are still available,
3. Because the algorithm always chooses one direction it is even more likely to deviate from the optimal solution,
4. The algorithm can create an infinity long path that just walks the same path again and again. For example if the algorithm turns left four times with the same amount of steps it ends up in the same location,
5. Forced paths are hard coded paths in software. This results in more hard coded software for edge cases that need to be solved. Hard coding results in more hard coding!

Implementation
The more forced steps, the bigger the change of deviating from the optimal path. This especially holds in a cluttered environment, where the local minimum problem has a high chance of occurring. The single line approach minimizes the amount of steps, that are made without the original APF algorithm, compared to the suggested hexagon approach. From this perspective the single line approach is chosen instead of the hexagonal approach.

The affect of disadvantages two and three can be reduced by the introduction of multiple possible directions instead of only one. This means that the rover will have the possibility to try walking the single line in different directions if one isn't working. It is chosen to have four different possible directions relative to the rover: 90° left, 90° right, 135° left and 135° right. This will also be executed in this particularly order. So, first the rover will try to go a predefined amount of steps 90° to the left, if this is not possible the rover will try to go a predefined amount of steps 90° to the right and so on. An invalid action, during the execution of the forced directions, is defined as setting a path point inside an obstacle or closer to the obstacle than the $RoverRadius$. In case of an invalid action, the next possible direction is tried.

The infinity long path problem (disadvantage four), can be tackled by introducing a maximum number of steps that can be set while doing a forced action. If during path planning the forced direction approach has to be used more than ten times, it is very likely the rover is stuck and makes an infinite long path. When exceeding the maximum number of forced directions during path planning the algorithm is stopped and returns a NaN, no possible paths.

Fig.5.2 shows a situation where the Forced direction APF escapes a detected local minimum. The local minimum is detected at point (22,22), where the algorithm forces the rover to take five steps in the direction that is 90° left to the rover orientation.

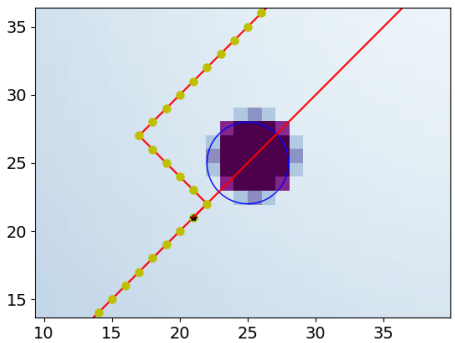


Figure 5.2: Forced direction APF escaping a local minimum at point (22,22)

Forced direction algorithm

The algorithm presented in Alg.3 showed the base APF algorithm. As the forced direction is only an addition to the base algorithm, the algorithm in Alg.3 is only extended. This extension is placed between lines 27 and 28 in the original algorithm, and is shown in Alg.4.

It starts with checking if there is a local minimum, if this is not the case the algorithm will work exactly as the original Artificial Potential Field algorithm. If a local minimum is detected, the points in the list $Path$ are removed depending on the amount of times the $Path$ was swinging around the local minimum($LocalMinimumDetectionLength$). After the removal of these path points, the forced directions can be tried to escape from the local minimum. This is achieved by determining the rover angle, where from all possible path angles can be deducted. From line 9, the path will step in the first calculated direction with a $PredefinedAmountOfSteps$. From line 12 it will be checked if these steps are not colliding with obstacles. If they are colliding, the next direction in $DirectionAngles$ will be checked instead. If the step is not colliding with obstacles, it will be added to the $Path$ list. When the forced direction steps are added, the original artificial potential field algorithm takes over the path planning from the last added point. The if condition on line 30 prevents the main algorithm from invoking the forced direction solution too many times. When the forced direction algorithm is invoked more than ten times, the algorithm will set the $NoLocalMinimum$ Boolean to False. This results in the termination of the main APF algorithm and thereby solving the 4th disadvantage presented in Sec.5.1.2.

Algorithm 4 Forced direction Algorithm

Require: Alg.3 between lines 27 and 28, $PredefinedDirectionCount = 0$
Ensure: Higher Succes/Fail ratio for the APF algorithm

1: **if** $LocalMinimum == True$ **then**
2: $PredefinedDirectionCount = +1$
3: **for** $LocalMinimumDetectionLength$ **do**
4: $RemoveLastItemFromList(Path)$
5: **end for**
6: $DetermineRoverAngle(Path)$
7: $DetermineDirectionAngles$
8: **for** $DirectionAngles$ **do**
9: **for** $PredefinedAmountOfSteps$ **do**
10: $APFN_{x,next} = APFN_{x,current} + Round(sin(DirectionAngle))$
11: $APFN_{y,next} = APFN_{y,current} + Round(cos(DirectionAngle))$
12: **if** $CollisionFree(APFN_{next}, ObstacleList, RoverRadius)$ **then**
13: $APFN_{current} = APFN_{next}$
14: $Path \leftarrow AddNodeToList(APFN_{current}, Path))$
15: $Distance \leftarrow CalculateDistance(APFN_{Goal}, APFN_{current})$
16: **if** $Distance < GridSize$ **then**
17: **return** $Path$
18: **end if**
19: **else**
20: **break**
21: **end if**
22: **if** $PredefinedAmountOfSteps$ **then**
23: $FoundPath = True$
24: **end if**
25: **end for**
26: **if** $FoundPath == True$ **then**
27: **break**
28: **end if**
29: **end for**
30: **if** $PredefinedDirectionCount > 10$ **then**
31: $NoLocalMinimum == False$
32: **end if**
33: **end if**

5.1.3. Enhanced curl-free vector field

A more advanced approach, but still relatively simple compared to all other improvements, is to start following the wall of obstacles when stuck in a local minimum. This technique is very similar to the traditional Bug algorithm. It is clearly chosen in Ch.4, that the path finding algorithm should be based on a artificial potential field path finding technique. The Bug algorithm is part of the sub set ”Artificial potential field type of algorithms”, so using this technique could be beneficial in terms of a Lunar Zebro application. However, the traditional Bug algorithm is still based on the fact that a local minimum has to be encountered first before the technique can be applied. Preventing being trapped in a local minimum instead of solving it, has an obvious preference. A technique that can follow the trajectory of an obstacle and makes sure that it can not end up in a local minimum is therefore highly preferable.

[5] proposes a technique called “Enhanced curl-free vector field”. This technique makes use of an attractive vector field towards the goal and a circular vector field around obstacles. Every grid point (multiple of the selected $GridSize$) holds two values, an X and a Y element. These two elements together form the vector at that specific point. In the conventional APF method each specific point holds just one value, which is the potential at that point. The original APF algorithm searches the lowest potential in the field, while the enhanced curl-free vector field follows the direction of the vectors instead. As the grid is still discreet, the vector direction is made discreet between the eight possible directions. These directions are the eight adjacent grid points which can become the next point in the path.

Advantages

The main purpose of the Enhanced curl-free vector field is avoiding the existence of local minima. There are more advantages to this approach which are listed below.

1. As every grid point has a direction (added degree of freedom) instead of potential, it is easier to manipulate the field into a desired path. This benefits possible swarm behaviour,
2. It is more advanced than the ”Forced direction approach”, but still relatively simple compared to AI algorithms,
3. The algorithm is able to decide in which direction the rover should pass the obstacle. This can be decided according to implemented rules instead of hard coding.

Disadvantages

There are some clear advantages compared to the original APF algorithm as well as to the ”Forced direction approach”. However, this improvement also comes with several downsides:

1. More degrees of freedom also leads into more possible bugs and can eventually results in more failures,
2. Even though the implementation is relatively simple, it still increases the computational complexity compared to ”Forced direction approach”,
3. The algorithm is not optimal in terms of path length,
4. The added degree of freedom also results in a more sensitive algorithm. In other words more parameters and gains must be tweaked to ensure a proper working algorithm.

Implementation

From the advantages and disadvantages discussed, it becomes clear that the enhanced curl-free vector field has a lot of potential benefits for the LZ application. To exploit these benefits, some improvements on the base algorithm as proposed in [5] are required. Before getting into these improvements, the basic principle and implementation of the enhanced curl-free vector field will be discussed. The main differences, compared to the original APF algorithm, are the way the attractive and repulsive potentials are defined and the corresponding decision on the direction of the next step.

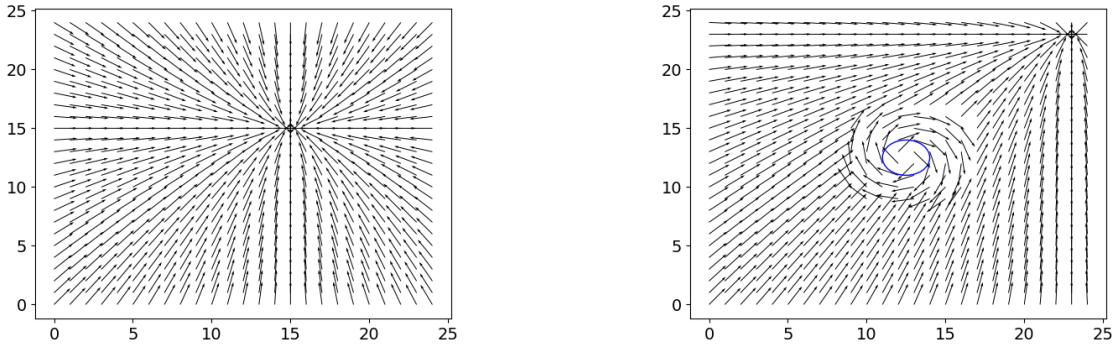
The attractive potential is not much different compared to the traditional APF method. The traditional APF method calculated the potential by using the Euclidean distance to the goal multiplied by an attractive gain. The enhanced curl-free vector field method defines the distance to the goal for the X and Y component separately. These distances are used as the vector components. This vector is than normalized and multiplied by a predefined attractive gain. This forms the attractive potential or

attractive vector. A field of these attractive vectors point towards the goal. Fig.5.3a shows such and attractive vector field with the goal located on (15,15).

The repulsive potential is calculated help of the circle gradient formula, as shown in Eq.5.1. The gradient of a circle consists of two base vectors, the unit vector **i** in the positive direction of the x-axis and the unit vector **j** in the positive y-direction. The constant c is used to denote if the vector field should be clockwise or counterclockwise around an obstacle. A positive c results in a clockwise vector field around an obstacle and a negative c results in a counterclockwise vector field. The direction of the rotation depends on how the rover approaches the obstacle. The correct rotation around an obstacle must ensure that the rover is taking the shortest path around the obstacle. This is dependent on the place of the obstacle, the place of the goal and the approaching side of the rover. Fig.5.3b shows a vector field with an obstacle located in point (12.5,12.5). The circular field is clockwise and ensures that the rover is forced around the obstacle in a clockwise direction. It is obvious that an increasing Euclidean distance between the denoted coordinate and the obstacle midpoint, results in an increasing repulsive vector size. As this is not necessarily needed, the vector is normalized before multiplied by the desired repulsive gain. The application of the repulsive rotational vector field should be limited to a certain distance from the obstacles. Within this distance the rover is forced in a circular motion around the obstacle. This distance is depending on the desired clearance around an obstacle.

$$\nabla f(x, y) = c \cdot y\mathbf{i} - c \cdot x\mathbf{j} \tag{5.1}$$

To form the actual vector field, the attractive vector and repulsive vector are added together on each grid point. This results in a vector field that can be used for navigation in a similar way as the potential field is used in the original APF algorithm. The last major difference compared to the original APF is how to navigate through the formed vector field. As already said in Ch.4, there are eight discreet possible directions when located on an arbitrary grid point. To keep this property, the angle of the vector in this grid point is calculated. The angle is calculated with respect to the normal vector, which is defined as the vector in the positive y-direction. The possible 360°is divided into eight equal segments, where each segment represents one of the eight possible directions. Depending on the pointing direction of the vector in these segments, the next point of direction is chosen. This next point($Node$) is than added to the $Path$ list. When this $Node$ is added the algorithm will start the search to the next $Node$ in exactly the same way.



(a) Vector field towards goal

(b) Vector field with a clockwise rotation around an obstacle

Figure 5.3: Vector fields corresponding with the Curl-Free vector field algorithm

Enhanced curl-free vector field algorithm

The basic principles of the enhanced curl-free vector field algorithm are shown in Alg.5. The functioning of the original APF algorithm as shown in Alg.3 is not much different from the working of the enhanced curl-free vector field algorithm. As said in the implementation section, mainly the calculations of the attractive potential and repulsive potential are different. Only these two functions will therefore be shown in Alg.5. Functions and variables that are already discussed in Tab.4.1 and Tab.4.2 will not be highlighted as they have the same purpose in Alg.5.

As only the attractive potential and repulsive potential functions are described in the pseudo algorithm, Alg.3 is still a requirement. Furthermore, the initialization of the variable $PotentialMap$ will not be done by single zero's but by empty vectors ($[0, 0]$). A new category of nodes is introduced, the VFN nodes for the vector field.

On line 1 the $CalculateAttractivePotential$ function is described. It has a simple return value which is a vector that represents the attractive gain at a specified X, Y coordinate. These are simply created by the x, y differences from the specified point to the goal. This results in a vector that is normalized ($AttractiveVector$) and finally multiplied by a predefined gain, the $AttractiveGain$.

The $CalculateRepulsivePotential$ is described from line 7 onward. To determine the direction of the enhanced curl-free vector field around an obstacle, the obstacle location relative to the rover and the VFN_{goal} is required.

The vector($RoverGoalVector$) between the rover($VPFN_{start}$) and $VPFN_{goal}$ is defined when a new obstacle is detected. On line 12 the vector between the obstacle centre and the rover is defined, the $RoverObstacleVector$. The $RoverGoalVector$ and $RoverObstacleVector$ together result in the $GoalObstacleAngle$, which is used to determine the direction of the enhanced curl-free vector field. From line 19 onward this information is used to calculate the repulsive vector field within a certain distance from the obstacles. The vector field is calculated by making use of the formula stated in Eq.5.1. On line 28 the calculated vector is added to $RepulsiveVector$ which may or may not already have a value depending on the interference of other obstacles in the field. Finally the $RepulsiveVector$ is returned multiplied by a predefined $RepulsiveGain$.

Algorithm 5 Enhanced curl-free vector field attractive and repulsive potential calculations [5]

Require: Alg.3, An empty vector([0,0]) initialized array $PotentialMap$
Ensure: Higher reachability compared to the original APF algorithm

1: $CalculateAttractivePotential(X, Y, VFN_{Goal})$:
2: $\quad AttractiveVector_X = VFN_{Goal,X} - X$
3: $\quad AttractiveVector_Y = VFN_{Goal,Y} - Y$
4: $\quad AttractiveVector = [AttractiveVector_X/VectorLength, AttractiveVector_Y/VectorLength]$
5: $\quad$ **return** $AttractiveVector * AttractiveGain$
6:
7: $CalculateRepulsivePotential(X, Y, VFN_{Goal})$:
8: $\quad RepulsiveVector = [0, 0]$
9: $\quad RoverGoalVector \leftarrow CalculateVector(VFN_{Goal}, VFN_{Start})$
10: $\quad$ **for** $Obstacles$ **do**
11: $\quad\quad ObstacleDistance \leftarrow CalculateDistance(Obstacle_{Centre(x,y)}, VFN_{current})$
12: $\quad\quad RoverObstacleVector \leftarrow CalculateVector(Obstacle_{Centre(x,y)}, VFN_{Start})$
13: $\quad\quad GoalObstacleAngle \leftarrow CalculateAngle(RoverGoalVector, RoverObstacleVector)$
14: $\quad\quad$ **if** $GoalObstacleAngle >= 0$ **then**
15: $\quad\quad\quad Obstacle_{CurlDirection} = 1$
16: $\quad\quad$ **else**
17: $\quad\quad\quad Obstacle_{CurlDirection} = -1$
18: $\quad\quad$ **end if**
19: $\quad\quad$ **if** $ObstacleDistance <= (RoverRadius + Obstacle_{Radius}) * SafetyFactor$ **then**
20: $\quad\quad\quad$ **if** $X, Y == Obstacle_{Centre(x,y)}$ **then**
21: $\quad\quad\quad\quad RepulsiveVector = [0, 0]$
22: $\quad\quad\quad$ **else**
23: $\quad\quad\quad\quad RepulsiveVector_X = Obstacle_{CurlDirection} * (y - Obstacle_{Centre(y)})$
24: $\quad\quad\quad\quad RepulsiveVector_Y = -Obstacle_{CurlDirection} * (x - Obstacle_{Centre(x)})$
25: $\quad\quad\quad\quad RepulsiveVector_X = RepulsiveVector_X/VectorLength$
26: $\quad\quad\quad\quad RepulsiveVector_Y = RepulsiveVector_Y/VectorLength$
27: $\quad\quad\quad$ **end if**
28: $\quad\quad\quad RepulsiveVector+ = [RepulsiveVector_X, RepulsiveVector_Y]$
29: $\quad\quad$ **end if**
30: $\quad$ **end for**
31: $\quad$ **return** $RepulsiveVector * RepulsiveGain$

5.2. Results

From Sec.5.1.2 and Sec.5.1.3 it becomes clear that both improvements are based on an artificial potential field method, but have an obvious difference in implementation. Before drawing any conclusions on which algorithm would be most applicable for Lunar Zebro, the quantitative results of the algorithms will be compared. Tab.5.1 shows the results of the two discussed improvement algorithms compared to the original APF algorithm. It shows the three relevant performance metrics for the six different scenarios. The scenario's are the same obstacle scenario's as used for the quantitative performance comparison in Sec.4.3. Each scenario again consists of 500 runs with the saved obstacle maps. All other standard conditions as described in Sec.4.2.3 are applied.

In summary, Tab.5.1 presents a conclusion on all three performance metrics. The last column of the table adds up the planning times, path lengths and amount of failed paths from all obstacle scenario's. These total amounts can be used as a direct quantitative comparison between the algorithms. First, The enhanced curl-free vector field algorithm needs around three times more computation time than the original APF algorithm. The forced direction approach uses just around 20% more computation time compared to the original APF algorithm.

Second, the path length of the forced direction algorithm is clearly longer that the other two algorithms. As already concluded, this is mainly caused by the fact that the forced directions are deviating from the optimal path. This can be observed in the path length results. The path length of the enhanced curl-free vector field algorithm is slightly longer than the original APF algorithm.

Third, the amount of failed paths of the forced direction algorithm is almost halved compared to the original APF algorithm, while the enhanced curl-free vector field algorithm just shows similar results.

A minor note to these results is that a higher reachability also means that the algorithm is able to form more complex paths to the goal. A more complex path, most of the time, means the avoidance of more obstacles. Avoiding more obstacles results in longer paths and longer computation time. Therefore the results of the mean time and mean path length have a small biased upwards when it has a higher reachability compared to other simulations. When comparing algorithms with similar results on the mean time and on the path length, this bias needs to be considered in further conclusions.

Table 5.1: Result comparison between APF, forced direction APF and basic enhanced curl-free vector field

Number of Random obstacles, Radius size in meters		*#5* *Radius 3 (m)*	*#10* *Radius 3 (m)*	*#15* *Radius 3 (m)*	*#15* *Radius 1 (m)*	*#30* *Radius 1 (m)*	*#60* *Radius 1 (m)*	**Total of all obstacle scenario's**
Mean Planning time (ms)	**Original APF [13]**	223	470	789	552	1342	3869	7245
	Forced Direction	235	502	834	584	1566	4848	8569
	Enhanced curl-free [5]	1280	2162	3192	2501	4585	8853	22573
Mean Path length (m)	**Original APF [13]**	58.96	59.89	61.05	57.93	58.96	60.27	357.06
	Forced Direction	65.74	73.58	81.35	64.37	76.10	99.88	461.02
	Enhanced curl-free [5]	59.98	61.73	62.90	58.59	59.79	61.58	364.57
Amount of NaN's (Out of 500)	**Original APF [13]**	98	235	348	79	186	390	1336
	Forced Direction	23	131	253	8	50	256	721
	Enhanced curl-free [5]	89	254	374	51	190	389	1347

5.3. Conclusion on the APF local minimum improvements

Looking at the results from the three algorithms, it seems that the forced direction algorithm is the best choice regarding reachability. However, when taking into consideration the importance of finding an optimal path regarding path length, the forced direction algorithm is not suitable enough. Second, the ”hard-coded” direction approach results in more hard-coding when improvements are implemented. This is obviously not desired over a rule based coded algorithm. Looking at the enhanced curl-free vector field approach, it becomes clear that it needs improvements in planning time and reachability. The algorithm shown in Alg.5 is a basic version of the enhanced curl-free vector field algorithm as presented in [5]. This implies that changes to the algorithm can still be made to improve the planning time and reachability.

The stated conclusions and improvements show that the enhanced curl-free vector field approach is the most suitable for the Lunar Zebro application. When implementing improvements the algorithm could be lower on computation time and higher on reachability. This improvements must be identified and researched to ensure the desired operation of the enhanced curl-free vector field as a path planning algorithm in LZ.

6

Rotational vector field

As suggested the enhanced curl-free vector field is a good solution for the implementation of a path finding algorithm in Lunar Zebro. To improve the performance of the algorithm, a few adjustments must be made to the basic enhanced curl-free vector field algorithm which is shown in Alg.5. These adjustments result in the creation of a new algorithm, the Rotational Vector Field (RVF) algorithm. The RVF algorithm has similar but not identical behaviour as the enhanced curl-free vector field algorithm. This chapter will present the core improvements that will form the RVF algorithm. The improvements will be mainly focused on decreasing the path planning time and to increase the reachability of the algorithm. Beside the planning time and reachability improvements, a small path length improvement will also be suggested. The chapter will conclude with the results of the improvements based on the previously described quantitative performance metrics.

6.1. Time optimization

The basic algorithm has a planning time which is far from optimal compared to the other algorithms. Even when taking the non-realtime system behaviour into account, the difference is a clear slow down for the hosting system hardware. This also causes extreme long simulation times, which slow down the generation of relevant data. Improving the RVF planning time will therefore also improve the time that is needed to run the simulation. Improving the path planning time will be carried out first, as it will benefit further computations.

6.1.1. Challenge

Taking a closer look at how the attractive and repulsive potentials are determined (calculated in the functions $CalculateAttractivePotential$ and $CalculateRepulsivePotential$), they are both calculated for every single grid point. Secondly, all grid points are recalculated every time the rover sees a new obstacle. This results in some unnecessary computations. These computations are listed below:

1. As the attractive potential will always stay the same, it is not needed to recalculate the attractive vectors when new obstacles are detected,
2. The rotation of the field around an obstacle does not have to be calculated every time the repulsive gain for a new grid point is calculated,
3. When a new obstacle is detected it is not needed to calculate the repulsive gain for the already seen obstacles.

These computations also happen for the original APF algorithm. However, because of the more extensive calculations for the RVF algorithm these parts are becoming highly inefficient. These slowdowns are mainly caused by the generation of the potential(vector) map. Changing the structure and invoking of this map, will have a significant influence on the planning time.

6.1.2. Improvements

Looking at Alg.3, the potential map is calculated between lines 4 and 8. For improvement of challenges one and two, this $PotentialMap$ calculation needs to be changed. Alg.6 shows the new calculation of the $PotentialMap$ between lines 1 and 18. The rotation direction around an obstacle is now calculated before the $NegativePotential$ of all grid points is determined. This results in calculating the rotation direction only once when the obstacle is detected. Moreover, the $PosistivePotential$ is only determined during the first map calculation and not every time a new obstacle is detected. Note that this is only possible if the $NegativePotential$ is added to the $PotentialMap$, as the $PositivePotential$ needs to be ”remembered” in the $PotentialMap$. If the attractive field is not remembered, it will be lost if a new obstacle is detected.

For the third challenge a simple improvement is introduced on lines 2 and 23. The rotational direction and $RepulsiveVector$ is only determined for new detected obstacles instead of for all obstacles.

The path length and reachability of the algorithms stays the same when the time improvement is implemented. This implies, as expected, that there were no (undesired) functional changes by improving the planning time of the algorithm.

Algorithm 6 Rotational vector field with time improvement

Require: Alg.3 between lines 4 and 8
Ensure: Time improved rotational vector field algorithm

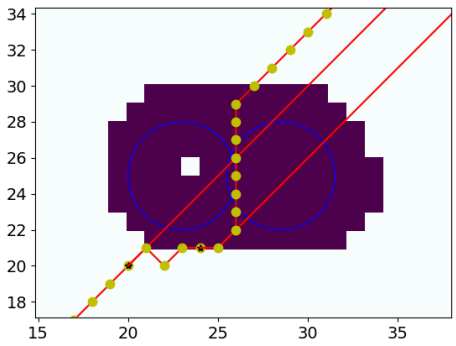
1: $RoverGoalVector \leftarrow CalculateVector(VFN_{Goal}, VFN_{Start})$
2: **for** $NewObstacles$ **do**
3: $ObstacleDistance \leftarrow CalculateDistance(Obstacle_{Centre(x,y)}, VFN_{current})$
4: $RoverObstacleVector \leftarrow CalculateVector(Obstacle_{Centre(x,y)}, VFN_{Start})$
5: $GoalObstacleAngle \leftarrow CalculateAngle(RoverGoalVector, RoverObstacleVector)$
6: **if** $GoalObstacleAngle >= 0$ **then**
7: $Obstacle_{CurlDirection} = 1$
8: **else**
9: $Obstacle_{CurlDirection} = -1$
10: **end if**
11: **end for**
12: **for** $X, Y \in Area$ **do**
13: **if** $FirstMapCalculation$ **then**
14: $PositivePotential \leftarrow CalculateAttractivePotential(X, Y, VFN_{Goal})$
15: **end if**
16: $NegativePotential \leftarrow CalculateRepulsivePotential(X, Y, ObstacleList)$
17: $PotentialMap \leftarrow TotalPotential(X, Y, PostivePotential, NegativePotential)$
18: **end for**
19:
20:
21: $CalculateRepulsivePotential(X, Y, VFN_{Goal})$:
22: $RepulsiveVector = [0, 0]$
23: **for** $NewObstacles$ **do**
24: **if** $ObstacleDistance <= (RoverRadius + Obstacle_{Radius}) * SafetyFactor$ **then**
25: **if** $X, Y == Obstacle_{Centre(x,y)}$ **then**
26: $RepulsiveVector = [0, 0]$
27: **else**
28: $RepulsiveVector_X = Obstacle_{CurlDirection} * (Y - Obstacle_{Centre(y)})$
29: $RepulsiveVector_Y = -Obstacle_{CurlDirection} * (X - Obstacle_{Centre(x)})$
30: $RepulsiveVector_X = RepulsiveVector_X / VectorLength$
31: $RepulsiveVector_Y = RepulsiveVector_Y / VectorLength$
32: **end if**
33: $RepulsiveVector+ = [RepulsiveVector_X, RepulsiveVector_Y]$
34: **end if**
35: **end for**
36: **return** $RepulsiveVector * RepulsiveGain$

6.2. Clustered rotation

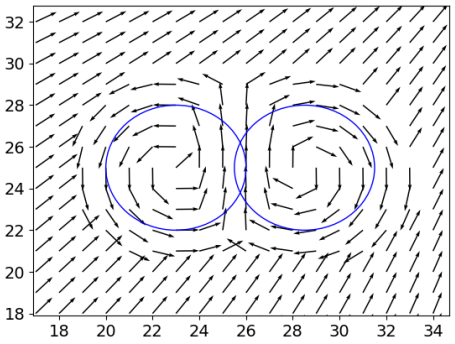
To improve the reachability it needs to be identified in what particular situations the algorithm is not able to find a path. A condition that will immediately results in the termination of the algorithm, is a path planned through an obstacle. A detailed look at the situations that fail to create a path, shows that in the majority of the cases, the algorithm tries to create a path through obstacles.

6.2.1. Challenge

Making a path through an obstacle can be caused when there are two conflicting vector fields. If the rover is approaching a single obstacle, it will move around it without any problems. However, when two obstacles are close to each other, the corresponding vector fields can interfere. When they are both having the same rotation direction this isn't a problem, as the rover will just pick up the rotation of the next obstacle and continues the path. The algorithm determines the rotation of the field depending on the locations of the rover, obstacle and the goal. This ensures taking the shortest path around an obstacle. However, this approach could lead to adjacent obstacles having different rotations. This is shown in Fig.6.1, where Fig.6.1a shows the obstacle situation with the planned path(red line) and walked path(yellow dots). For demonstration purposes, the condition which makes it impossible to plan a path through an obstacle is removed here. The figure shows a planned path through two adjacent obstacles. Fig.6.1b shows that this is caused by the joint vector fields of both obstacles. These vectors have a pointing, and therefore suggested direction, through the obstacles. This problem will not occur when both obstacles have the same vector rotation, which can either be clockwise or counterclockwise.



(a) Obstacles situation



(b) Vector field caused by obstacles

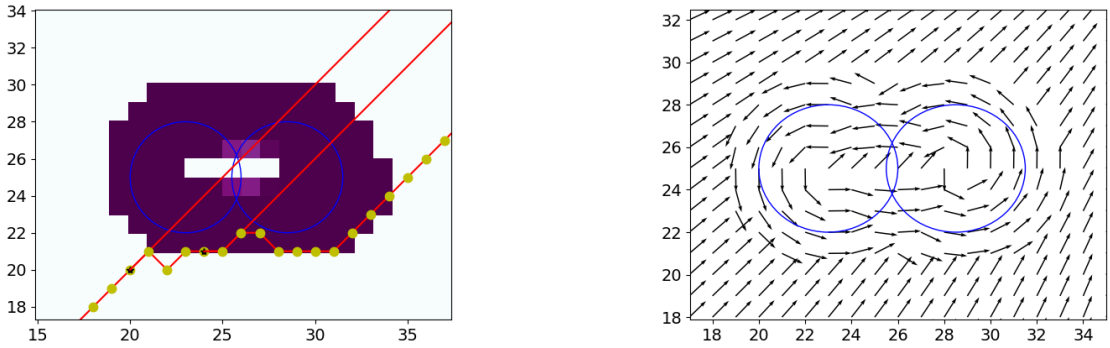
Figure 6.1: Non clustered obstacles

6.2.2. Improvements

The suggested solution for this challenge is to force both obstacles in the same vector direction. Forcing all obstacles in the map to the same direction would make the algorithm inefficient regarding path length, as this would look similar to the bug algorithm. The only obstacles that should have the same rotation, are the ones that are too close to each other for the rover to pass through. The solution for this problem would be to give new obstacles that are too close to an already detected obstacle, the same rotation as the already detected obstacle. This check needs to be done when the rotation of an obstacle is determined. The algorithm for determining the rotation of the obstacle is already shown in Alg.6. The improvement must be implemented after line 10 in Alg.6. The pseudo code for this new obstacle check is shown in Alg.7. A $FreeSpace$ of at least three times the $RoverRadius$ is chosen instead of a just two times the $RoverRadius$. This is due to the discreet grid, which can result in a grid point not exactly in the middle of two obstacles. When this is the case the $FreeSpace$ could still be two times the $RoverRadius$, but the discreet grid point could still be too close to the obstacle. When taking three times the $RoverRadius$, there will always be a grid point far enough from both obstacles. This approach requires that the $GridSize$ should be equal or smaller than the $RoverRadius$. The result can be seen in Fig.6.2. Fig.6.2a and Fig.6.2b show the new planned path and corresponding vector field respectively.

Algorithm 7 Rotational vector field with cluster improvement

Require: Alg.6 after line 10, $GridSize \leq RoverRadius$
Ensure: Reachability improved rotational vector field algorithm
1: **for** $DetectedObstacles$ **do**
2: $FreeSpace \leftarrow CalculateDistanceBetweenObstacles_{DetectedObstacle,NewObstacle}$
3: **if** $FreeSpace \leq 3 * RoverRadius$ **then**
4: $NewObstacle_{rotation} = DetectedObstacle_{rotation}$
5: **end if**
6: **end for**



(a) Obstacles situation

(b) Vector field caused by obstacles

Figure 6.2: Clustered obstacles

6.3. Outward potential

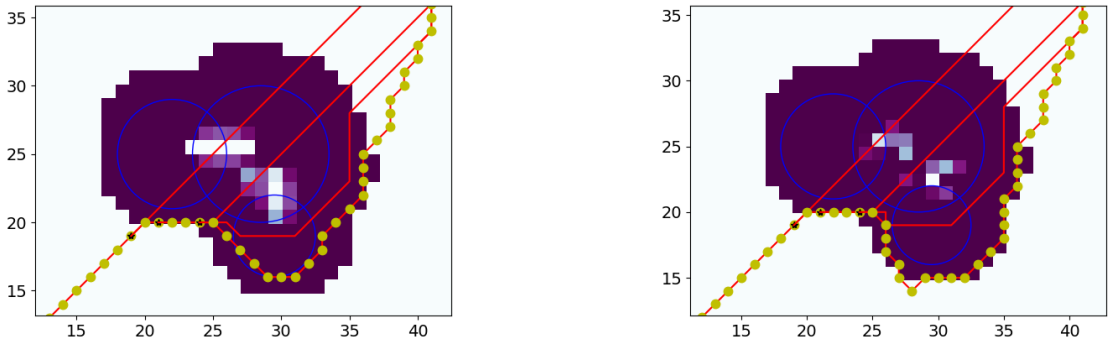
Further improvement on the reachability can still be made by reducing the planned paths through obstacles. The clustered rotation improvement explained and discussed in Sec.6.2, is also causing a new problem. Looking at Fig.6.2b, it shows that when the rover is following the left obstacle closely it could potentially end up with a path inside the second obstacle. This is caused by the vector field from the left obstacle that is ”throwing” the rover into the right obstacle.

6.3.1. Challenge

This problem is caused by the vectors that have a circular motion around the obstacle and don’t allow the rover to ”detach” from the obstacle. Naturally because of the attractive vector field towards the goal, this problem is slightly solved on the side of the obstacle that is facing the goal. At this side the vectors are pointing a little bit outward, as adding the attractive vector to the repulsive vector causes a slight deviation of the total vector towards the goal. However, this effect is negative on the side of the obstacle that is not facing the goal, as adding the attractive vector will result in a slightly facing inward(towards the obstacle) vector. As the rover is typically approaching the obstacle form this side, it could be the case that the rover is getting too close to the obstacle and is therefore not able to detach on time. Especially when two obstacles are close to each other, this problem occurs often, resulting in a path that is planned through an obstacle. Fig.6.3a shows an example of an obstacle situation where the large obstacle ”throws” the rover too close to the smaller obstacle that is located below the large obstacles. The path is therefore formed through the smaller obstacle.

6.3.2. Improvements

When the environment is relatively cluttered with large obstacles, this can be a problem as shown in Fig.6.3a. If the rover is forced away from the obstacle the chance of crashing in an adjacent obstacle would be reduced. For this reason a slight outward vector is added to the repulsive vector field around an obstacle. This outward vector is added to the already calculated repulsive vector from the



(a) Path formed without outward potential

(b) Path formed with outward potential

Figure 6.3: Outward potential obstacle situations

obstacle itself. Looking at Alg.6 this part of the code is therefore added between lines 32 and 33. Alg.8 shows how this outward potential is calculated and added to the original repulsive vector. Note that the $OutwardSafetyFactor$ should be less than the $SafetyFactor$ for determining the size of the repulsive field. Having the whole repulsive field pointing outward could cause oscillations(local minima) when bumping in the repulsive field. Therefore it is chosen to only use the outward potential when the rover reaches a critical distance to the obstacle. The affect of adding this outward vector to the algorithm can be seen in Fig.6.3b. The planned path is not crossing the obstacle boundaries anymore.

Algorithm 8 Rotational vector field with outward vector improvement

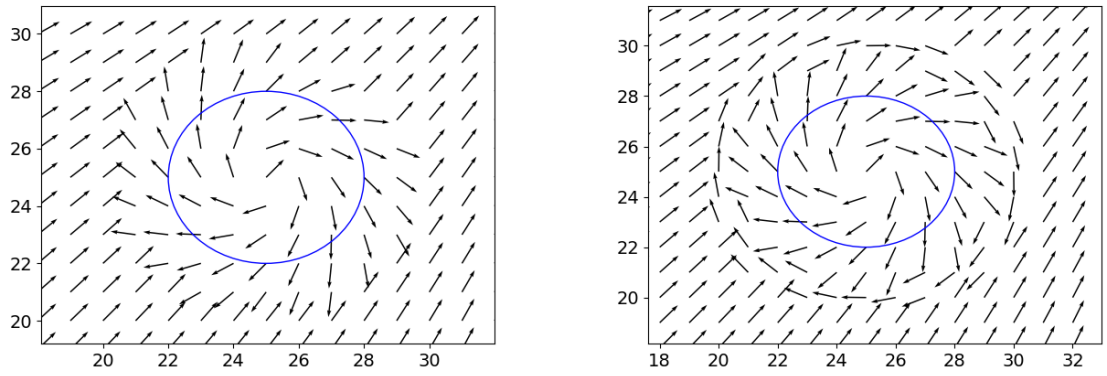
Require: Alg.6 between lines 32 and 33
Ensure: Reachability improved rotational vector field algorithm
1: **if** $ObstacleDistance <= (RoverRadius + Obstacle_{Radius}) * OutwardSafetyFactor$ **then**
2: $\quad RepulsiveVector_X = RepulsiveVector_X + (X - Obstacle_x)/3$
3: $\quad RepulsiveVector_Y = RepulsiveVector_Y + (Y - Obstacle_y)/3$
4: **end if**

6.4. Gridsize dependency

The variables $SafetyFactor$ and $OutwardSafetyFactor$ are arbitrary determined and of large influence on the size of the repulsive fields. These factors determine how far the repulsive field can spread around an obstacle by multiplying it with the ”no-go” radius from obstacles. This spreading is therefore dependent on the size of the obstacles. As the size of the obstacle has nothing to do with the clearance area that is needed for the rover around the obstacle, it should be eliminated from the equation. This would make the algorithm scalable to any obstacle size instead of just the simulated and tested radius sizes.

6.4.1. Challenge

As pointed out in Sec.6.3, if the outward vector is impacting the whole repulsive field it will create oscillations. A differentiation between the complete repulsive vector field and the outward pointing vector must therefore be implemented. Second, the field must be large enough to ensure a safe distance between the rover and the obstacle edge. Fig.6.4a shows an obstacle of radius 3 where the outward potential is spread across the complete field. The outward potential will create a path that is not following the curve of the obstacle as it will always throw the rover outside the repulsive potential field. To meet the edge following requirement, a ring of grid points that is not influenced by the outward potential must be created. A second challenge that should be resolved is the uneven spread field around an obstacle. From Fig.6.4a it becomes clear that the vector field reaches more grid points towards the origin of the field than towards the goal. This suggests that the field is not of equal size at every point around the obstacle.



(a) Vector field without grid dependency

(b) Vector field with grid dependency

Figure 6.4: Vector potential for an obstacle with radius 3

6.4.2. Improvements

To obtain a ”ring” of repulsive vectors that is not influenced by the outward potential, the repulsive field must at least reach two grid points outside the $ObstacleRadius + RoverRadius$. One grid point for the outward vector and one for the normal repulsive vector field that is not influenced by the outward pointing vector. The repulsive field must therefore be dependent on the grid size instead of just a safety factor. To make a clear distinguish between the ”no-go” area and the repulsive outward potential, the rover radius is included in the ”no-go” area. As the rover radius is determined to be zero during these simulations, the ”no-go” area is defined as the obstacle(blue circle). Effectively the functioning of the code is not changing, but the determination of the repulsive rings around the obstacle becomes more clear.

The second challenge is the uneven spread repulsive field around the obstacle. Knowing the importance of the field size, it gives an urge to create an even distributed field. When creating an even spread field with a discreet grid, the diagonal length of a grid point at an angle from the X-axis or Y-axis must be used. The largest distance is obtained when the point is found at an angle of 45° relative to the X-axis or Y-axis. This point holds the largest euclidean distance to the obstacle center. To ensure the inclusion of the grid points at this angle, the diagonal(Pythagoras) length of a grid point must be used instead of the $GridSize$. The diagonal length is used to determine the size of the field around an obstacle. Taking into account both improvements, the new boundary conditions for determining the repulsive vectors and outward vectors are shown in Eq.6.1 and Eq.6.2 respectively.

$$ObstacleDistance <= (Obstacle_{Radius} + RoverRadius + 2 * (\sqrt{GridSize^2 + GridSize^2})) \quad (6.1)$$

$$ObstacleDistance <= (Obstacle_{Radius} + RoverRadius + (\sqrt{GridSize^2 + GridSize^2})) \quad (6.2)$$

Fig.6.4b shows the new repulsive vector field around an obstacle with a radius of three meters. The figure shows a field with an outward vector field and a circular vector ring without the influence of the outward vectors. Furthermore, the vector field is evenly distributed around the obstacle. With the implementation of this improvement the algorithm is scalable to any obstacle size as it is only dependent on the chosen $GridSize$.

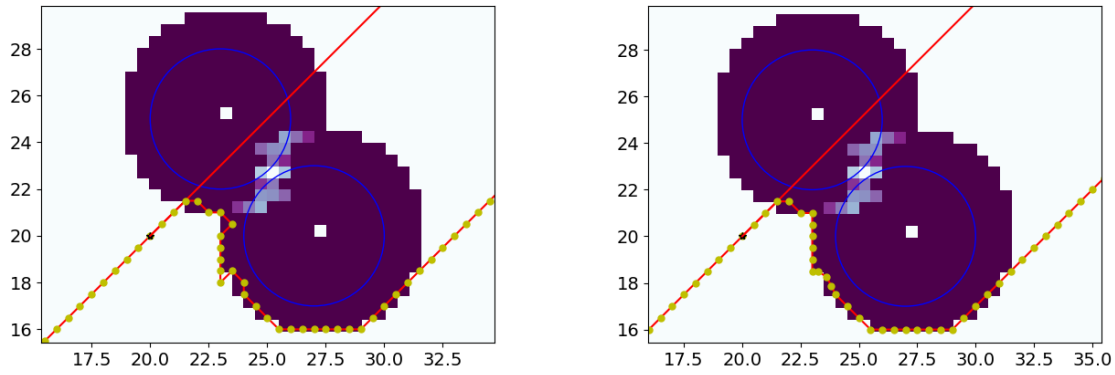
6.5. Path optimization

The final aspect of improvement can be found in the path length. As denoted in Ch.5, the rotational vector field is not as optimal as the traditional APF algorithm regarding path length. However, the algorithm could become closer to optimal with some improvements. Looking at the current hardware and software state of Lunar Zebro, it is concluded that taking corners can not yet be executed during

walking. Taking corners will be quite time consuming. Reducing the amount of corners is therefore relatively beneficial for Lunar Zebro. In summary, path optimising is found in reducing the amount of corners and reducing the path length.

6.5.1. Challenge

Fig.6.5a shows a constructed(red lines) and walked(yellow dots) path that is generated by the original RVF algorithm. The algorithm constructs a path that takes a lot of unnecessary corners. This unnecessary corners especially occur when the rover is thrown from the first obstacle(left) to the second obstacle(right) and a small gridsize (0.5m) is selected. The improvement here would be to remove those corners and replace the path point by one that is exactly in the middle of that two surrounding path points, such that the path becomes straight.



(a) Obstacle situation without path optimization

(b) Obstacle situation with path optimization

Figure 6.5: Obstacle situations with path optimization, gridsize 0.5 meters

6.5.2. Improvement

To achieve this reduction in path length and corners, the algorithm described in Alg.9 is used. This algorithm improvement can be applied to any formed path independently of the algorithms that created the path. The placement of this improvement must be done just before the final created path is returned. The improvement checks if the distance between a $PathPoint$ and the second next $PathPoint$ is smaller or equal than two times the $GridSize$. If that is the case, the next point must be exactly in the middle. Note that for lines that are going parallel to the X-axis or Y-axis this is already the case. Therefore, the next $PathPoint$ will be replaced by the same point. If the line between these two points is not parallel to the X-axis or Y-axis and the distance is still smaller than 2 times the $GridSize$, the algorithm created an unnecessary corner. The unnecessary $PathPoint$ is than replaced by one exactly between these two points. The result of this improvement is shown in Fig.6.5b, which shows a path that is not making unnecessary turns compared to path shown in Fig.6.5a.

Algorithm 9 Rotational vector field with path improvement

Require: Alg.6
Ensure: Path length improved rotational vector field algorithm

for $PathPoints$ **do**
  $DistanceBetweenNodes \leftarrow CalculateDistacne(VFN_{PathPoint}, VFN_{PathPoint+2})$
  if $DistanceBetweenNodes <= 2 * GridSize$ **then**
    $New_X \leftarrow (VFN_{PathPoint+2,X} - VFN_{PathPoint,X})/2 + VFN_{PathPoint,X}$
    $New_Y \leftarrow (VFN_{PathPoint+2,Y} - VFN_{PathPoint,Y})/2 + VFN_{PathPoint,Y}$
    $VFN_{PathPoint+1} \leftarrow [New_X, New_Y]$
  end if
end for

6.6. Structuring and gain tweaking

The last step in code development is structuring, commenting and gain tweaking. Structuring and commenting will ensure readability and comprehensibility for external readers. However, this is not the only advantage. Structuring the code includes removing unnecessary calculations and use function based methods where possible. This will not have any influence on the functionality of the code, but it will impact the computational power that is needed for the algorithm. Therefore the path length and reachability will not change, while the planning time is most likely to be positively affected.

Gain tweaking, on the other hand, will improve the functionality of the code. Gain tweaking results in decreasing path lengths and increasing reachability. The tweaked gains are summed up below.

- Attractive gain,
- Rotational repulsive gain,
- Outward pointing repulsive gain.

It is determined that the repulsive gain must be larger than the attractive gain. The repulsive gain must be around two orders of magnitude larger than the attractive gain. This ensures that the attractive field has minor interference with the repulsive field, such that the rotational field is not deformed. Furthermore, it is suggested that the rotational repulsive vector has the same magnitude as the outward pointing repulsive vector. This results in an outward vector that has a 45° angle relative to the normal vector on the obstacle edge. An outward vector with a 45° angle, shows the best results regarding path length and reachability. Alg.11 as shown in the appendix, presents the complete rotational vector field algorithm as implemented to obtain the final results.

6.7. Results

To compare the results of the improved and the original RVFalgorithm, the same obstacle scenarios will be used as discussed in Sec.4.2.3. The saved obstacle location will again be used to create the 500 obstacle maps for all scenario's. This ensures that the path length and number of NaN's are a direct comparison between the improvements and the basic algorithms. The planning time however may differ even when the algorithm is creating the exact same paths. This is again caused by the behaviour of a non-realtime system. For every improvement only the results of the rotational vector field are determined and shown, as the performance of the other algorithms will not change if the RVF algorithm is improved. The results for all obstacle scenario's will be added together for each algorithm or improvement. These total results will again be used to determine the quantitative performance improvements relative to the proposed algorithms in literature. First the planning time of all improvements will be discussed, followed by the path length and finally the reachability.

6.7.1. Planning time

The results for the planning time of all improvements are shown in Tab.6.1. A significant decrease in planning time is obtained when the time improvement is introduced. Some new improvements result in a slightly increased planning time again. This is caused by two factors. The first being the small bias that is introduced when the reachability is increased. This affect is extensively described in Sec.5.2. Second, when introducing new functionalities or improvements, it could increase the computational complexity of the algorithm. When introducing the gridsize dependency and path optimization the planning time is again further reduced, despite the increase in reachability. The final gain tweaked implementation even reaches a planning time that is lower than the original APF algorithm. The final RVF algorithm accomplishes a planning time reduction of 62% on the original APF, a 68% reduction on the forced direction APF and finally a remarkable 88% reduction on the original RVF algorithm.

Fig.6.6 shows the final results for the planning times in a boxplot analysis. The top figure shows the planning times for the original RVF algorithm and the bottom figure shows the planning times for the improved RVF algorithm. The obstacle scenario's correspond respectively with the obstacle scenario's in Tab.6.1. Obstacle scenario zero corresponds with five random obstacles with radius of three meters. Obstacle scenario five corresponds with 60 random obstacles with radius of one meter. The improved algorithm shows a significant reduction in planning time for all obstacle scenario's compared to the original RVF. It can therefore be concluded that the introduced improvements have a strong positive impact on the planning time.

Table 6.1: Mean planning time of the APF base algorithms and RVF algorithm improvements for all obstacle scenario's

Number of Random obstacles, Radius size in meters		*#5* *Radius 3 (m)*	*#10* *Radius 3 (m)*	*#15* *Radius 3 (m)*	*#15* *Radius 1 (m)*	*#30* *Radius 1 (m)*	*#60* *Radius 1 (m)*	**Total of all obstacle scenario's**
Mean planning time improvements (ms)	**Original APF [13]**	223	470	789	552	1342	3869	7245
	Forced Direction	235	502	834	584	1566	4848	8570
	Vector Field original [5]	1280	2162	3192	2501	4585	8853	22573
	Time optimization	182	300	413	319	564	1078	2856
	Clustered rotation	193	330	457	329	600	1162	3071
	Outward potential	195	335	470	330	611	1178	3119
	Gridsize dependency	183	312	449	316	589	1173	3022
	Path optimization	180	304	438	309	571	1151	2953
	Gain Tweaked	175	302	433	296	531	1035	2772

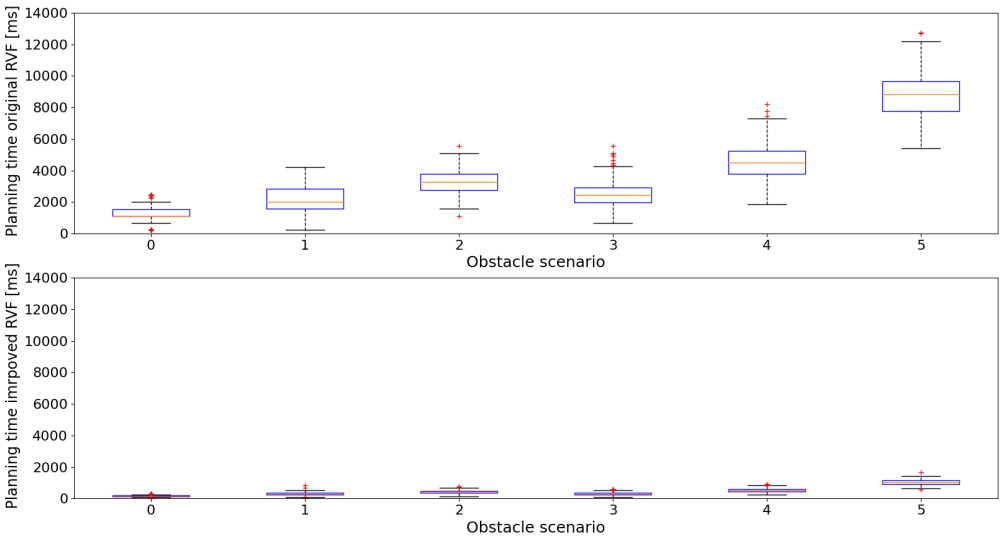


Figure 6.6: Planning times on different obstacle scenario's for the original and improved RVF algorithm (500 runs)

6.7.2. Path length

Tab.6.2 shows the performance results of the improvements on the path length. Each improvement introduces a small increase in path length. It can be concluded that this small increase is again created by the upward bias in path length due to an increased reachability. The last two improvements reduce the path length, while again the reachability is increased. These measures decrease the path length as intended. More important, the path optimization reduces the amount of corners which is not shown in this table. This improvement benefits the efficiency of LZ. Overall the path optimization is functional but has minor influence on the performance of the algorithm, especially when taking into account the upward bias from the increased reachability. However, the final RVF algorithm is close to the path length as presented by the original APF algorithm for the simple obstacle scenario's. The distribution of the path lengths for different obstacle scenario's can be found in the boxplot representation shown in Fig.6.7. It shows the outliers that are the result of solving more difficult paths.

Table 6.2: Mean path length of the APF base algorithms and RVF algorithm improvements for all obstacle scenario's

Number of Random obstacles, Radius size in meters		*#5* *Radius 3 (m)*	*#10* *Radius 3 (m)*	*#15* *Radius 3 (m)*	*#15* *Radius 1 (m)*	*#30* *Radius 1 (m)*	*#60* *Radius 1 (m)*	**Total of all obstacle scenario's**
Mean path length improvements (m)	**Original APF [13]**	58.69	59.89	61.05	57.93	58.96	60.27	356.79
	Forced Direction	65.74	73.58	81.35	864.37	76.10	99.88	461.02
	Vector Field original [5]	59.98	61.73	62.90	58.59	59.79	61.58	364.57
	Time optimization	59.98	61.73	62.90	58.59	59.79	61.58	364.57
	Clustered rotation	60.99	65.46	68.96	58.89	61.28	65.75	381.33
	Outward potential	61.07	65.83	70.0	60.16	63.57	69.41	390.04
	Gridsize dependency	60.79	65.27	69.6	60.06	63.2	70.29	389.21
	Path optimization	60.38	64.36	68.24	59.52	62.23	68.58	383.31
	Gain Tweaked	59.84	62.88	66.32	59.22	61.21	64.71	374.18

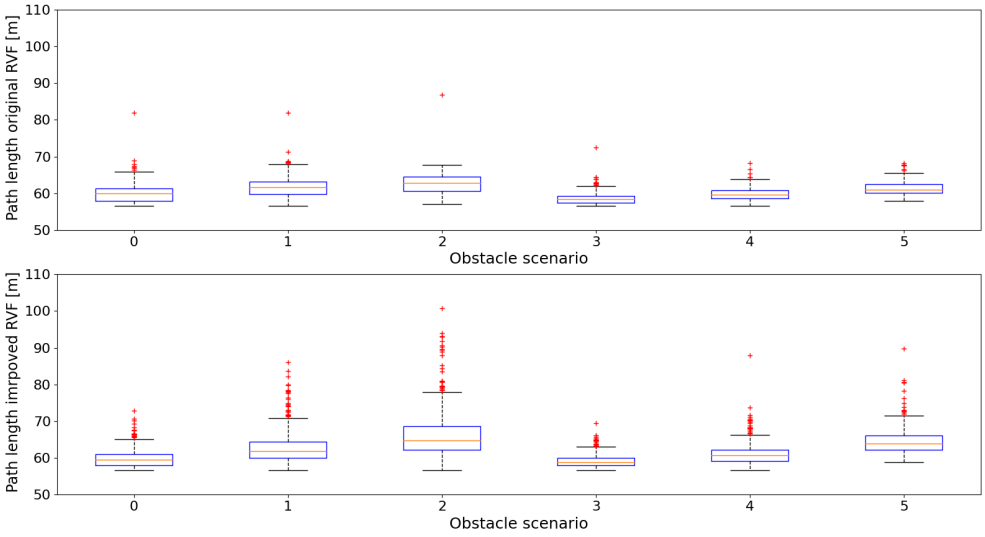


Figure 6.7: Path lengths on different obstacle scenario's for the original and improved RVF algorithm (500 runs)

6.7.3. Reachability

The results of the reachability analysis are presented in Tab.6.3. Beside the planning time, the reachability was a bottleneck for the application of the rotational vector field algorithm. furthermore, the reachability was a concern for any APF based algorithm and therefore the initial problem that had to be solved. The improved RVF algorithm shows an increased reachability compared to both the original APF and original RVF algorithms. In some cases the reachability almost reaches 100%. All in all it can be concluded that the suggested improvements reduced the amount of non-found paths and therefore increases the reachability.

The total amount of runs for all obstacle scenario's is six times 500 is 3000 runs. The amount of failures of all obstacle scenario's combined are presented in the last column of Tab.6.3. With 1336 failed runs, the APF algorithm has a reachability of 55%. The forced direction APF a reachability of 76% and the original RVF accomplishes a reachability of 55%. The final RVF algorithm has a reachability of 90%, which is a significant improvement compared with the other three algorithms.

Table 6.3: Reachability of the APF base algorithms and RVF algorithm improvements for all obstacle scenario's

Number of Random obstacles, Radius size in meters		*#5* *Radius 3 (m)*	*#10* *Radius 3 (m)*	*#15* *Radius 3 (m)*	*#15* *Radius 1 (m)*	*#30* *Radius 1 (m)*	*#60* *Radius 1 (m)*	**Total of all obstacle scenario's**
Amount of NaN's improvements (out of 500)	**Original APF [13]**	98	235	348	79	186	390	1336
	Forced Direction	23	131	253	8	50	256	721
	Vector Field original [5]	89	254	374	51	190	389	1347
	RVF Time optimization	89	254	374	51	190	389	1347
	RVF Clustered rotation	21	95	201	19	66	241	643
	RVF Outward potential	3	34	114	8	63	252	474
	RVF Gridsize dependency	2	37	106	13	61	251	470
	RVF Path optimization	1	37	106	12	57	244	457
	RVF Gain Tweaked	1	9	40	12	51	181	294

6.8. Improvement conclusions

Looking at the presented results in Sec.6.7 and the results of the forced direction and original APF algorithms in Tab.5.1, the RVF algorithm stands out. Overall it performs significantly better in planning time than the original and forced direction APF algorithms with improvements of at least 62%. Regarding path length, it is not yet as optimal as the original APF algorithm, but performs better than the forced direction APF algorithm. Last but not least the reachability is clearly improved and outperforms in almost all cases both algorithms. The total reachability of all obstacle scenario's combined for the improved RVF algorithms even reaches 90%. There is a slight benefit for the predefined direction APF algorithm in two obstacle scenario's. However, putting this in perspective with all the other improvements and results, the improved rotational vector field algorithm is an obvious winner for the path finding algorithm in the Lunar Zebro mission.

7

Architecture & C/C++ Implementation

The conclusion in Ch.6 discusses a suitable improved RVF implementation for a path finding algorithm in Lunar Zebro. Having a structured written algorithm and pseudo code, does not automatically mean it can be directly used in Lunar Zebro. Before the deployment of the algorithm in Lunar Zebro, several steps have to be undertaken. These steps and challenges are discussed in this chapter with the help of the software development process that is used during the development of code. The implementation conclusion will be drawn at the end of this chapter based on the information discussed. The discussed objectives are the used environment, algorithm architecture and the implementation challenges that are still unsolved.

7.1. Process

Before getting into the architecture of the code, the process of software development and application on the LZ path finding algorithm is explained in detail. In general the process consists of six stages, which are described below:

- **Requirements:** this is the first stage in development. During this stage the software architect will gather relevant information around the project. From the information a list with requirements is made.
- **Design:** the requirements are translated into a relevant design and software architecture.
- **Implementation:** the design that meets the requirements is now implemented in actual code.
- **Testing:** the software written in the implementation stage is now tested and verified to be bug free.
- **Deployment:** the tested software is deployed on the intended platform and is used according to the defined purpose.
- **Maintenance:** During the lifetime of the software, it is updated and maintained to ensure proper functioning and staying relevant for the intended application.

With this process and the explained stages, the software state of the current written code can be determined. Ch. 4 explains the first step as gathering data by simulating the algorithms with the help of the Python programming language. One of the given requirements of a finished algorithm is the deployment in a Lunar Zebro rover. As this can only be achieved if the code is written in C/C++, the presented path planning algorithm has to be rewritten in C or C++.

The Python implemented code would be ready for deployment as the implementation is completed and the code is fully tested. Therefore the Python code can be defined as being in the deployment stage. However, for a C/C++ implementation the code is still to be rewritten. Therefore the C/C++ code is at the implementation stage. The requirements for a C/C++ implementation are set and the design is identical to the developed Python code. Further work would be the implementation in C/C++, which is followed by testing, deployment in the rover and maintenance. Before the implementation of C/C++ is created, the Python implementation will be explained in detail.

7.2. Environment

The IDE that is used is VSC. To use VSC as an IDE for a Python written code, some extensions must be installed. The following extensions are installed to enable full operation of the Python language:

- Python: Version 2022.18.2
- Pylance: Version 2022.12.20
- Isort: Version 2022.8.0

With the installation of these extension, the IDE is ready to be used with Python. During the development of code, existing software functionality is used. This reduces coding time and ensures the correct use of legacy code. This existing code can be added in the form of imported packages. These packages are imported in the beginning of the code file and can be found after the keyword $import$. However, just importing this legacy code in the Python files is not sufficient. As there are many existing packages, these are not pre-installed in the coding environment. It therefore requires a separate downloading and installing of these packages. The IDE provides the possibility to generate a $requirements.txt$ file which contains all the necessary packages and corresponding versions. This file is shown in App.C.1. Installing the correct version of these packages allows the Python implemented path planning algorithm to run in the created environment.

7.3. Planning algorithm architectures

Ch.4 proposed three base algorithms that are researched. Ch.5 proposed two improved algorithms based on the Artificial Potential Field algorithm. In total these two chapters extensively describe five path finding algorithms. These algorithms are fully coded and tested such that they can be used for comparison with each other. This is executed to generate data that can be used to determine which algorithm performs the best in given conditions. To ensure a fair comparison and a structured code each algorithm is build up with the same architecture. Each algorithm is described in a separate $.py$ file and consists of three building blocks: a Planning class, a Rover class and a Main function. The Planning class is different for each algorithm while to Rover class and Main function are identical in each file. The only difference in the Main functions are the parameters and function names which are deducted from the planning class. Each building block is described in detail in the next three sections.

7.3.1. Planning class

This class contains the actual search algorithm that must find a path between the start and the goal while avoiding obstacles. Each planning class has different function or methods. However, they all at least have the following common methods:

- An initialization method: This is where the parameters, that are available for all methods in the class, are initialized.
- A path planning method: This contains the actual algorithm that determines the next step in the path that has to be walked to reach the goal.
- A map designing method: This method makes a map for the rover and inserts the obstacles when they are detected. Different planning algorithms use different kind of maps to represent their obstacles. More detailed information about these maps is found in Ch.4.
- A ”create rover path” method: The last step for the planning algorithm is to output the feasible path. The path changes as soon as an obstacle is detected. This method always has an array of path point for a complete path from start to goal as output, independent of the rover location on the path.
- Graph methods: A few methods that initialize a graph and show the search and path planning of the algorithm. This methods are only used when the user desires an animation of the path planning.

These are not the only methods described in the planning class. However, these are the most relevant methods, while others are always used as an embedded method in one or more of the above stated methods.

7.3.2. Rover class

The Rover class is identical for each planning algorithm and has several methods that ensure the possibility of simulating the rover. These methods are used to show the rover in a graph and simulating the behaviour of a Lunar Zebro rover to define a realistic testing environment. The following methods define the characteristics for this class:

- An initialization method: This is where the parameters, that are available for all methods in the class, are initialized.
- A method to get the rover: This method creates the rover with the corresponding view area based on the orientation and position of the rover.
- A method for obstacle checking: This method defines if the view area of the rover is intersecting with a yet unknown obstacle. If this is the case, the obstacle will be added to the known obstacle list. With this information the planning algorithm can replan the path to the goal if necessary.
- A method to simulate the rover: When the user desires to see an animation, the rover is shown and simulated in the graph by this method.

7.3.3. Main

The Main function uses both classes to create a planning object from the planning class and an LZ_rover object from the Rover class. To get a better insight in how these classes are used and at what moment the actual path planning is considered, an algorithm description is made. This pseudo algorithm can be found in Alg.10. The main function consists of two parts. The first part is described between lines 1 and 13. This software block initializes all relevant parameters, the used planning algorithm, the LZ rover and if desired the shown graph. The second software block is the $while$ loop that is described between lines 17 and 43. The $while$ loop ensures the walking of the planned steps on the $RoverPath$. Each loop corresponds with one single rover step. Each step checks possible collisions and simulates the rover in a graph if desired. The loop is terminated when a path to the goal is found or if the algorithm failed to find a path. In the last case, the loop terminates with an empty $Path$ list. All individual functions that are used to present the main pseudo algorithm are the explained methods in the planning class(7.3.1) and rover class(7.3.2) sections.

Algorithm 10 Main function in planning algorithm files

Require: Path planning class and the Rover class
Ensure: Planned and shown path between start and goal.

$ShowAnimation = True$
$DefineFieldParameters(Start, Goal, GridSize, AreaBound, UnSeenObstacleList)$
$DefineRoverParameters(RoverRadius, ViewDistance)$

$Planner \leftarrow InitiatePlanningClass(Gridsize, RoverRadius, AreaBound)$
$Path \leftarrow Planner.PathPlanning(Start, Goal, SeenObstacles)$
$RoverPath \leftarrow Planner.CreateRoverPath(Path)$

$LZRover \leftarrow InitiateRoverClass(Start, Goal, Path, UnseenObstacleList, SeenObstacleList)$
if $ShowAnimation$ **then**
    $Planner.GraphInitialization$
    $Planner.DrawGraph(Start, Goal, Path, UnseenObstacleList, SeenObstacleList)$
end if

$Finished = False$
$i = 0$
while $NOT(Finished)$ **do**
    $LZRover.SimulateRover(RoverPath, i)$
    $ReplanPath = LZRover.ObstacleCheck(UnseenObstacleList, SeenObstacleList)$
    if $ShowAnimation$ **then**
        $LZRover.SimulateRover$
    end if
    if $RaplanPath$ **then**
        $NewPath \leftarrow Planner.PathPlanning(RoverPath[i], Goal, SeenObstacles)$
        if $NewPath == NaN$ **then**
            $RoverPath \leftarrow EmptyList$
            $Finished = True$
            Continue
        else
            $RoverPath \leftarrow Planner.CreateRoverPath(NewPath, RoverPath, i)$
            if $ShowAnimation$ **then**
                $Planner.DrawGraph(RoverPath[i], Goal, Path, ObstacleLists)$
            end if
        end if
    end if
    if $ShowAnimation$ **then**
        $LZRover.RemoveRover$
    end if
    if $i >= Length(RoverPaht) - 1$ **then**
        $Finished = True$
    end if
    $i = i + 1$
end while

7.4. Performance comparison architecture

With these separate Python files for each planning algorithm, it is possible to run each algorithm on different obstacle scenario's and check their behaviour. However, it is not possible to compare all these algorithms with each other yet. Comparison between the algorithms generated the relevant information for a fully substantiated conclusion. To obtain this comparison a separate comparison code is written. This file imports all planning algorithms by importing the individual planning algorithm files. The goal of this file is to form a valuable comparison between the planning algorithms based on the different performance metrics. This file again contains the Rover class as described in previous section and a

main function. Every planning algorithm is assigned an LZ_rover object in the main function. If five different planning algorithms are compared, five different rover objects will be created.

7.4.1. Main function

Essentially this function is similar to the main function of the separate planning algorithm files as described in Sec.7.3.3. The difference can be found in planning all algorithms in one $while$ loop and execute the planning 500 times to create data sets instead of single data points. The second objective is recording the performance metrics: planning time, path length and success/fail of the algorithms in each loop or run. This information is saved in lists and used in further comparison between the planning algorithms.

7.4.2. Obstacle scenario's

As described in Ch.6 there are six different scenario's where the amount of obstacles and size of obstacles is varied. Each scenario is simulated 500 times, where the obstacles are placed randomly even distributed over the field. As these experiments must be repeatable, again the saved obstacle locations of previous experiments are used. The main function loops over all different scenario's. Depending on the scenario, the corresponding csv file is opened and the obstacle locations are subtracted from this file. Independently of the computer and/or operating system the obstacle locations of all scenario's and maps are identical. This ensures identical results for path length and reachability. Obviously the planning time is still depending on the machine specifications that is running the simulation.

7.5. C or C++ implementation

As discussed in the beginning of this chapter, the rover deployment stage can only be reached when the algorithm is written in a language that is able to run on the rover. Python is a high-level language and is mainly developed to write code for (large) projects to keep code development easy and structured. Furthermore, it is extremely useful for the generation of data on a large variety of topics. This is exactly why Python is used for generating the data on the performance of different algorithms. However, Python is not a hardware focused language and is therefore not optimal in communicating with actual hardware. When implementing software in the Lunar Zebro microcontroller, it must communicate with the hardware in the rover. This ensures that the microcontroller can send information to the actuators and gather information from the sensors. For this reason C or C++ is the chosen language for implementing code on the microcontroller of Lunar Zebro. C and C++ are low level languages that are extremely powerful for communicating with hardware without the use of extensive computational power. Before deployment of the final planning algorithm in Lunar Zebro, the planning algorithm has to go back to the implementation stage such that it can be rewritten to C or C++ implementation.

7.5.1. Planning class implementation

In the implementation state it is decided which parts of the code are rewritten in C/C++. The major part of the written code in Python is used only to gather data and compare the functioning of different algorithms with each other. Looking from a Lunar Zebro perspective, the only relevant parts of the code are the algorithm planning classes. Ch.6 concluded that the improved rotational vector field algorithm is the path planning algorithm for Lunar Zebro. The improved rotational vector field planning class is therefore the only class that has to be rewritten. As described in Sec. 7.3.1 a part of this class is also used to initialize and draw the map for the simulation. Obviously this is not needed while the planning algorithm is running on the rover. The software part that has to be rewritten in C or either C++ is significantly reduced compared to the software that is written to test and optimize the algorithm in Python. From this section forward, the implementation of the C or C++ code will only be focused on the rotational vector planning class. Methods for initializing and drawing the graphs are excluded from the C or C++ implementation.

7.5.2. Language preference

The preference between C or C++ is extremely dependent on the exact implementation in Lunar Zebro. When using the planning class in a single rover it does not need multiple objects, as it is just used for one single rover. When implemented in a single rover, the planning algorithm can be seen as a single function with some globally available variables. A single operating rover could therefore be provided

with a C implemented planning algorithm. However, if the full functionality of a class is required, it is recommended to implement the planning algorithm in C++. C++ is an object oriented based code language, which makes coding with classes much easier than in C. The second advantage of a C++ implementation is the availability of a larger standard library with predefined functionalities.

7.5.3. Challenges

The requirements for an LZ path planning algorithm state that the final implementation has to be programmed in a low level language. During development in Python this was taken into account by minimising the usage of packages and predefined function in the planning algorithms. However, due to the simplicity of some predefined functions and the difference in programming language, there are still a few challenges that need attention. The challenges for the rotational vector field planning algorithm are discussed in this section.

Class variables
Variables or methods in Python that are available for the whole class are denoted wit the keyword $self$. In C/C++ these have to be set as $member$ variables and $member$ functions respectively. These can be set $public$(accessible from inside and outside the class) or $private$(accessible only inside the class) depending on the use case. These $member$ functions and variables must always be declared at the initialization of the class.

Lists
Lists in Python are preferable in case small amounts of data need to be stored. Lists are extremely convenient for data manipulation and storage of heterogeneous data types. However, in a C/C++ implementation, array's are preferable as these are faster in the sense of computational time. As there is no data manipulation in the planning class and the data in the lists are homogeneous, array's are recommended in the final implementation.

The Numpy package
The $Numpy$ package contains mathematical equations and formulas that can be used right away in code. However, when programming in a low level language it is advised to implement these functions directly in the code to minimise computation time. The following functions are used and can be implemented with use of simple mathematics.

- **Hypot** is the function which returns the euclidean norm between the origin and a given X,Y coordinate.
- **Norm** is the function which returns the norm or length of a given vector.
- **Dot** is the function which returns the dot product between two given vectors.
- **Arccos** is the function which returns the inverse cosinus value in radians of the given x value

Deque
The double ended queue has the possibility to remove and add elements from both sides of an array. In C++ this function is available in the standard template library. In C this functionality can be achieved by making a simple function that adds or removes elements from the beginning or the end of the array depending on the user request.

Set
A set is a list or an array that only contains unique values. It is not possible to have two of the same values on different locations in the list or array. In C++ this function is available in the standard template library. In C this functionality can be achieved by making a simple function that is not allowing the addition of new elements if they are already present in the set.

Process time
Currently the process time of the algorithm is timed. As this is not relevant for functioning of the algorithm, this has no mandatory implementation. However, it could be beneficial to time the process time when a final implementation is made to check the time improvements when implemented in a low level language. To achieve the same timing functioning as the Python code, the $high\_resolution\_clock :: now()$ from the $chrono$ package could be used for a low level implementation.

7.6. Implementation conclusion

Writing the rotational vector field planning algorithm in a lower level language is the last step for deployment of the algorithm in Lunar Zebro. With the current algorithm written in Python, the challenges for rewriting to a lower level algorithm are not a showstopper for correct operation of the algorithm. To gain full potential of the algorithm it is recommended to use a C++ implementation. However, a C implementation would be possible but requires some extensive coding compared to a C++ implementation. If the low level written code is deployed in Lunar Zebro, the code must be tweaked and maintained. Parameters can be adjusted to the desired behaviour of the rover and it is recommended to maintain the code for optimal functioning.

8

Conclusion

Lunar Zebro is a unique six-legged rover with the objective of operating in a swarm on the moon. The unique characteristics are formed by the size, walking behaviour and swarming capabilities. The first mission will focus on walking 200 meters in one Lunar day autonomously with a single rover. Ever since the first moon rover landed on the moon, the localization and navigation on the moon environment has been a challenge. The navigation problem for LZ was partly addressed in this thesis. A path planning algorithm was researched and developed, which enables the rover to construct a path to the target location while avoiding the present obstacles. The final suggested path planning algorithm is an improved artificial potential field type of algorithm. The suggested algorithm constructs an attractive vector field towards the goal and a repulsive rotational vector field around obstacles. The so called Rotational Vector Field algorithm accomplishes local path planning to the target location with any desired grid accuracy and relatively low computational power. The RVF algorithm accomplishes a 62% reduction in planning time and a similar path length compared to the traditional Artificial Potential Field algorithm. Furthermore, the reachability of the RVF algorithm is 90%, while the traditional APF just accomplishes a reachability of 55% for the tested obstacle scenario's. The proposed algorithm is fully tested and ready for a low level implementation without compromising possible swarm functionality. It can therefore be concluded that the objectives and functionality for a path finding algorithm in LZ are achieved.

8.1. Recommendations and future work

The remainder of the work can be split up in three parts. The first one being described in Ch.7. The next step in the development process would be the deployment and testing of the path planning algorithm in a LZ rover. For deployment of the algorithm, it has to be implemented in a low level programming language. This programming language can either be C or C++, depending on the desired implementation for LZ. When addressing the challenges presented in Sec.7.5.3 with the suggested solutions, a simple and robust implementation in either C or C++ can be achieved.

The second recommendation is improving the performance of the rotational vector field algorithm. The motive for developing the rotational vector field was increasing the reachability, decreasing the planning time and decreasing the path length. The applied improvements have had a significant positive affect on all three performance metrics. The choice of taking an APF type algorithm was mainly made on the large variety of possibilities that are offered by this algorithm type. As time was the bottleneck for improving the algorithm even further, it is suggested to research further development to exploit the full potential of the RVF algorithm. A few suggestions for further performance improvement are given in Sec.4.4.2

Last but not least, the Lunar Zebro is designed for operation in a swarm. Smooth implementation of swarm behaviour is one of the beneficial possibilities that the RVF algorithm is offering. To take advantage of these possibilities, additional functionalities could be implemented. Two powerful functionalities could be swarm computation or path improvement by swarm behaviour. It is suggested that these two functionalities are deployed in the rovers as soon as the path planning algorithm is fully deployed in a single Lunar Zebro.

A

State of the art research

A.1. Environment modeling methods

In this Appendix section each environment modeling technique will be explained in more detail. The application of the methods is discussed in Ch.3

Grid model
The environment is split up in a grid, every grid cell has the same size. Each cell could either be free or blocked duo to an obstacle. The model can be used in a 2D or 3D environment. This model contains a lot of data as each cell has to be presented, when a higher accuracy is desired the grid gets a higher resolution. A higher resolution results in more points and therefore more memory is needed.

Cell tree model or cell decomposition approach
This approach is similar to the grid model. However, the cells are of different sizes. This reduces the amount of used memory. If a cell is filled with an obstacle the cell will be split up in 4 equal squares. If a large cell is free it will not split up and the algorithm can assume a large free space. This way the amount of unnecessary data is reduced. For 2D environments, the quadtree is sufficient and for 3D environments, the Octree is sufficient.

Voronoi diagram model
The obstacles edges and the barrier of the work field are set as boundaries. All vertices are constructed to be of equal distance away from three or more barrier boundaries. Edges are constructed between two boundaries, with equal distance to the boundaries [4]. It results in high calculation speed but does not guarantee a shortest path. It is more difficult to use in a dynamic environment than in a static environment, but can be used in 2D and 3D environments.

Tangent Graph method
All tangents points of obstacles and the environment barriers are connected(edges) if they are not intersecting an obstacle. These edges represent the possible path that can be used for path planning. It can not results in an optimal path between start and goal. Can be used in 2D and 3D environments.

Visibility graph model
Is a bit more extensive as the Tangent graph as every obstacle is represented as a polygon. All corners of the polygons and the environment barrier are connected. All these edges form again the possible paths. Can also be used in 2D and 3D environments.

Free Space Approach
The free space between obstacles within the environment is represented by only convex regions. The midpoints of the convex regions are used as nodes. The connection between these nodes are the links. This is called a MAKLINK graph, which can be used to determine a path between start and goal through a field of obstacles. It is a flexible method which can adapts quickly in dynamic environments. However, in an environment with a lot of obstacles this approach may fail.

Topological Method
This method is based on reducing dimensions and therefore reducing complexity. Instead of making nodes of all possible points, this method looks only to relevant spaces, dimensions, obstacles, etc. Than nodes and edges are created. This method is hard to use in a changing environment because the process of establishing the topology network is complex.

Probabilistic Roadmap Method
Nodes between start and finish point are randomly created and are obtained milestones. The method starts to connect start and finish point to nodes surrounding S(start) and F(finish). Path planning can than be done by searching a path that connects S and F. The accuracy is dependent on the maximum length is allowed between an existing point and a following random generated point.

A.2. Navigation algorithms

In this Appendix section each relevant navigation algorithm will be explained in more detail. The application of the methods is discussed in Ch.3

Dijkstra Algorithm
Edges between nodes have a value that is determined by a weight function. The algorithm applies a greedy strategy to determine the sequence of edges that result in the minimized sum of the edge weights. The optimal edges are saved in an array. The algorithm has a high success rate as all nodes are considered. However, this algorithm becomes very inefficient in large environments.

A* Algorithm
A* is an improvement on Dijkstra's algorithm. It adds the estimated cost of the target point to the current node. The algorithm makes use of an evaluation function which is f(n) = g(n) + h(n). g(n) is the actual cost from initial node to node n and h(n) is the estimated cost from node n to the target node. When minimizing this, it can be guaranteed that the search will always proceed in the direction of the target node. Therefore it is more efficient than Dijkstra's algorithm. It is best suitable for global search in a static environment.

D* Algorithm
The D* algorithm is in principle the same as the A* algorithm. However, the D* algorithm is very effective in searching a route in a dynamic environment. So this is a combination of global planning and local information. According to some papers this is still only a global planning algorithm. There is also a D* lite algorithm which is focused on a changing starting point in time with a fixed target point. D* lite is not very applicable for local planning. When the local environment is carefully planned it caused significant loss in time and efficiency.

Level set Method
The level set method is typically used in the definition of a front between two contours or planes in either 2D or 3D. This front can be represented as single point where the accuracy is obviously determined by the concentration of points on the front. The level set method is used to add dynamics to the front and mimic the behaviour between the surface motions [24]. [19] propose a time-optimal path planning by making use of the level set concept.

Fast marching algorithm
Very similar to Dijkstra Algorithm, but it is not updated with the euclidean distance between two nodes. It uses approximate partial differential equation reduced by the nonlinear Eikonal equation[26].

Boustrophedon Decomposition Algorithm
Commonly used and simple coverage path search algorithm. It transforms the configuration space into cell regions. Each cell is than completely covered by a path in ”zig-zag” motion [6]. This algorithm is very use full if an area need to be searched and completely covered. But is not optimal regarding shortest path problems from start to goal nodes. The coverage path determined by this algorithm does not contribute to the goal of Lunar Zebro to reach a target, preferably with the optimal path length.

Internal Spiral Algorithm
This algorithm is similar to the Boustrophedon Decomposition Algorithm, it moves within the boundaries of a covering area. The difference is the movement to cover the area, it goes spiral and ends therefore in the middle of the covered area. Similar to the Boustrophedon Decomposition Algorithm, it is not optimal regarding shortest path problems. Again the coverage path determined by this algorithm does not contribute to the goal of Lunar Zebro to reach a target, preferably with the optimal path length.

Fuzzy
This method is deducted from the Boolean logic, in Boolean logic values can either be 0 or 1. In fuzzy logic it can also be 0.5 or every other value between 0 and 1. The algorithm contains a set of rules that are used as reference and that are based om human expert knowledge. The usage and output of rules is dependent on the values between 0 and 1. First the input is ”fuzzified” such that the rules can apply and after this the output is ”defuzzified” to a relevant output for the actuators. It does not need an accurate mathematical model and good results can be obtained regarding obstacle avoidance. However, it can not adapt to the environment and is very dependent on the rules that are given beforehand. Which means in complex environment is looses its robustness. Note that the fuzzy rule set is commonly used as an addition or in combination with another (path finding) algorithm. [20], [15] and [3] propose a few examples of these possible implementation of a fuzzy rule set with another algorithm for navigation problems. The fuzzy rule set is not commonly used on its own to solve the navigation problem. However, this does not mean it cannot be used for navigation problems. During the improvement of basic algorithms, fuzzy rule sets can still be considered as an improvement or addition.

Probabilistic Roadmap Method
This method is already described as a method to model the environment. However, it can also be used as a path finding algorithm. It first establishes a random roadmap and uses the A* algorithm to form a path. As already said the forming of the nodes is random and therefore the path will not be completely optimal in terms of length.

Rapidly exploring random tree
This makes use of spatial search technique. It expands the tree during the search while the endpoint is not yet known. This is also not the optimal path regarding path length, as it is still based on random generated nodes. This algorithms also has a relatively high computation resource consumption. Many forms of the RRT algorithm exist that improve the algorithm for different environments, such as dynamic vs static or 2D vs 3D environments.

Artificial potential field
The basics consist of a potential function that has an attractive force to the target location and a repulsive force from obstacles. Size and direction of the sum of these two form the next steps in the path finding problem. Very usefull for online obstacle avoidance but has difficulties with dynamic obstacles and is easily trapped in local minima. Large search field require relatively low computation resources compared to other algorithms.

Bug algorithm
It connects the target point and starting point in a straight line. It uses edge tracking of obstacles to avoid them and after the avoidance it continues in a straight line to the goal. The algorithms is not able to find the optimal shortest path, but has relies on minor calculations and has a fast path search.

Genetic Algorithm
It is based on the survival of the fittest strategy by making use of an iterative process search. The optimal solution is found by a random solution and it has a memory function. Genetic Algorithm (GA) can find the global optimum according to the optimal value of the current state. It is able to do local planning and find an optimal path and has rapid global converges speed in early stages but has a slow converges speeds during the end. Also it can fall into local minima easily and has a poor stability.

Differential Evolution
This approach has a lot of similarities with GA. The only difference is that the fitness level is calculated using the difference vectors between individuals. In general it performs better than the GA and is more robust regarding path planning.

Imperial competition algorithm
This algorithm is inspired by the concepts of imperialism and imperialistic competition process[39]. It starts with an initial population that is divided in two groups. One group will have the particles that have the best objective function (imperialists) and the other group(s) will form the colonies of these imperialists. The more powerful the imperialist, the more colonies it will have. Due to competition, the most powerful empires tend to grow while the weaker ones collapse. This leads the algorithm to finally converge into just one remaining empire. It finds optimal paths in less time than the GA algorithm and obtains equal robustness.

Q-learning/dynamic programming
This provides an agent(robot) in a Markov environment, an environment where the next state only depends on the current state. In path planning this relates to the next step on the path is only dependent on the place where the rover is right now. The process has no memory. The ”Q-value” determines how useful an action. for example left,right or forward, is. This is done by making use of the so called state-action value function. The modeling of this algorithm is simple and doesn’t require extensive training. The two main disadvantages are falling into a local minimum and being weak regarding optimal path generation.

Tabu Search
Is similar to Q-learning. However, this algorithm starts with an initial solution and start iteration on this solution. During the iterations it is improving the solution towards an optimum [27]. The initial search can be generated random or with a greedy strategy. The process of optimal search stops when the stop criterion is reached, this means that when a local optimum is found, moves are still allowed. Therefore, falling in a local minimum is less likely. It is a relatively unexplored method for path planning and it is highly dependent on the initial solution.

Artificial neural network
This network is based on human behaviour and realizes the function of nonlinear algorithms using a large number of simulated neurons like in the human brain. It needs training with sample data before it can be used for actual path planning and obstacle avoidance. It has the ability for dynamic obstacle avoidance with the correct training. It has slow processing speed and poor generalization performance. However, it has strong learning and adaptive capabilities and has a high robustness/reachability. It is therefore used in many path planning applications.

Simulated annealing algorithm
This algorithm is based on the cooling down of an object with an initial temperature. With the decrease of temperature an optimal can be found. The equivalent of temperature is a measure of the randomness by which changes are made to the path. High temperature means high randomness. Solutions can differ every search. It has slow convergence speed, long execution time and is highly dependent on the initial value.

Animal based bio-inspired intelligence algorithms
The following algorithms are all based on animals and their colony behaviour. These techniques are mostly applied in swarm to imitate the swarming behaviour of the colonies.

Ant colony Optimization is a probabilistic algorithm that makes use of heuristic search. The fundamental idea behind the ant colony algorithm is that it uses the behavior of a single ant to represent one feasible solution of the path optimization problem, and the behavior of the entire ant colony constitutes the solution space of the problem[4]. As time passes, the algorithm will convert to the shortest path around obstacles. It has slow convergence in the beginning and fast convergence at the end. The total solution speed is still slow.

Particle swarm optimization is a heuristic algorithm based on predation and return of the bird populations. Birds only know their own location and distance to the food. They try to find a bird that is closer to the food and follow it. It can be adopted to obstacle avoidance. It is an easy and robust algorithm and has a fast convergence speed in the beginning and slow convergence speed at the end. It is easy to fall in local optimums.

Wolf pack algorithm makes use of three dominant ”Wolfs” that find the prey first. After that, all omega(less dominant group) ”wolfs” are surrounding the target and coming closer. This way a path can be formed even in a complex search space. The algorithm has good convergence and strong robustness. However, the algorithm is more complex due to relatively more parameters that has to be set.

Bacterial Foraging Optimization is based on bacterial movement which consists of four stages. First the particles can move towards nutrient-rich areas and away from nutrient-poor areas. Secondly when in a nutrient-rich area it will release an attract signal such that more particles can come. Then the reproduction will start, the particles with a low fitness value are in high nutrient areas and will therefore reproduce. In the last step particles are eliminated if they are in harsh environments. It is robust, easy to implement and good at exploitation. However, it has poor convergent behaviour and decreasing performance with an increasing search space and complexity.

Bee Colony Optimization Technique is inspired on the bee colony foraging behaviour, it explores the search space with random samples(bees) and then converges to a solution. This solution can be the target or nectar in case of bees. After the bees have set their target they learn the path to the tarhet such that they can share this path with other bees. Literature proposes that it can outperform GA, Differential Evolution (DE), Particle Swarm Optimization (PSO) or Evolution Strategies (ES) on a large set of unconstrained test functions. However, it is not widely used and described in path planning problems.

Firefly algorithm is based on firefly technique to attract other fireflies by the use of flashing their lights with a certain frequency and intensity. This flashing behaviour can be used as an objective function for an optimization technique. It has low computational cost, it can give an optimal path with good dynamic obstacle avoidance. Again this technique is not very widely used.

SLAM
SLAM, short for Simultaneous Localization And Mapping, seems at first not to be related to a navigation algorithm. However, looking further into the use of SLAM it can used as a basis for a navigation algorithm. One way or another, each navigation algorithm makes use of the location of the rover. This can either be the actual location of the rover or this can be a location that is estimated from the steps that are taken from the navigation algorithm. SLAM determined the location of the rover within a self constructed map of the environment[41]. It can therefore be said that this is the actual location and not the location based on trace-back of steps in the navigation algorithm. So it adds the possibility to a navigation algorithm to determine the exact location and plan a path with this information. [41], [21] and [34] show different implementations of SLAM with a navigation algorithm. However, they also state that the combination of SLAM with a navigation algorithm is still in an early development phase.

Behaviour decomposition method
This based on breaking down the navigation problem in multiple independent sub problems. For example target tracking and obstacle avoidance or breaking up the whole path in smaller pieces. These

solutions together form the navigation algorithm. The definition of this method is to break down the problem in smaller algorithms for different tasks. The sub problems will again rely on the implementation of an actual path planning algorithm. This way of implementing the solution can still be used but will not be considered as an individual algorithm.

Cased-Based Learning Method
A database with predefined cases is set up before deployment of the algorithm. This database contains cases that are relevant for the path planning and obstacle avoidance algorithm. During operation the algorithm tries to find the best matching saved case to the current situation and operates according to the predefined solution. It is not very robust for divergent cases and it needs a lot of data collection beforehand. However, it reduces the search space and works according to the predefined preferences for well-known cases. This method relies heavily on the predefined cases. An artificial network is more robust and can also train on predefined cases. An ANN would be a form of cased-based learning method and therefore considering Cased-Based Learning as an algorithm would be irrelevant.

Rolling Window Algorithm
The local environment, that can be obtained by sensors of the robot, is formed to a window with information. For each window sub-targets are obtained to form a path. This sub-target and path is updated every new window until the task is completed. It has good obstacle avoidance capabilities. However, it may be trapped in local optima and the obtained path is not guaranteed to be the optimal path regarding length. To avoid obstacles, this algorithm still needs another path planning algorithm to find a path in each created window. An example of this is presented in [42], where a artificial potential field algorithm is used as path planner.

Vector-polar histogram
The method uses a two-dimensional Cartesian histogram grid. Each cell in the grid holds a value, the value represents the confidence of the algorithm in the existence of an obstacle at that location. This existence value can make up for uncertainty from the sensors or from far distance readings. With this information the robot plans a path by making use of for example a repulsive force from these obstacles. The algorithm is computationally efficient, robust and it allows continuous motion without stopping for obstacles. It is quite similar to the Artificial potential field algorithm but has a slight difference in obstacle mapping and avoidance [32].

Reinforcement learning
This method allows the agent(robot) to interact with the environment and learn on trial and error base. The agent will be rewarded if it finds a way around an obstacle and keeps its distance to obstacles. It is very effective for unknown environments as it can learn and adapt from the environment. It can deal with any shape or form of obstacles and doesn't need information beforehand. However, before it is able to find a collision free path it needs to learn about the environment. This process is time consuming and hard to validate outside the simulating space(real world).

Deep Reinforcement learning
This combines the properties of Reinforcement learning with an artificial neural network. This way large state space (lot of sensor data) and large action space(lots of movement possibilities) can be handled. The neural network can be partly trained beforehand but the reinforcement decision process needs to be trained by trial and error. Doing this in simulation environment will again be difficult regarding the real world and would again take a lot of time.

Both reinforcement learning and deep reinforcement learning would therefore require training during the mission of Lunar Zebro. Currently training of these algorithms is very time expensive. Looking at the mission of LZ it will be around 30 days. Time expensive training of the rover will possibly harm the goal of the mission. [35] propose a path planning algorithm based on deep reinforcement learning. Where accessing free path will be rewarded and "hitting" obstacles has a penalty. It took the robot 200000 steps of training to reach a reward of around 80% of the maximum possible reward. This amount of training steps for Lunar Zebro would cost around 40 full earth days of training. This is based on sequence walk with interwalk, which is the fast way of walking for LZ and the walking height is set to the average height of 25 mm [2]. So even in the best scenario, it will cost LZ to much time to train and reach the goal safely.

B

Simulation Results

B.1. Base algorithm comparison

This appendix section will show the boxplot analysis of all tested environmental factors. The boxplots present the performance metrics planning time and path length for the A*, RRT and original APF algorithms.

B.1.1. GridSize adjustment results

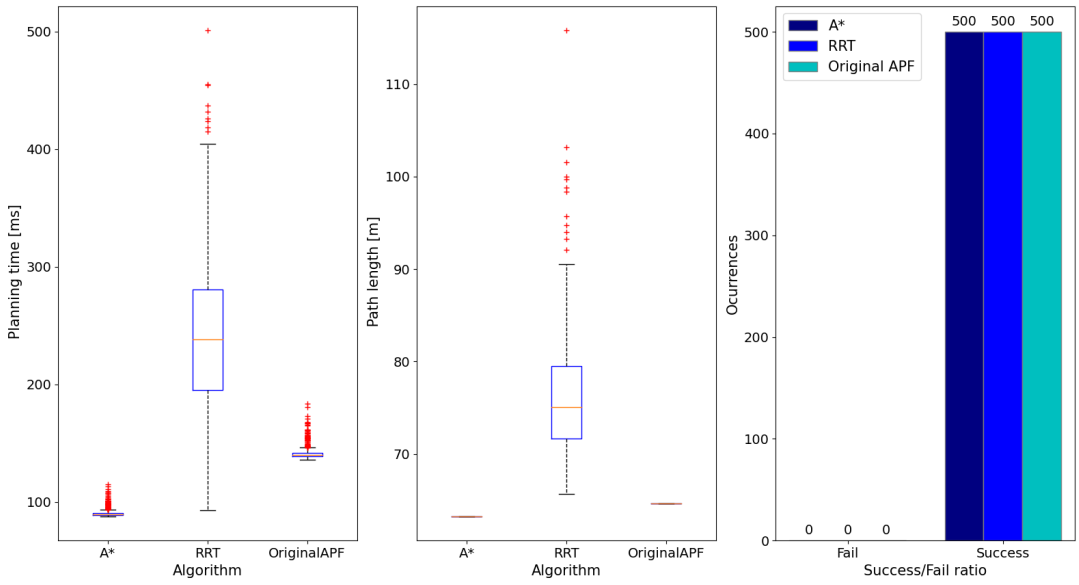


Figure B.1: Algorithm performance with 50x50 grid (standard conditions) for 500 runs

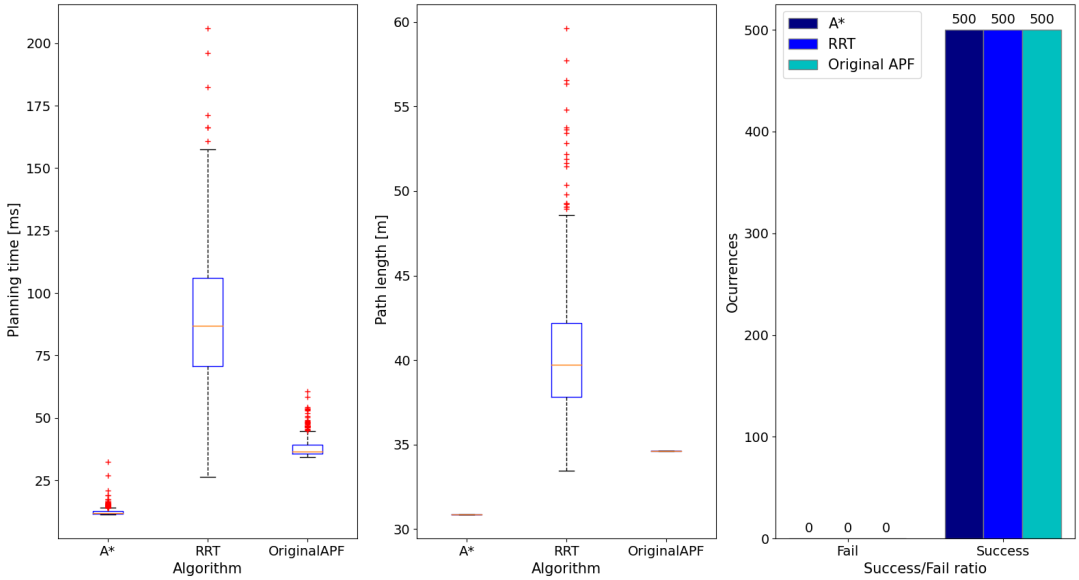


Figure B.2: Algorithm performance with 25x25 grid for 500 runs

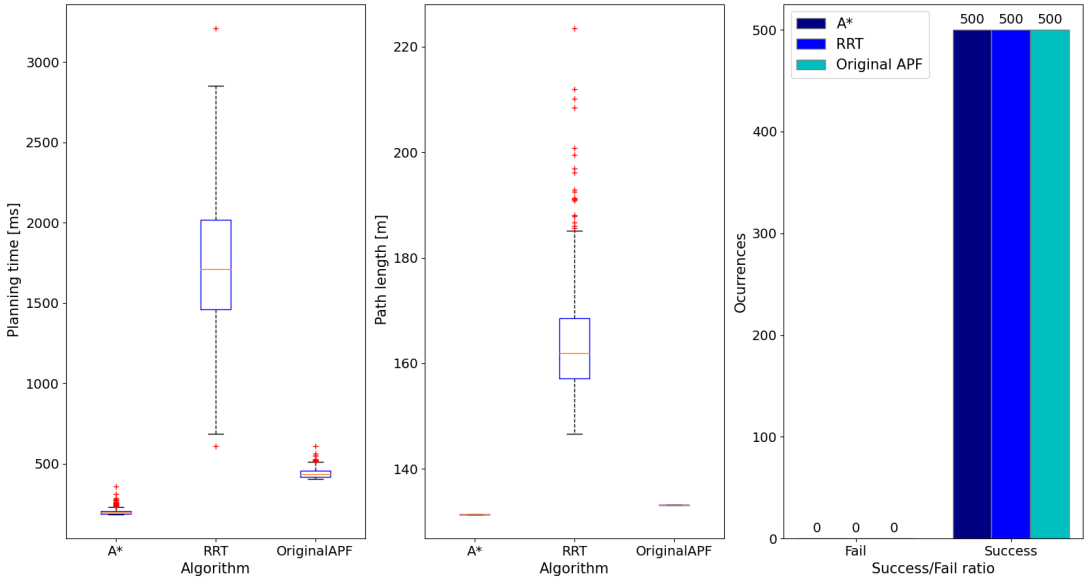


Figure B.3: Algorithm performance with 100x100 grid

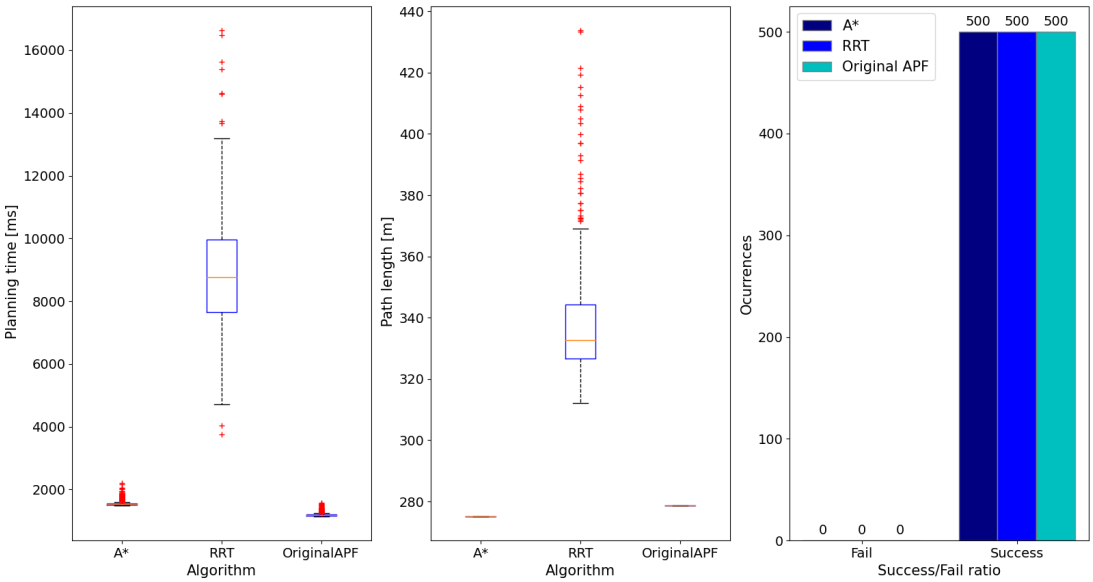


Figure B.4: Algorithm performance with 200x200 grid for 500 runs

B.1.2. Object size adjustment results

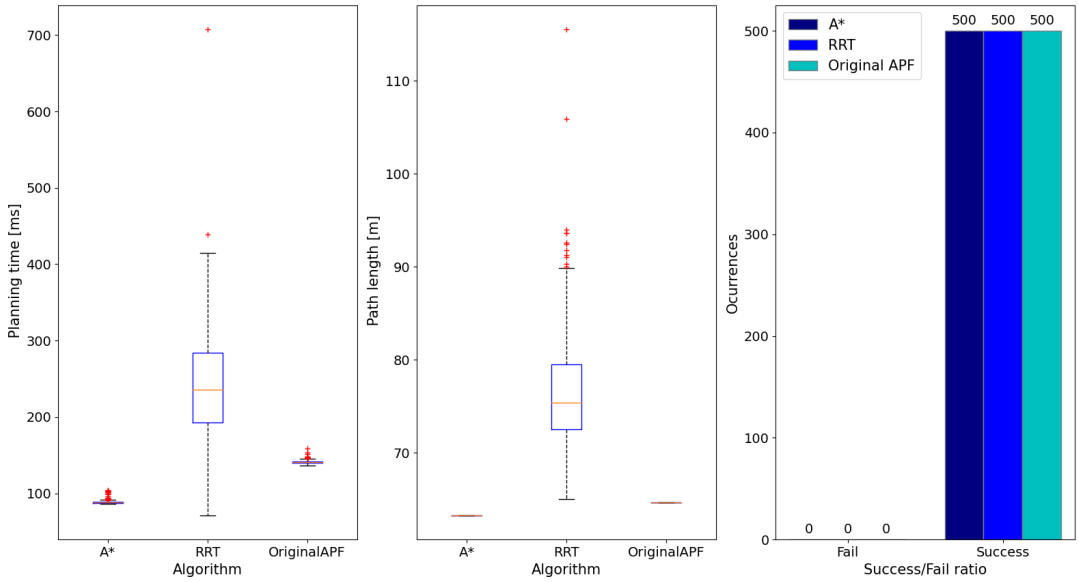


Figure B.5: Algorithm performance with object radius of 3 metres(standard conditions) for 500 runs

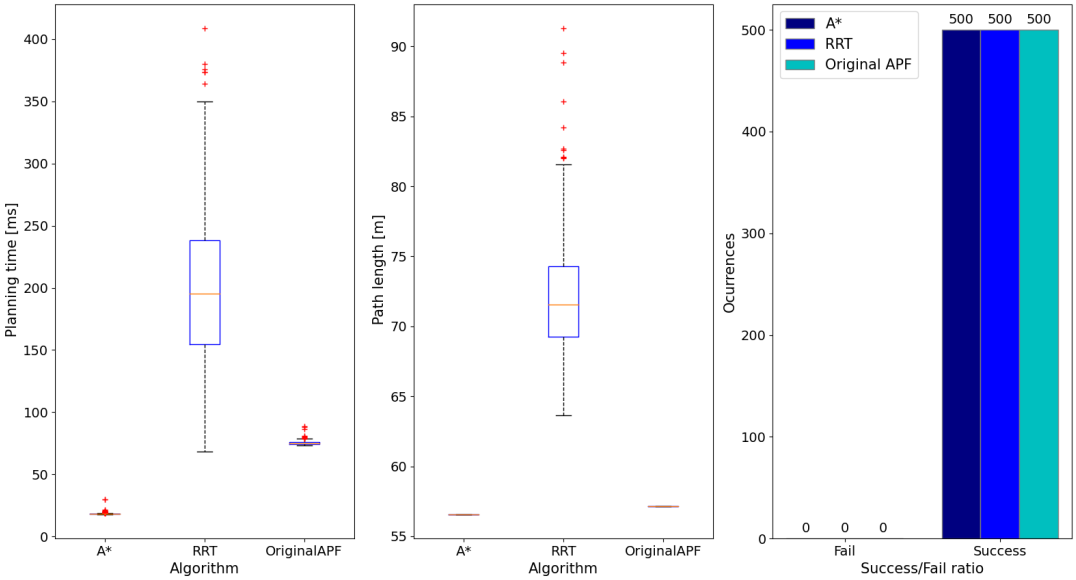


Figure B.6: Algorithm performance with object radius of 1 metres for 500 run

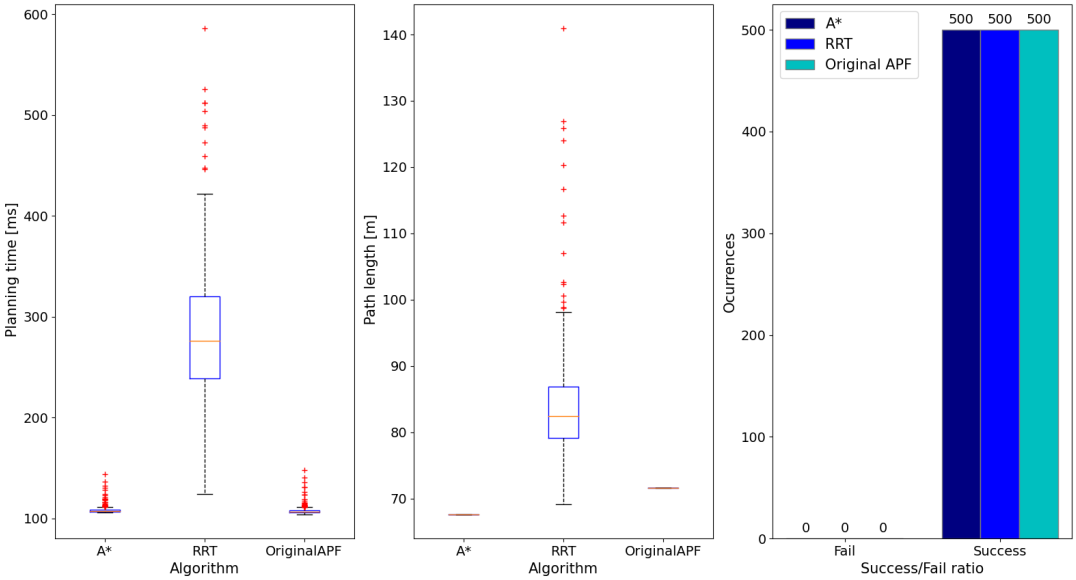


Figure B.7: Algorithm performance with object radius of 5 metres for 500 run

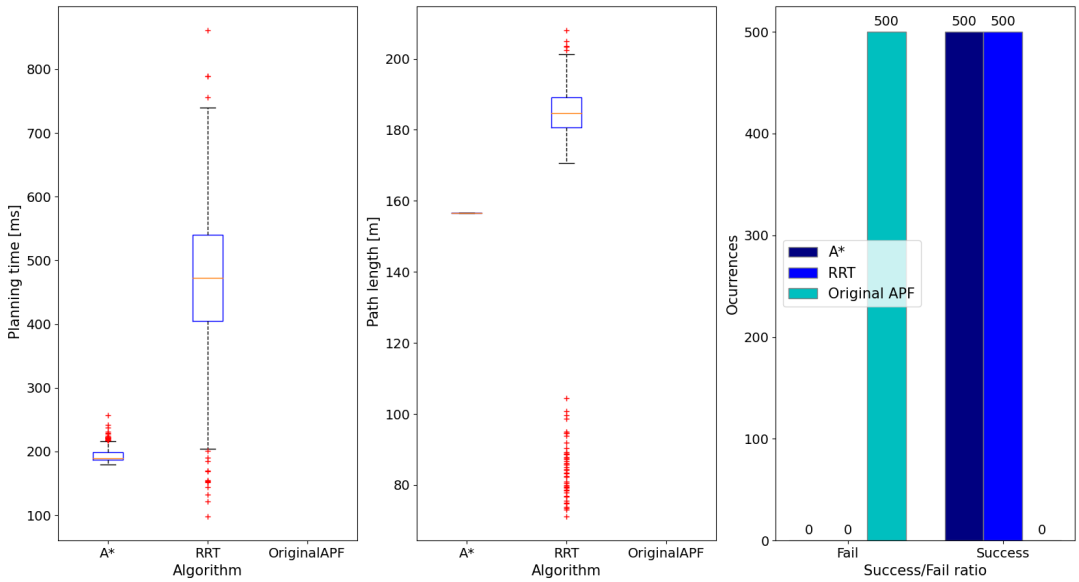


Figure B.8: Algorithm performance with object radius of 8 metres for 500 run

B.1.3. View distance adjustment results

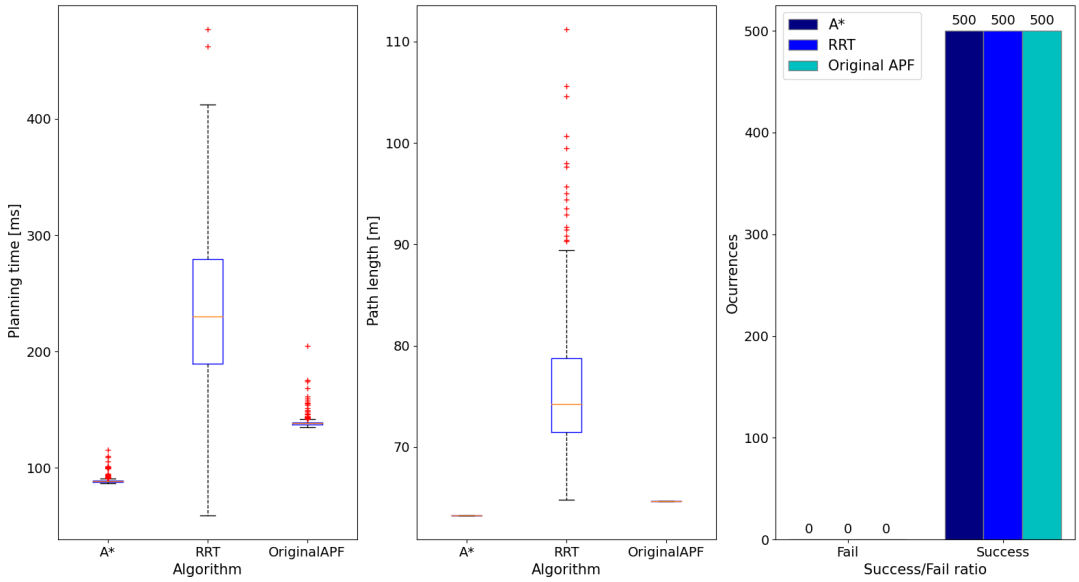


Figure B.9: Algorithm performance with view distance of 3 metres(standard conditions) for 500 runs

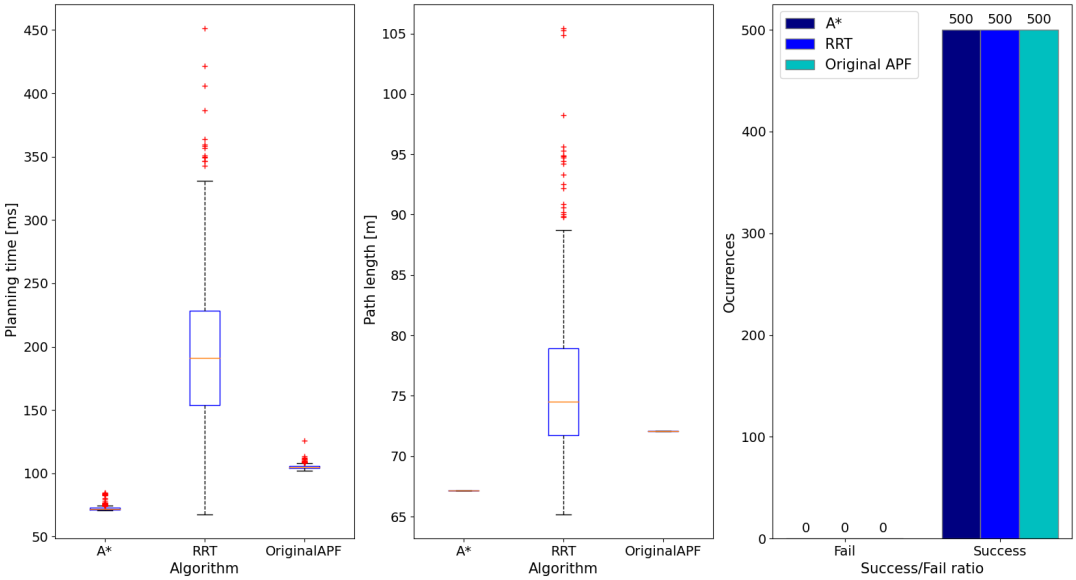


Figure B.10: Algorithm performance with view distance of 1 metres for 500 runs

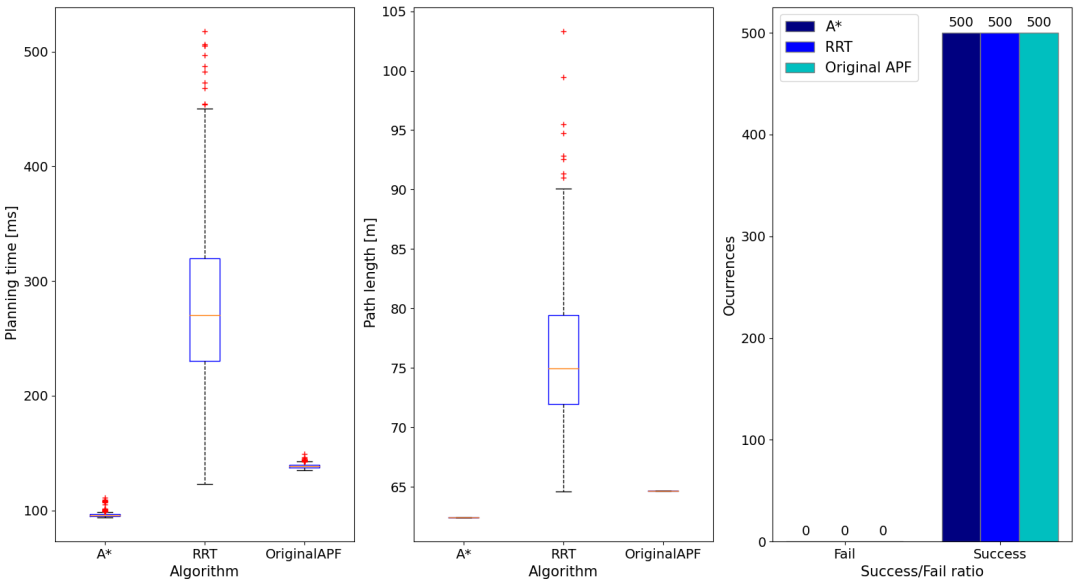


Figure B.11: Algorithm performance with view distance of 5 metres for 500 runs

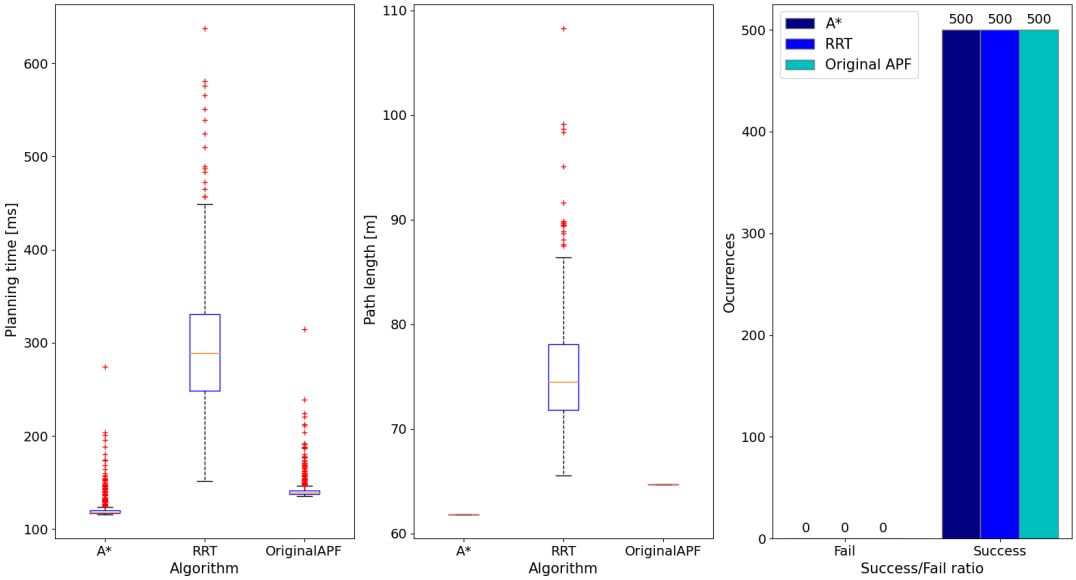


Figure B.12: Algorithm performance with view distance of 8 metres for 500 runs

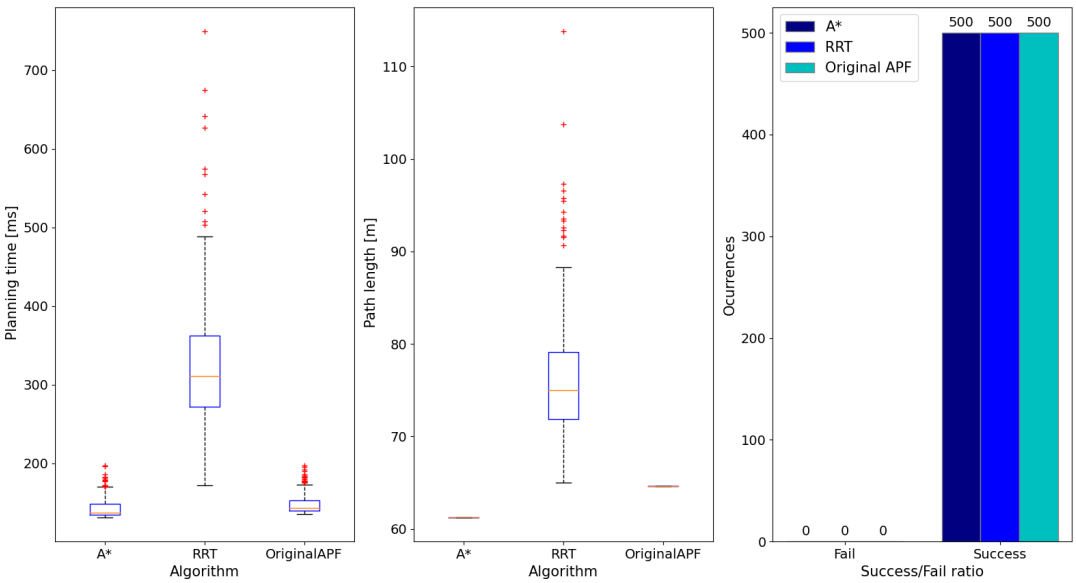


Figure B.13: Algorithm performance with view distance of 10 metres for 500 runs

B.1.4. Obstacle scenario adjustment results

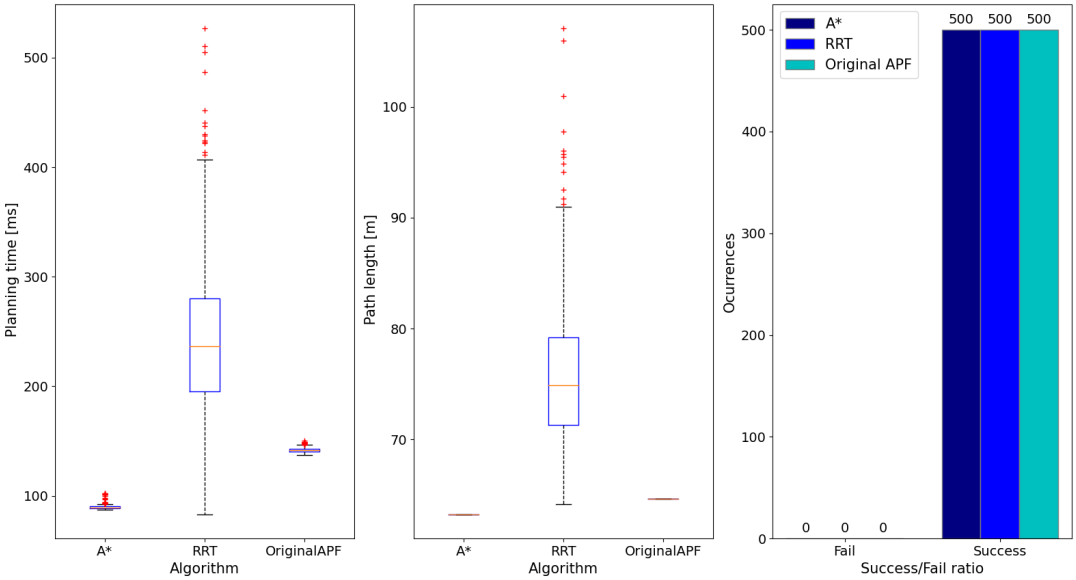


Figure B.14: Algorithm performance with standard obstacle scenario for 500 runs

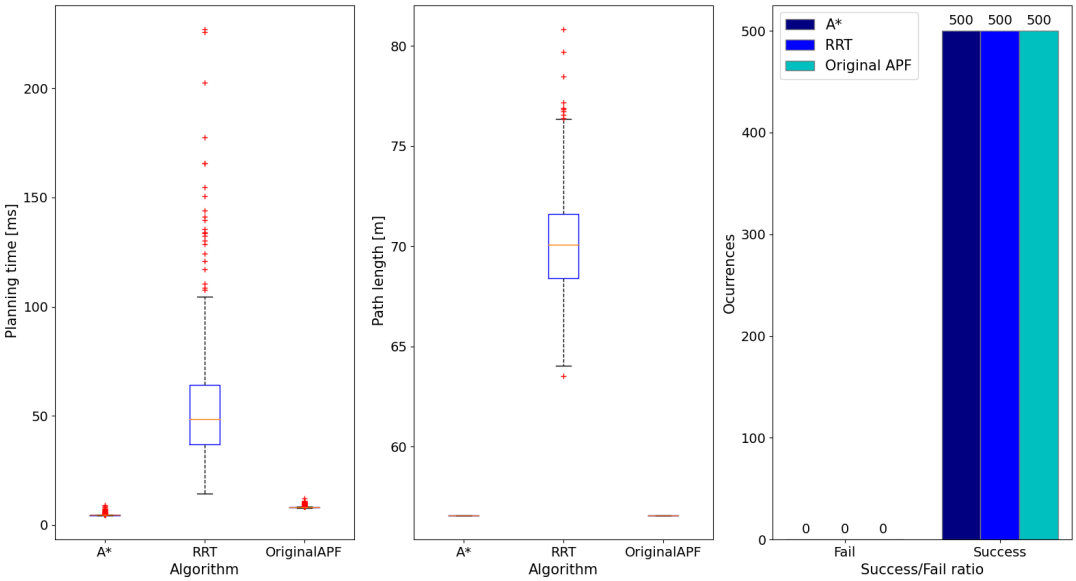


Figure B.15: Algorithm performance with zero obstacles for 500 runs

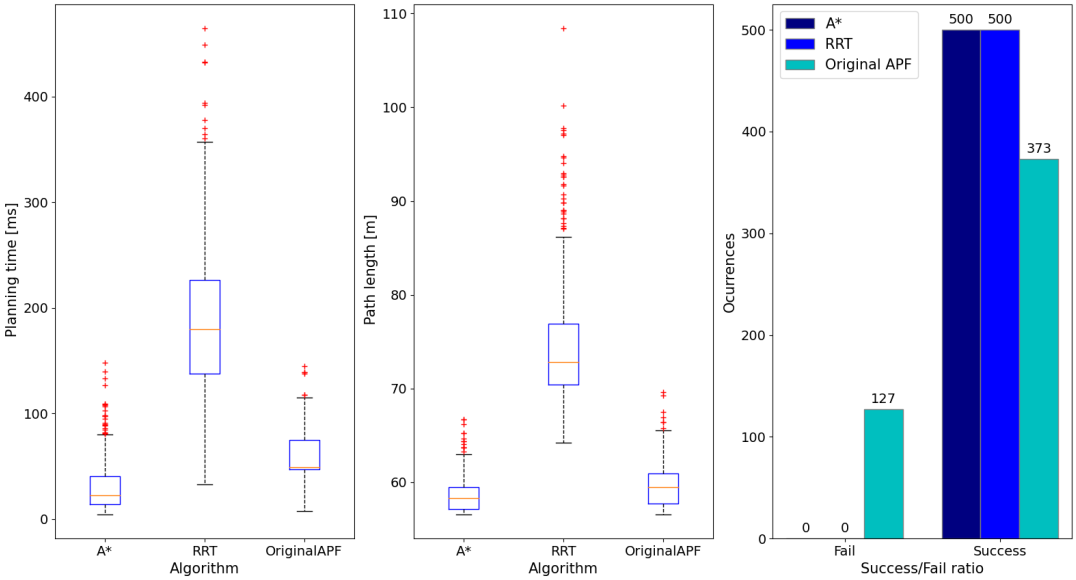


Figure B.16: Algorithm performance with 5 random obstacles(radius 3 meter) for 500 runs

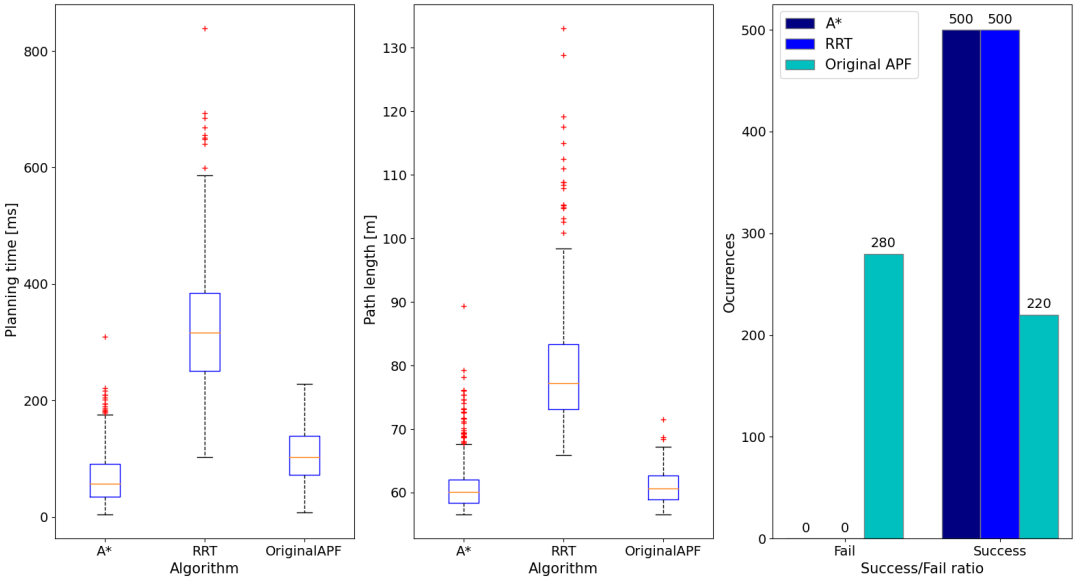


Figure B.17: Algorithm performance with 10 random obstacles(radius 3 meter) for 500 runs

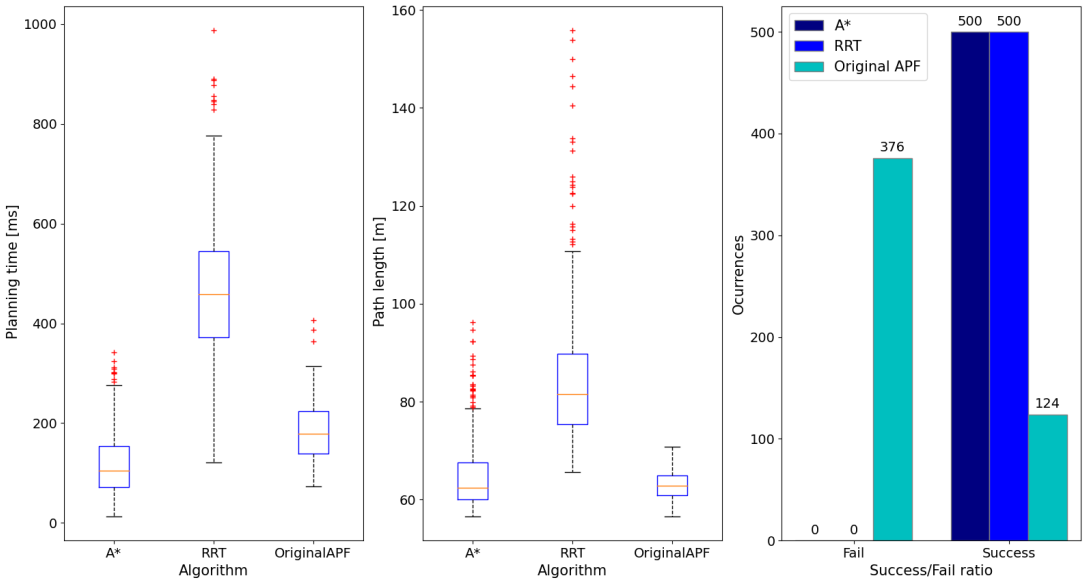


Figure B.18: Algorithm performance with 15 random obstacles(radius 3 meter) for 500 runs

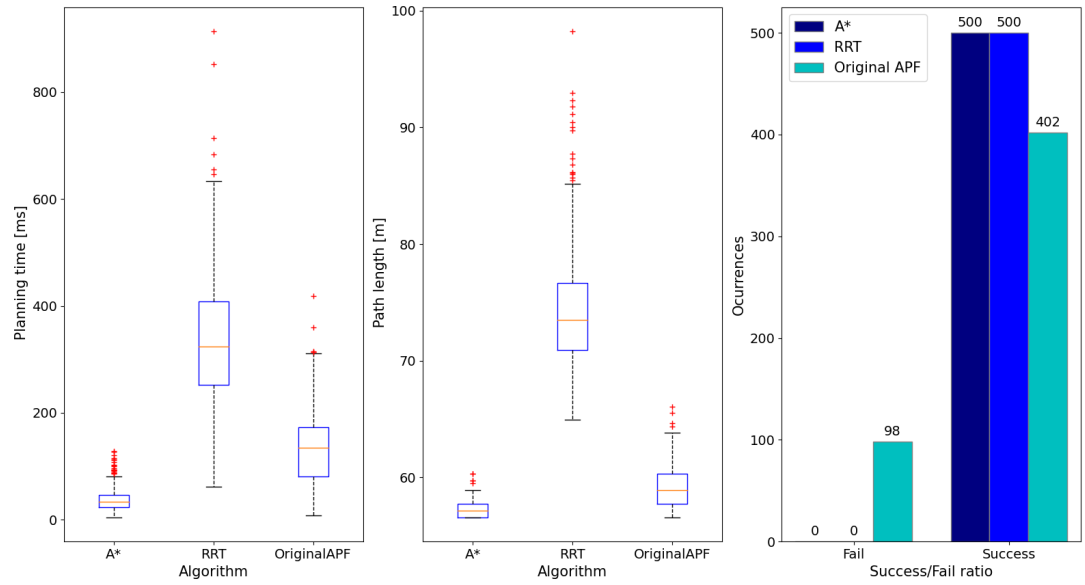


Figure B.19: Algorithm performance with 15 random obstacles(radius 1 meter) for 500 runs

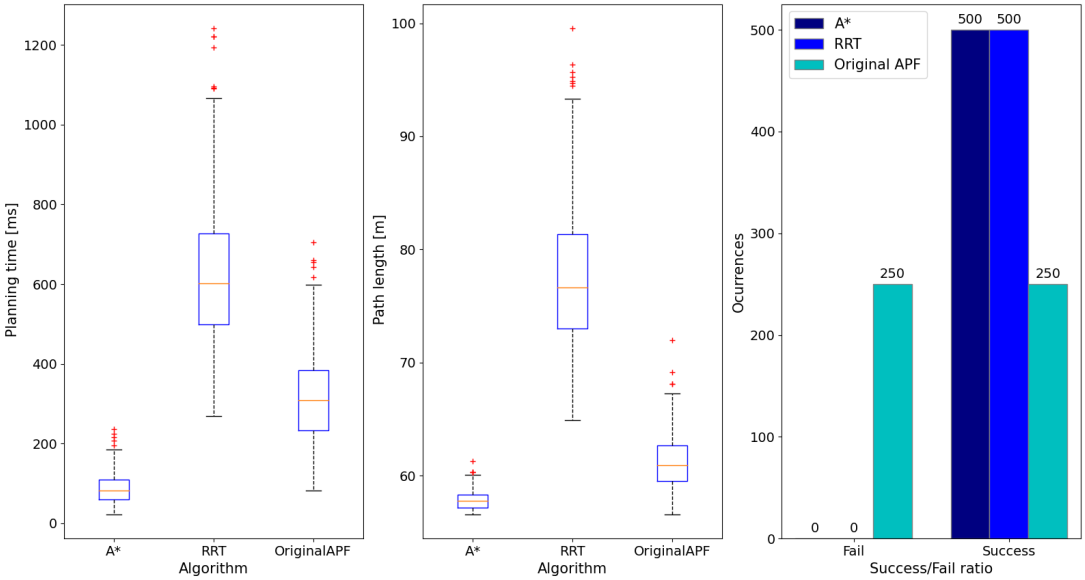


Figure B.20: Algorithm performance with 30 random obstacles(radius 1 meter) for 500 runs

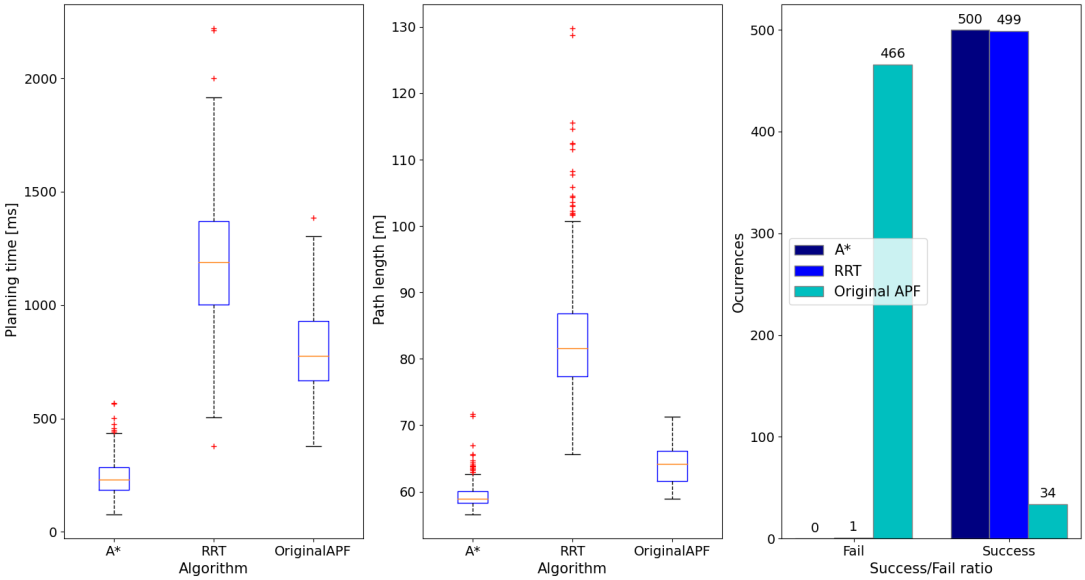


Figure B.21: Algorithm performance with 60 random obstacles(radius 1 meter) for 500 runs

C

Miscellaneous

C.1. Python package requirements

Listing C.1: Python environment packages with versions

```
matplotlib==3.5.2
numpy==1.22.3
scipy==1.8.1
```

C.2. Full rotational vector field algorithm

This section describe the complete Rotation vector field algorithm. The pseudo code is given in Alg.11. Fig.C.1 shows corresponding flowchart of the complete rotational vector field algorithm.

Algorithm 11 Rotational Vector Field Algorithm

Require: $VFN_{Start}, VFN_{Goal}, ObstacleList, RoverRadius, GridSize, AreaBounds, PotentialMap$
Ensure: $Collision\ Free\ Path$

$Path \leftarrow [VFN_{start}]$
$PreviousNodes \leftarrow deque()$

$DetermineObstacleRotation(VFN_{Start}, VFN_{Goal}, ObstacleList)$

for $X, Y \in Area$ **do**
    if $FirstMapCalculation$ **then**
        $PositivePotential \leftarrow CalculateAttractivePotential(X, Y, VFN_{Goal})$
    end if
    $NegativePotential \leftarrow CalculateRepulsivePotential(X, Y, ObstacleList)$
    $PotentialMap \leftarrow TotalPotential(X, Y, PostivePotential, NegativePotential)$
end for

$Distance \leftarrow CalculateDistance(VFN_{Goal}, VFN_{Start})$
$VFN_{current} \leftarrow VFN_{Start}$
while $NoLocalMinimum$ **do**
    $PointVector \leftarrow GetVectorFromNode(VFN_{current})$
    $PointAngle \leftarrow GetAngleOfVector(PointVector)$
    $VFN_{next} \leftarrow DetermineNextPoint(PointAngle)$
    $VFN_{current} = VFN_{next}$
    $Path \leftarrow AddNodeToList(VFN_{current}, Path))$
    $Distance \leftarrow CalculateDistance(VFN_{Goal}, VFN_{current})$
    if $Distance < GridSize$ **then**
        for $PathPoints\ in\ Path$ **do**
            $DistanceBetweenNodes \leftarrow CalculateDistacne(VFN_{PathPoint}, VFN_{PathPoint+2})$
            if $DistanceBetweenNodes <= 2 * GridSize$ **then**
                $New_X \leftarrow (VFN_{PathPoint+2,X} - VFN_{PathPoint,X})/2 + VFN_{PathPoint,X}$
                $New_Y \leftarrow (VFN_{PathPoint+2,Y} - VFN_{PathPoint,Y})/2 + VFN_{PathPoint,Y}$
                $VFN_{PathPoint+1} \leftarrow [New_X, New_Y]$
            end if
        end for
        return $Path$
    end if
end while
return $None$

$NoLocalMinimum(VFN_{current})$:
    $PreviousNodes \leftarrow AddNodeToList(VFN_{current}, PreviousNodes))$
    if $length(PreviousNodes) > 3$ **then**
        $RemoveFirstItemFromList(PreviousNodes)$
    end if
    if $NodeInListIsDouble(PreviousNodes)$ **then**
        return $True$
    else
        return $False$
    end if

```
47:
48: DetermineObstacleRotation(VFN_Start, VFN_Goal, ObstacleList) :
49:     RoverGoalVector ← CalculateVector(VFN_Goal, VFN_Start)
50:     for NewObstacles do
51:         ObstacleDistance ← CalculateDistance(Obstacle_Centre(x,y), VFN_current)
52:         RoverObstacleVector ← CalculateVector(Obstacle_Centre(x,y), VFN_Start)
53:         GoalObstacleAngle ← CalculateAngle(RoverGoalVector, RoverObstacleVector)
54:         if GoalObstacleAngle >= 0 then
55:             Obstacle_CurlDirection = 1
56:         else
57:             Obstacle_CurlDirection = −1
58:         end if
59:         for DetectedObstacles do
60:             FreeSpace ← CalculateDistanceBetweenObstacles_DetectedObstacle,NewObstacle
61:             if FreeSpace ≤ 3 ∗ RoverRadius then
62:                 NewObstacle_rotation = DetectedObstacle_rotation
63:             end if
64:         end for
65:         for DetectedObstacles do
66:             FreeSpace ← CalculateDistanceBetweenObstacles_DetectedObstacle,NewObstacle
67:             if FreeSpace ≤ 3 ∗ RoverRadius then
68:                 NewObstacle_rotation = DetectedObstacle_rotation
69:             end if
70:         end for
71:     end for
72:
73: CalculateRepulsivePotential(X, Y, VFN_Goal) :
74:     GridDiagonal = √(GridSize² + GridSize²)
75:     RepulsiveVector = [0, 0]
76:     for NewObstacles do
77:         if ObstacleDistance <= (RoverRadius + Obstacle_Radius + 2 ∗ GridDiagonal) then
78:             if X, Y == Obstacle_Centre(x,y) then
79:                 RepulsiveVector = [0, 0]
80:             else
81:                 RepulsiveVector_X = Obstacle_CurlDirection ∗ (Y − Obstacle_Centre(y))
82:                 RepulsiveVector_Y = −Obstacle_CurlDirection ∗ (X − Obstacle_Centre(x))
83:                 RepulsiveVector_X = RepulsiveVector_X / VectorLength
84:                 RepulsiveVector_Y = RepulsiveVector_Y / VectorLength
85:             end if
86:             if ObstacleDistance <= (RoverRadius + Obstacle_Radius + GridDiagonal) then
87:                 RepulsiveVector_X = RepulsiveVector_X + (X − Obstacle_x) / VectorLength
88:                 RepulsiveVector_Y = RepulsiveVector_Y + (Y − Obstacle_y) / VectorLength
89:             end if
90:             RepulsiveVector+ = [RepulsiveVector_X, RepulsiveVector_Y] ∗ RepulsiveGain
91:         end if
92:     end for
93:     return RepulsiveVector
94:
95: CalculateAttractivePotential(X, Y, VFN_Goal) :
96:     AttractiveVector_X = VFN_Goal,X − X
97:     AttractiveVector_Y = VFN_Goal,Y − Y
98:     AttractiveVector = [AttractiveVector_X / VectorLength, AttractiveVector_Y / VectorLength]
99:     return AttractiveVector ∗ AttractiveGain
```

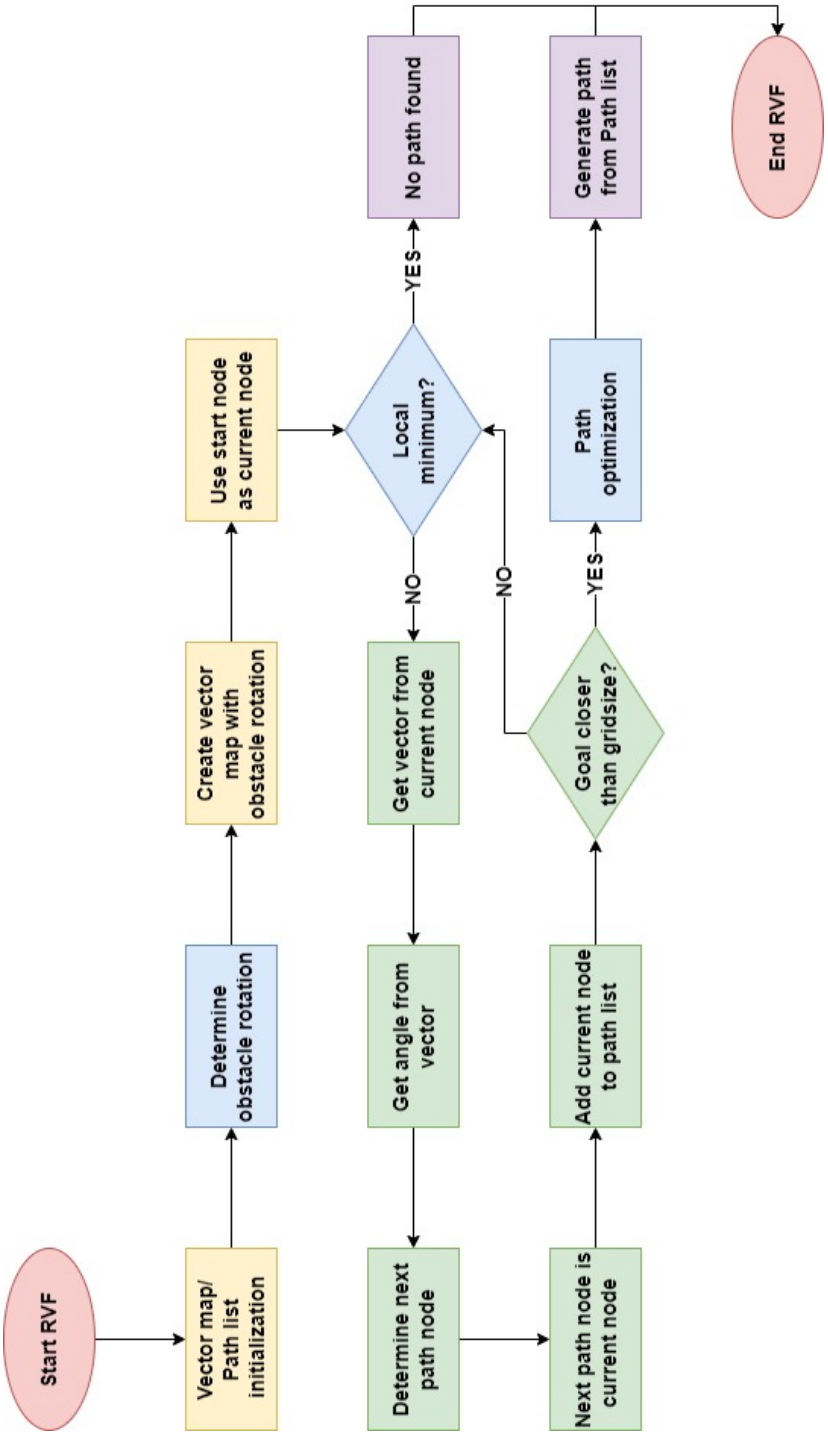


Figure C.1: Flowchart for Rotational Vector Field algorithm. The initialization is denoted in yellow, path planning in green, use of external functions in blue and the outputs in purple.

Bibliography

[1] Gianpietro Battocletti et al. “RL-based Path Planning for Autonomous Aerial Vehicles in Unknown Environments”. In: *AIAA AVIATION 2021 FORUM*. 2021, p. 3016.

[2] Sander Betz. *Locomotion Simulation of The Lunar Zebro Moon Rover Drive Train*. 2022.

[3] Said Broumi et al. “Shortest path problem in fuzzy, intuitionistic fuzzy and neutrosophic environment: an overview”. In: *Complex & Intelligent Systems* 5.4 (2019), pp. 371–378.

[4] Chunxi Cheng et al. “Path planning and obstacle avoidance for AUV: A review”. In: *Ocean Engineering* 235 (2021), p. 109355.

[5] Daegyun Choi, Kyuman Lee, and Donghoon Kim. “Enhanced potential field-based collision avoidance for unmanned aerial vehicles in a dynamic environment”. In: *AIAA Scitech 2020 Forum*. 2020, p. 0487.

[6] Howie Choset and Philippe Pignon. “Coverage path planning: The boustrophedon cellular decomposition”. In: *Field and service robotics*. Springer. 1998, pp. 203–209.

[7] Ian A Crawford, Katherine H Joy, and Mahesh Anand. “Lunar exploration”. In: *Encyclopedia of the Solar System*. Elsevier, 2014, pp. 555–579.

[8] Xiao Cui and Hao Shi. “A*-based pathfinding in modern computer games”. In: *International Journal of Computer Science and Network Security* 11.1 (2011), pp. 125–130.

[9] Mosab Diab, Mostafa Mohammadkarimi, and Raj Thilak Rajan. “Artificial Potential Field-Based Path Planning for Cluttered Environments”. In: *Accepted in IEEE Aerospace conference*. Accepted. Mar. 2023.

[10] Xiaojing Fan et al. “Improved artificial potential field method applied for AUV path planning”. In: *Mathematical Problems in Engineering* 2020 (2020).

[11] Iswanto Iswanto, Oyas Wahyunggoro, and Adha Imam Cahyadi. “Path planning based on fuzzy decision trees and potential field”. In: *International Journal of Electrical and Computer Engineering* 6.1 (2016), p. 212.

[12] Lucas Janson, Brian Ichter, and Marco Pavone. “Deterministic sampling-based motion planning: Optimality, complexity, and performance”. In: *The International Journal of Robotics Research* 37.1 (2018), pp. 46–61.

[13] Oussama Khatib. “Real-time obstacle avoidance for manipulators and mobile robots”. In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. IEEE. 1985, pp. 500–505.

[14] Jinseok Lee, Yunyoung Nam, and Sangjin Hong. “Random force based algorithm for local minima escape of potential field method”. In: *2010 11th International Conference on Control Automation Robotics & Vision*. IEEE. 2010, pp. 827–832.

[15] Sang-Min Lee, Kyung-Yub Kwon, and Joh Joongseon. “A fuzzy logic for autonomous navigation of marine vehicles satisfying COLREG guidelines”. In: *International Journal of Control, Automation, and Systems* 2.2 (2004), pp. 171–181.

[16] Patrick Lester. “A* pathfinding for beginners”. In: *online]. GameDev WebSite. http://www. gamedev. net/reference/articles/article2003. asp (Acesso em 08/02/2009)* (2005).

[17] Daoliang Li, Peng Wang, and Ling Du. “Path planning technologies for autonomous underwater vehicles-a review”. In: *Ieee Access* 7 (2018), pp. 9745–9768.

[18] Guanghui Li et al. “An efficient improved artificial potential field based regression search method for robot path planning”. In: *2012 IEEE International Conference on Mechatronics and Automation*. IEEE. 2012, pp. 1227–1232.

[19] Tapovan Lolla et al. “Path planning in time dependent flow fields using level set methods”. In: *2012 IEEE International Conference on Robotics and Automation*. IEEE. 2012, pp. 166–173.

[20] Mohamed Slim Masmoudi et al. “Fuzzy logic controllers design for omnidirectional mobile robot navigation”. In: *Applied soft computing* 49 (2016), pp. 901–919.

[21] Ivan Maurović et al. “Path planning for active SLAM based on the D* algorithm with negative edge weights”. In: *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 48.8 (2017), pp. 1321–1331.

[22] Floor Thomas Melman et al. “LCNS Positioning of a Lunar Surface Rover Using a DEM-Based Altitude Constraint”. In: *Remote Sensing* 14.16 (2022), p. 3942.

[23] J.B. Miog. *DESIGN OF THE LOCOMOTION SUBSYSTEM FOR LUNAR ZEBRO*. 2018.

[24] Stanley Osher, Ronald Fedkiw, and K Piechor. “Level set methods and dynamic implicit surfaces”. In: *Appl. Mech. Rev.* 57.3 (2004), B15–B15.

[25] Min Gyu Park and Min Cheol Lee. “A new technique to escape local minimum in artificial potential field based path planning”. In: *KSME international journal* 17.12 (2003), pp. 1876–1885.

[26] Clement Petres et al. “Underwater path planing using fast marching algorithms”. In: *Europe Oceans 2005*. Vol. 2. IEEE. 2005, pp. 814–819.

[27] Vishnu Kumar Prajapati, Mayank Jain, and Lokesh Chouhan. “Tabu search algorithm (TSA): A comprehensive survey”. In: *2020 3rd International Conference on Emerging Technologies in Computer Engineering: Machine Learning and Internet of Things (ICETCE)*. IEEE. 2020, pp. 1–8.

[28] Seyyed Mohammad Hosseini Rostami et al. “Obstacle avoidance of mobile robots using modified artificial potential field algorithm”. In: *EURASIP Journal on Wireless Communications and Networking* 2019.1 (2019), pp. 1–19.

[29] Stijn Rovers. *OPAL, A Stereo Vision Obstacle Processing ALgorithm for a Walking Lunar Rover*. 2021.

[30] Benjamn Tovar, Rafael Murrieta-Cid, and Steven M LaValle. “Distance-optimal navigation in an unknown environment without sensing distances”. In: *IEEE Transactions on Robotics* 23.3 (2007), pp. 506–518.

[31] Thomas T Trexler and Robert B Odden. “Lunar Surface Navigation”. In: *IEEE Transactions on Aerospace and Electronic Systems* 6 (1966), pp. 252–259.

[32] Hongjian Wang et al. “A vector polar histogram method based obstacle avoidance planning for AUV”. In: *2013 MTS/IEEE OCEANS-Bergen*. IEEE. 2013, pp. 1–5.

[33] Xinda Wang et al. “Collision-free path planning method for robots based on an improved rapidly-exploring random tree algorithm”. In: *Applied Sciences* 10.4 (2020), p. 1381.

[34] Shuhuan Wen et al. “Path planning for active SLAM based on deep reinforcement learning under unknown environments”. In: *Intelligent Service Robotics* 13.2 (2020), pp. 263–272.

[35] Jing Xin et al. “Application of deep reinforcement learning in mobile robot path planning”. In: *2017 Chinese Automation Congress (CAC)*. IEEE. 2017, pp. 7112–7116.

[36] Qingfeng Yao et al. “Path planning method with improved artificial potential field—a reinforcement learning perspective”. In: *IEEE Access* 8 (2020), pp. 135513–135523.

[37] Xiaoping Yun and Ko-Cheng Tan. “A wall-following method for escaping local minima in potential field based motion planning”. In: *1997 8th International Conference on Advanced Robotics. Proceedings. ICAR’97*. IEEE. 1997, pp. 421–426.

[38] Mohd Nayab Zafar and JC Mohanta. “Methodology for path planning and optimization of mobile robots: A review”. In: *Procedia computer science* 133 (2018), pp. 141–152.

[39] Zheng Zeng et al. “Imperialist competitive algorithm for AUV path planning in a variable ocean”. In: *Applied artificial intelligence* 29.4 (2015), pp. 402–420.

[40] Han-ye Zhang, Wei-ming Lin, and Ai-xia Chen. “Path planning for the mobile robot: A review”. In: *Symmetry* 10.10 (2018), p. 450.

[41] Xuexi Zhang et al. “2D LiDAR-based SLAM and path planning for indoor rescue using mobile robots”. In: *Journal of Advanced Transportation* 2020 (2020).

[42] Yalong Zhang, Zhenghua Liu, and Le Chang. “A new adaptive artificial potential field and rolling window method for mobile robot path planning”. In: *2017 29th Chinese Control And Decision Conference (CCDC)*. IEEE. 2017, pp. 7144–7148.

[43] Qidan Zhu, Yongjie Yan, and Zhuoyi Xing. “Robot path planning based on artificial potential field approach with simulated annealing”. In: *Sixth International Conference on Intelligent Systems Design and Applications*. Vol. 2. IEEE. 2006, pp. 622–627.