University of Manchester

School of Computer Science

Project Report 2020

Counterexample-Guided Optimisation Applied to Mobile Robot Path Planning

Author: Laura Mihai

Supervisor: Dr Lucas Cordeiro

Abstract

This report offers an introduction to *Model Checking* as an automated verification technique and discusses the use of the *Counterexample Guided Inductive Optimisation* algorithm as a method of approaching *off-line path planning*.

It further presents a possible implementation of *trajectory planning* using *ROBOTC* as an Integrated Development Environment for programming a *Lego Mindstorms EV3* robot.

Acknowledgements

I wish to express my deepest appreciation to my supervisor, Dr Lucas Cordeiro, for his incredible encouragement, patience, and constant belief in my abilities to finish this project.

I would also like to thank my family and friends who have supported me along my studies.

CONTENTS

CHAPTER 1 – INTRODUCTION

As society makes its shift towards a more automated future, the practice of using industrial and service robots to help or replace human workers is getting more and more common. One of the core requirements for robots such as the Tesla Self-Driving Car and iRobot Roomba Series is the ability to decide their next step, ideally without any human intervention. For this particular reason, the autonomous navigation field is experiencing a steady increase in demand, with the global market estimated to reach $6.15 Billion by 2025 [1].

Robot navigation takes into account three main problems: self-localisation, path planning and map-building. In its most basic form, path planning can be described as the problem of finding a sequence of positions which allows the robot to move continuously and safely from one point to another. Numerous approaches to this problem can be found in literature, ranging from the pivotal node-based algorithms to sampling-based, mathematical model-based, bioinspired and multifusion-based methods [2].

This report is concerned only with the path planning problem and the resulting trajectory planning, since the method used to generate the solution can be applied only in off-line mode planning (i.e., planning is done before movement). It is based on the work of Araújo *et al.* in which a counterexample guided inductive optimisation (CEGIO) algorithm is used to obtain optimal two-dimensional paths for autonomous mobile robots [3].

The path planning task can be modelled as an optimisation problem, with the path encoded as a decision variable and the cost function representing a given criteria or metric whose value is to be optimized (e.g., distance, energy consumption, and execution time) [4]. Consequently, it can be solved using various optimisation techniques, such as genetic algorithm (GA), particle swarm optimisation (PSO), nonlinear programming (NLP) and ant colony [5]. These techniques have the advantage of fast processing time, but global optimality is not ensured.

CEGIO employs non-deterministic representation of decision variables and performs iterative executions of successive verifications based on counterexamples produced by a Boolean Satisfiability (SAT) or Satisfiability Modulo Theories (SMT) solver in order to achieve complete global optimization without employing randomness [6]. It has been shown in previous studies that CEGIO is capable of solving a wide variety of functions, even non-linear and non-convex ones [7]. Compared to the other optimisation techniques mentioned above, which are usually trapped by local minima, CEGIO always finds the correct global minima [8].

1.1 OBJECTIVES AND CONTRIBUTION

The objective of this project is to successfully implement the method described in the original paper and then use the results for the practical application of getting a robot to navigate accordingly. This implies that the agent must have prior knowledge of the starting and target coordinates, environmental limits and obstacle data, as well as the path it must follow. Therefore, the main original contribution of this paper is the trajectory planning done for a Lego Mindstorms EV3 robot.

1.2 IMPACT OF COVID-19

The practical work for this report was completed before the outbreak of COVID-19.Therefore, there were no major changes or adjustments made in order to complete this project.

CHAPTER 2 – PATH PLANNING PRELIMINARIES

2.1 OPTIMISATION PROBLEMS OVERVIEW

Optimisation is indisputably a core part of computer science, as many important advances are based on optimisation theory, such as planning and decidability problems, resource allocation problems and computational estimation and approximation [9].

Rao defines optimisation as the process of finding the parameters that give the minimum value of a function, as the maximum of a function can be found by seeking the minimum of the negative of the same function [10]. Therefore, the following definition is presented:

Let $f : X \to \mathbb{R}$ be a cost function, such that $X \subset \mathbb{R}^n$ represents the decision variables vector $x_1, x_2, \ldots, x_n$ and $f(x_1, x_2, \ldots, x_n) \equiv f(\mathbf{x})$. Let $\Omega \subset X$ be a subset settled by a set of constraints.

Definition 1. *A multi-variable optimization problem consists in finding an optimal vector* **x,** *which minimizes f in* **Ω** [11].

Resulting from this definition is the following formulation of an optimization problem:

$$\begin{aligned} \min \quad & f(\boldsymbol{x}), \\ s.t. \quad & \boldsymbol{x} \in \Omega. \end{aligned} \qquad \textbf{(1)}$$

A classification can be done based on the nature of the cost function f, decision variables domain and constraints. The domain X and constraints heavily influence the size of the optimization search-space, thus influencing the performance of the optimization algorithms [12]. Depending on the type of the cost function f (e.g., linear or non-linear, continuous, discontinuous or discrete, convex or non-convex, and single- or multi-objective) specific optimisation techniques are employed.

Optimisation problems with a non-convex cost function are particularly hard to solve, the difficulty arising from the various inflexion points that can trap the employed algorithm to a sub-optimal solution i.e., a solution that is not a global minimum of *f*, but only locally minimizes *f*. Note that the problem is necessarily non-linear in this case, and it can furthermore also be discontinuous [13].

A definition for a global optimal solution for an optimisation problem is presented as follows:

Definition 2. *A vector $x^* \in \Omega$ is a global optimal solution of f in Ω iff $f(x^*) \leq f(x), \forall x \in \Omega$* [14].

The problem of finding the global optimal solution is a particularly hard one to solve, research area in this field being still on its way to maturity [15].

2.2 OPTIMIZATION TECHNIQUES

The methods developed for solving the optimization problems can be divided into two groups, depending on whether they make use of stochastic elements. Deterministic methods are those that do not involve any stochastic concepts and employ a search engine, where each step is directly and deterministically related to the previous steps [16]. On the other hand, stochastic methods employ randomness to avoid the local minima and are often based on meta-heuristics [17].

Deterministic approaches can be classified as gradient-based or enumerative search-based. Gradient-based techniques, such as gradient-descent and Newton's optimization, search for points, where the gradient of the cost function is null ($\nabla f(x) = 0$). They have the advantage of producing fast result, however they are unusable for non-convex or non-differentiable problems. Enumerative search-based optimization algorithms (e.g., dynamic programming, branch and bound, pattern search) scan the search-space by taking into account all possible points and compare the value of the cost function with best previous values [18].

As of latest, SMT-based optimization techniques are being used to solve optimization problems. Those techniques employ non-deterministic representation of decision variables and lead to the global optima by using counterexamples produced by SMT solvers, which constrain a search-space that is symbolically defined [19]. The global optima is the set of values for the decision variables that makes an optimization proposition satisfiable [20].

2.3 MODEL CHECKING

Model checking is the most widely used technique for automatic formal verification of finite state transition systems. It consists of modelling the desired design as a finite state machine, with the specification formalised by writing temporal logic properties. The first decade of research on this topic was focused on deciding which specification formalism is more appropriate: linear time logic (LTL) or branching time logic, i.e computational tree logic (CTL) [21].

Research concludes that BDD-based checking is biased towards CTL and SAT-based checking has a bias towards LTL [22]. Furthermore, BDD-based systems can check models with no more than a few hundred latches, whereas SAT-based can handle instances with hundreds of thousands of variables and millions of clauses [23].

In model checking all reachable states of the system are traversed in order to verify 'safety' (what should not happen) or 'liveness' (what should eventually happen) properties. In the case of a failed property, a counterexample is generated in the form of a sequence of states which led to the particular failure [24].

This technique usually implies three separate steps. In the first step, called modelling, the system is converted to a formalism accepted by a verifier. The second step is the specification, which describes the system's behaviour and the property to be checked [25].

Model checking provides ways to check whether a given specification satisfies a system property, but it is difficult to determine whether such specification covers all properties which the system should satisfy [26].

Finally, the verification step checks whether a given property is satisfied with respect to a given model, i.e., all relevant system states are checked to search for any state that violates the verified property. In case of a property violation, the verifier reports the system execution trace (counterexample), which contains all steps from the initial state to the property violating state [27].

2.3.1 BOUNDED MODEL CHECKING (BMC):

Bound Model Checking originated as an attempt to replace the use of Binary Decision Diagrams (BDD) in symbolic model checking with SAT. BMC is based on the idea of traversing a finite prefix of states, restricted by some bound *k*, such that there may be a trace that contradicts the required property. An important remark is that the existence of a back loop from the last state of the prefix to any of the previous states leads to an infinite path [28].

BMC involves the translation of the k-bounded model into a SAT or SMT instance, which is polynomial in the original planning problem and the bound. Compared to deterministic planning, which is only concerned with simple safety properties (i.e, whether and how the goal state can be reached), model checking also takes liveness and nested temporal properties into consideration [29]. In summary, BMC checks the satisfiability of a verification condition, which is the result of translating the problem into a formalism accepted by the verifier.

Definition 4. *A set of formulas $\{p_1, p_2, \dots, p_n\}$ is said to be satisfiable if there is some structure A in which all its component formulas are true, i.e., $\{p_1, p_2, \dots, p_n\}$ is SAT iff $A \vDash p_1 \wedge A \vDash p_2 \dots A \vDash p_n$* [30].

Definition 5. *Given a transition system M, a property ϕ, and a bound k; BMC unrolls the system k times and translates it into a verification condition (VC) ψ, which is satisfiable iff ϕ has a counterexample of depth less than or equal to k* [31].

In BMC, the associated problem is formulated by constructing the following logical formula:

$$\psi_k = I(S_0) \wedge \vee_{i=0}^{k} \wedge_{j=0}^{i-1} (\gamma(s_j, s_{j+1}) \wedge \neg\ \phi(s_i)) \qquad \textbf{(2)}$$

where φ is a property and S_0 is a set of initial states of *M*, and $\gamma(s_j, s_{j+1})$ is the transition relation of M between time steps j and j + 1 [32].

The above condition ψ_k can be satisfied if and only if, for some i ≤ k there exists a reachable state at time step *i* in which ϕ is violated. If the logical formula is satisfiable (i.e., returns true), then the SMT solver provides a satisfying assignment (counterexample) [33].

Definition 6. *A counterexample for a property ϕ is a sequence of states $s_0, s_1, \dots, s_k$ with $s_0 \in S_0$, $s_k \in S_k$ and $\gamma(s_j, s_{j+1})$ for 0 ≤ i ≤ k that makes ψ_k satisfiable. If it is unsatisfiable (i.e., returns false), then we can conclude that there is no error state in k steps or less* [34].

2.4 MODELLING OPTIMIZATION PROBLEMS USING A SOFTWARE MODEL CHECKER

As mentioned in Section 2.4, model checking involves three separate steps: modelling, specification and verification. In order to model and control the verification process, two code directives in the C/C++ programming language may be used to check desired constraints: ASSUME and ASSERT. The ASSUME directive can define constraints over (non-deterministic) variables, and the ASSERT directive is used to check system's correctness regarding a given property [35].

Any off-the-shelf C/C++ model checker that includes those two intrinsic functions can be used as a verification engine (e.g. ESBMC, CBMC, CPAChecker). In this report, ESBMC is used since it is one of the most efficient BMC tools [36].

In the presented CEGIO-based algorithms, verification is done iteratively in order to lead to the global optimal result. The basic idea is to incrementally use ESBMC in order to verify the satisfiability of a desired property and update the value of the candidate to the cost function based on the provided counterexample.

2.5 COUNTEREXAMPLE GUIDED INDUCTIVE OPTIMIZATION (CEGIO)

The CEGIO technique is based on iterative executions to constrain a verification procedure, in order to perform inductive generalization, based on counterexamples extracted from SAT and SMT solvers [37]. It employs non-deterministic representation of decision variables.

CEGIO has been proven to successfully optimize a wide variety of functions, given a desired precision for the respective decision variables, including non-linear and non-convex optimization problems. Based on SAT and SMT solvers, data provided by counterexamples is employed to guide the verification engine, thus reducing the optimization domain [38].

Araújo *et al.* proposed three different versions of CEGIO, each suited for a particular class of problems. The Generalized Algorithm (CEGIO-G) can be applied to both convex and non-convex functions, the Simplified Algorithm (CEGIO-S) is best suited for problems where there is some prior data given about the function(e.g. semi- and positive-definite functions), and the Fast Algorithm (CEGIO-F) which is applied only to convex functions.

The main advantage of CEGIO compared to the other discussed path planning algorithms is that it will always find the global optimal point, though it is more time-costly depending on the problem.

Since the cost function in path planning is convex and always positive, the proposed method is based on CEGIO-F.

2.5.1 CEGIO – F

The CEGIO-F algorithm requires three inputs: a cost function $f(x)$; the space for constraint set Ω; and a desired precision ϵ. The output consists of the decision variables vector $x^*$and minimum optimal value of the cost function $f(x^*)$.

The algorithm begins with variable declaration and initialization (lines 1-2 of **Algorithm 1**). $f_c^{(0)}$ represents a given candidate for the minimum value of the cost function. The auxiliary variables X are declared as non-deterministic integer variables in order to minimise the state-space search.

The variable ϵ reflects the number of decimal places of the decision variables values, $\boldsymbol{x}$. If $\epsilon = 0$, then obtained solution is included in $\mathbb{Z}$. As seen in line 3, ϵ starts off with the 0 value and is incremented after each iteration of the outer loop, in order to increase the decision variables domain by one decimal place in the next iteration of the loop [39].

The search domain Ω^ϵ is defined in line 4 with the help of the upper and lower bounds of the auxiliary variables X. Thus, Ω^ϵ limits are given by:

$$\lim\{\Omega^\epsilon\} = \lim\{\Omega\} \times 10^{\epsilon} \qquad \textbf{(3)}$$

Then the model for the objective function $f(x)$ is defined in line 5, with consideration to the desired precision, *i.e.*, $\boldsymbol{x} = \boldsymbol{X}/10^{\epsilon}$.

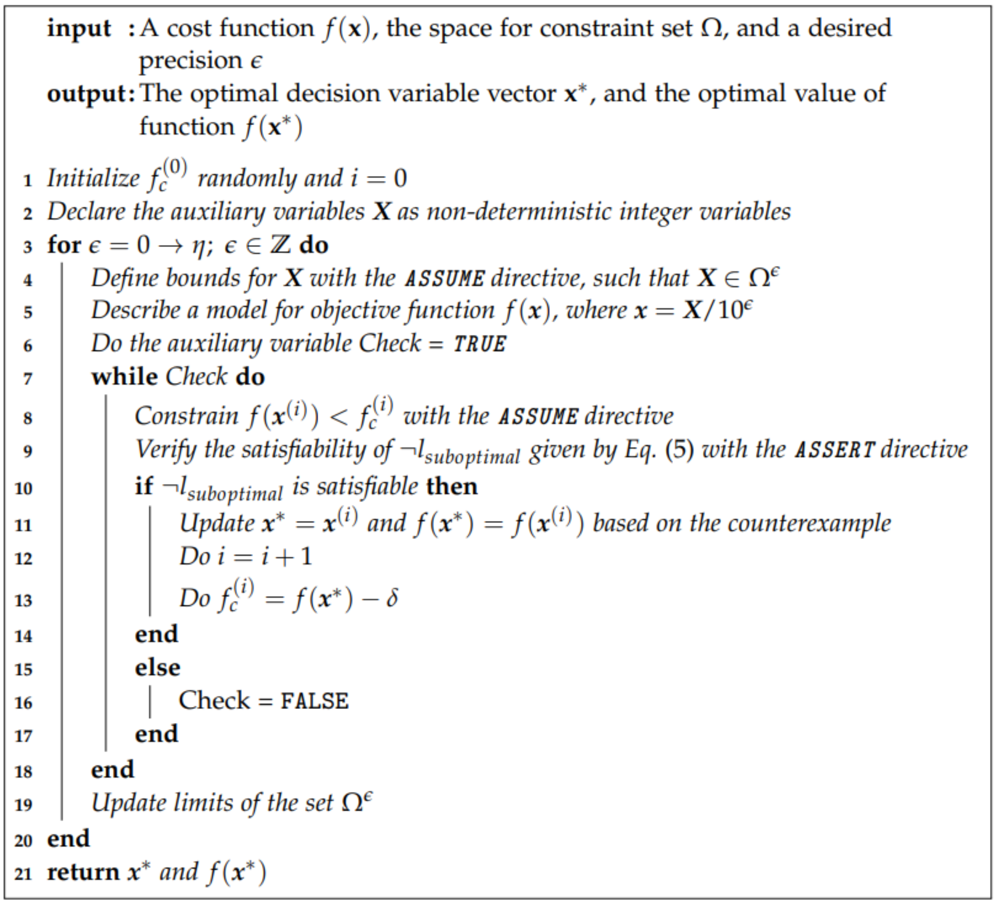
The optimality condition is modelled as:

$$l_{suboptimal} \leftrightarrow f(\boldsymbol{x}) > f_c - \delta, \qquad \textbf{(4)}$$

where δ must be sufficiently high to reduce the effects of roundoff and truncation errors in computations [40]. δ can also be used with the purpose of determining the minimum improvement at each iteration in the cost function [41].

At each iteration of the while loop, the algorithm checks if $\neg l_{suboptimal}$ is satisfiable. If there is a value for $f(\boldsymbol{x})$ such that $f(\boldsymbol{x}) < f_c$, then the decision variables vector and minimum value of cost function are updated based on the resulting counterexample, and the value of the candidate function $f_c^{(i)}$ is replaced with the new found minimum. Otherwise, the latest value of f_c is the minimum value of the cost function for the current precision ϵ.

The algorithm concludes if the outer loop reaches the limit imposed by η, which represents the desired number of decimal places of the decision variables.



```
input  : A cost function $f(\mathbf{x})$, the space for constraint set $\Omega$, and a desired
         precision $\epsilon$
output : The optimal decision variable vector $\mathbf{x}^*$, and the optimal value of
         function $f(\mathbf{x}^*)$

1  Initialize $f_c^{(0)}$ randomly and $i = 0$
2  Declare the auxiliary variables $\mathbf{X}$ as non-deterministic integer variables
3  for $\epsilon = 0 \to \eta$; $\epsilon \in \mathbb{Z}$ do
4      Define bounds for $\mathbf{X}$ with the ASSUME directive, such that $\mathbf{X} \in \Omega^{\epsilon}$
5      Describe a model for objective function $f(\boldsymbol{x})$, where $\boldsymbol{x} = \mathbf{X}/10^{\epsilon}$
6      Do the auxiliary variable Check = TRUE
7      while Check do
8          Constrain $f(\mathbf{x}^{(i)}) < f_c^{(i)}$ with the ASSUME directive
9          Verify the satisfiability of $\neg l_{suboptimal}$ given by Eq. (5) with the ASSERT directive
10         if $\neg l_{suboptimal}$ is satisfiable then
11             Update $\mathbf{x}^* = \mathbf{x}^{(i)}$ and $f(\mathbf{x}^*) = f(\mathbf{x}^{(i)})$ based on the counterexample
12             Do $i = i + 1$
13             Do $f_c^{(i)} = f(\boldsymbol{x}^*) - \delta$
14         end
15         else
16             Check = FALSE
17         end
18     end
19     Update limits of the set $\Omega^{\epsilon}$
20 end
21 return $\boldsymbol{x}^*$ and $f(\boldsymbol{x}^*)$
```

Algorithm 1. CEGIO-F

2.5.2. NOTE ON ROUNDOFF AND TRUNCATION ERRORS

The optimality condition is modelled as above in order to combat potential problems risen from verifiers which implement finite-precision arithmetic. Without the introduction of δ, solvers such as Z3 and MathSAT (which support floating-point arithmetic) are prone to roundoff and truncation errors.

There is not much that a programmer can do about the roundoff error, since it is a characteristic of computer hardware [42]. However, a truncation error represents the difference between the obtained answer of a practical calculation and the actual true answer, and it arises from the employed program or algorithm. This type of error is unavoidable even if calculation is performed on a perfect computer that had an infinitely accurate numerical representation and no roundoff error [43].

2.6 THE PATH PLANNING PROBLEM

Given a random environment, path planning can be considered the task of finding a set of positions that connect the initial and target points. This set must abide the particular constraints given by the problem (e.g. must not collide with any obstacles, any two successive positions must be connected). The path finding process typically consists of firstly representing the robot and its environment and then searching for a solution.

The parameters that define the shape of the robot determine how the robot is represented in the map. In this report the robot is considered to be a rigid object moving in a two-dimensional plane with static obstacles, and its position can be uniquely defined by two point-coordinates. The obstacles are encoded using an approximate representation, all of them having a predetermined shape (e.g. circle, square, etc).

The desired goal of path planning is to find the optimal solution, i.e. the path involving the minimum movement cost.

Definition 3. *An optimal path is a set of straight segments successively connected to guide the mobile robot from one initial point to a target point, which minimizes a cost function related to that path* [44].

Yang *et al.* remark on the symbiotic relationship between path and trajectory planning, by explaining trajectory planning as the problem of taking the solution from a path planning algorithm and determining how to move the robot along the resulting path with respect to the kinodynamic constraints, i.e. velocity, acceleration [45].

Definition 3. *A trajectory is a set of states that are associated with time, which can be described mathematically as a polynomial $X(t)$, and velocities and accelerations can be computed by taking derivatives with respect to time* [46].

2.7 OVERVIEW OF PATH PLANNING TECHNIQUES

Yang *et al*. aim to clarify on the taxonomy of path finding techniques by classifying them based on their approach of modelling the environment (kinematic constraints), the movement of the agent (dynamic constraints) and on the property of performing either on-line or off-line. Thus, they contour five different categories: sampling-based (e.g. Probabilistic Road Map, Rapidly Exploring Random Trees), node-based optimal (e.g. Dijkstra's algorithm, A*, D*), mathematical model-based (e.g. Mixed-Integer Linear Programming), bioinspired (e.g. Genetic Algorithm, Particle Swarm Optimisation) and multifusion-based algorithms [47].

Sampling-based techniques require prior complete information of the workspace, i.e. a mathematical representation. Rapidly Exploring Random Trees (RTT) explore the configuration space by sampling a new node in each step, the resulting path being the set of all sampling vertices generated by the exploration process [48]. Probabilistic Road Map (PRM) on the other hand considers a range of choices for the states of which connections are attempted [49]. Both of these techniques unfortunately offer nonoptimal solutions.

Node-based optimal algorithms encode map locations as nodes, and the cost of travelling from one node to the other is represented as edge information. Dijkstra's algorithm finds the shortest path in a graph which depends purely on local path cost [50]. A* can be seen as Dijkstra's algorithm with an added evaluation function which consists of post calculation toward the initial state and heuristic estimation toward the goal [51]. The estimation function of each state is usually the shortest path to the target, hence A* performs faster than Dijkstra's. Because of this, A* is better suited for on-line path planning.

Mathematic model-based algorithms consider the robot as a point and optimize by describing kinodynamic constrains in combination of polynomial forms. Linear algorithms, such as MILP, are able to fully describe the environment and they can also handle control disturbance and model uncertainty. MILP uses discrete decisions in an optimization procedure, thus allowing on-line application [52].

Bioinspired algorithms have originated from mimicking natural processes and are divided into two categories: Evolutionary Algorithm (EA) and Neural Network (NN) algorithm. EA includes population-based algorithms and most of them are based on the same procedures: reproduction, mutation, recombination, and selection. They are able to solve NP-hard problems but are very time costly. The most popular algorithm of this type is the Genetic Algorithm (GA), however Particle Swarm Optimization (PSO) is much faster than GA but coming with a downside of high parameter sensitivity. NN has the advantage of stability in case of sudden changes in the network, however it displays high time complexity [53].

Finally, multi-fusion-based algorithms include methods which are made of a combination of approaches in order to achieve a fast and global optimal algorithm. They are further categorised into Embedded Multifusion Algorithms (EMA) and Ranked Multifusion Algorithms (RMA). As an example of a RMA solution, PRM cannot compute an optimal path on its own, so it is combined with A*. This results in global path optimisation.

Yang *et al.* offer a complete and much needed classification of path planning techniques, though others can be found in literature.

CHAPTER 3 – TRAJECTORY PLANNING PRELIMINARIES

3.1 LEGO MINDSTORMS

Lego Mindstorms EV3 Home is a robotics kit designed to make computer science more approachable for young users by also providing an easy to understand visual programming language. However, its popularity has exceeded the target clientele since the programmable brick of EV3 has a powerful ARM9 processor operating at 300 Mhz with a 64MB of RAM and 16MB of flash drive. It supports multiple programming languages, such as Java, C/C++ and C#, and it runs a Linux distribution as the operating system.

The kit contains four sensors (colour, touch, ultrasonic, gyroscopic), two types of motors (large and medium) and various technical building components. The programmable brick has a 178×128 pixel monochrome LCD display with a six-button interface and it also comes with WiFi and Bluetooth connectivity. Connection with the computer can be done also via the integral USB 2.0.

3.2 ROBOT DESIGN

The final version of the presented EV3 robot (Picture 1) uses front-wheel drive transmission by connecting the two large motors only to the front wheels of the robot. The rear wheels do not have tires in order to minimise friction with the terrain. It does not make use of any of the previously mentioned sensors, as the objective is to implement a functionable trajectory planning algorithm using only the built-in motor encoders and mathematical geometrical functions.

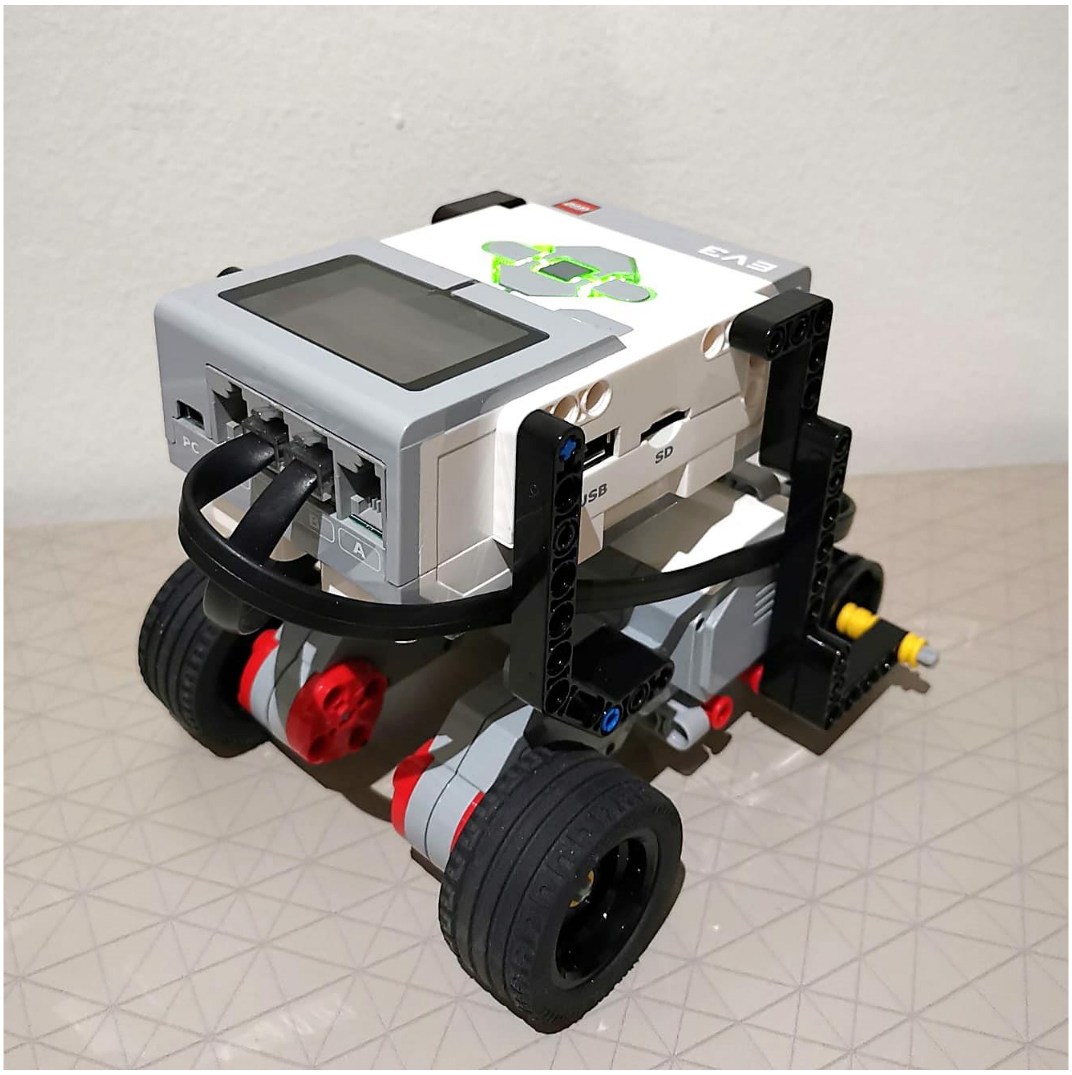
The design of the robot was changed along the way based on the observational movement tested on a hard surface floor. Naturally, the characteristics of the environment heavily influence the robot's performance: if tested on a fuzzy surface (e.g. carpet) or one which displays different areas of high angles of inclination, the presented robot will not move ideally. However, this problem is beyond the scope of this paper and is presented as a proposed future improvement.

The previous version of the robot was based on the same skeleton, with three wheels on each side connected by a rubber band, similar to the caterpillar track used for military vehicles. This design was aborted after observing that sections of the band do not make continuous contact with the tested surface, mainly caused by the loss of elasticity over time and erosion of the band's tread. This resulted in a higher movement error, the robot tending to displace by approximately 10 cm on the right side from the goal when a moving distance of 100 cm is required.

The changes made for the final presented version improve significantly the aforementioned error, with a displacement of 1 to 2 cm in experimental evaluation. Note that this data is observed in the case of a fully charged battery.

3.2 SOFTWARE

Coding the robot can be done in different ways based on the user's preference to a particular programming language. The chosen programming method is by using ROBOTC, which is Integrated Development Environment (IDE) that extends the C programming language. It comes with a large number of built-in variables and functions in order to provide control over the robot's hardware components, mainly its motors and sensors. There are two compiler targets: Physical Robot and Virtual Worlds, which is a high-end simulation environment. The ROBOTC compiler has a powerful code optimizer that shrinks the program by half of its size before being sent to the robot.



Picture 1. Robot built with Lego Mindstorms EV3 kit

CHAPTER 4 - CEGIO-BASED PATH PLANNING

The main objective of a path planning algorithm is to generate points needed to guide the mobile robot in an environment with obstacles. Usually this implies a considerable amount of processing time, as the complexity of the problem increases with the number of points visited and the different types of obstacles. As a result, there is an obvious necessity for developing new methods that can be employed for optimal path finding.

4.1 PROBLEM FORMULATION

Using the CEGIO-based approach to the path planning problem consists on firstly formulating it as an optimisation problem, then encoding the given environmental data, i.e. environmental limits, position and radius of obstacles and finally using a path search method that consists of points in the movement space to find the optimal path that satisfies the given constraints [54].

Providing definitions for the cost function and problem constraints is necessary for a complete optimisation problem formulation.

Definition 7. *Cost function:* Given the starting point (**S**) and the target point (**T**) defined as $S = P_1$ and $T = P_n$, the objective is to find a decision variables matrix, $L = [P_1, P_2, \ldots, P_{n-1}, P_n]$, such that, $J(L)$ is the path length function. The cost function is therefore defined as:

$$J(L) = \sum_{i=1}^{n-1} ||P_{i+1} - P_i|| \,, \tag{5}$$

where n is the number of points that compose the path and for the bi-dimensional case, $P_i = (x_i, y_i)$ is a path vertex. Note that, if $n \to \infty$ the path will be a smooth trajectory; furthermore, the optimization problem dimension will also tend to infinity [55].

Definition 8. *Constraints:* Each point $p_{i\lambda}$ that composes the i-th straight segments must not intercept any obstacle and must be within environmental limits [56].

Resulting from **Definitions 1, 7** and **8**, the path finding optimisation problem can be written as:

$$\begin{array}{ll} \min\limits_{L} & J(L), \\ & p_{i\lambda}(L) \notin \mathbf{O} \\ s.t. & p_{i\lambda}(L) \in \mathbf{E} \\ & i = 1, \ldots, n-1, \end{array} \tag{6}$$

where **O** is the set of points defined by obstacles; **E** is the set of points defined by environment limits; n is the number of points that compose the path; and $p_{i\lambda}(L)$ represent the set of points belonging to the i-th straight segment of the path defined by vector L.

Each $p_{i\lambda}(L)$ point is defined as:

$$p_{i\lambda}(L) = (1 - \lambda)P_i + \lambda P_{i+1}, \forall \lambda \in [0, 1] \quad [57]. \tag{7}$$

Regarding environmental modelling, for simplicity, we will consider the lower and upper limits of the decision variable set to define a rectangle, i.e the dimensions of the map. Each individual obstacle is associated with a set of points (x_i, y_i) which define its position with respect to the centre of the obstacle, and a circle radius such that it surrounds the whole shape of the obstacle. The equation that models the interaction between obstacles and path segments is as follows:

$$(x_{i\lambda} - x_0)^2 + (y_{i\lambda} - y_0)^2 \geq (r + \sigma)^2, \qquad \textbf{(8)}$$

where σ is a safety margin [58].

4.2 PATH PLANNING ALGORITHM

4.2.1 MODELLING

Modelling consists of defining the constraints given by the problem, which is of utter importance since ESBMC is not efficient for unconstrained optimization. Note that in the above path finding formulation given by Eq. (6), the state-space search is therefore reduced to the numeric interval based on environment limits, excluding members which define static obstacles. Based on this, Fig. 2 represents the resulting C code after the modelling step.

The matrix **x** holds the coordinates for each point of the path, excluding the given initial and target positions. Note that **x** is declared as an integer number and is initialised with non-deterministic values.

The ASSUME statements in lines 21-24 are used to constrain the state-space search by reducing the possible values of **x** to be within the map limits. Additional ASSUME functions are added based on preliminary conventions. Line 26 conditions path points not to be on the same column. Line 29 ensures that the y-coordinate of the first generated point cannot be equal to the y-coordinate of the given initial position. In the same fashion, line 30 ensures that the y-coordinate of the last generated point cannot be equal to the y-coordinate of the given target position.

A function *rest_points* (Fig. 1) is used to insert additional constraints on the path points based on obstacle information. This function must be called for every individual obstacle and uses Eq. (8) to assume the distance between the agent and obstacle to be within limits.

Finally, the objective function J is computed as the sum of the lengths of the found path segments.

4.2.2 SPECIFICATION

This second step consists of describing the system behaviour and the property to be checked. The result is the C program shown in Fig. 3, which will be iteratively given to the chosen verifier.

An integer variable p (line 2) is created to represent the path points coordinates precision such that

$$k > \log p, \qquad \textbf{(9)}$$

where k represents the number of decimal places of the coordinate variables. Now, the decision variables' initialisation depends directly on p. For each successive execution the precision is increased by multiplying p by 10 in order to converge to an optimal solution.

Another variable is needed to hold the candidate value for the cost function, called J_i. It must be initialised with a random high value for the first iteration of the verifier. For example, in Fig. 3 the candidate function is initialised to 25.

An additional constraint needs to be inserted (line 62), namely the one which models the assumption that the current value of the cost function J is smaller than the new found value (J_i). This corresponds to line 10 of Algorithm 2, which imposes that the new value of the objective function must be less than the value obtained in the previous iteration. At the beginning, all members of the state-space search Ω are considered optimal candidates, and this particular constraint helps to remove a number of candidates on each iteration.

To ensure convergence to the minimum point on each iteration, a final property needs to be modelled with the help of the ASSERT statement. Line 63 verifies whether the literal $J_{optimal}$ given by the formula below is satisfiable for the current value of J_i.

$$J_{optimal} \Leftrightarrow J(\boldsymbol{L}) > J\_i \quad \textbf{(10)}$$

The verification procedure stops when $\neg J_{optimal}$ is found satisfiable, meaning that the generated counterexample will contain a value for J which is smaller than the current value J_i. If it is impossible to find an optimal solution with the current number of points, n, in the path, then the number of points is increased automatically. If $J_{optimal}$ is found repeatedly unsatisfiable, then p must be increased as discussed above.

Note that the value of J_i must be updated for each iterative execution with the value returned from the counterexample, which can be done with the help of scripts.

4.2.3 VERIFICATION

In the final step of the proposed methodology, the C program shown in Fig. 3 is given to ESBMC as input and a counterexample will be returned in the case of an unsuccessful verification, containing a set of decision variables $\boldsymbol{x}$ and the value of the resulting cost function. In the case of a successful verification of a C program, this means that ESBMC has found the optimal value for the specified objective function, for a precision defined by p and number of points in the path, n.

As an example, for the algorithm shown in Fig. 3, a counterexample with the following data is generated: $x[0][0] = 2, x[0][1] = 7$ and $J = 12.28$. These values are used for calculating the next minimum value of the cost function, as seen in Algorithm 2, thus getting closer to the global optimal solution on each iteration. Consequently, the data extracted from the counterexample is of crucial importance for state-space search reduction and algorithm convergence.

The number of points in the path n impacts Algorithm 2's efficiency. As expected, the problem's complexity increases as n is updated with a bigger value. However, a large number of path points is not needed, since smoothness (i.e. interpolating the points found) is handled by trajectory planning [59].

4.3 EXPERIMENTAL EVALUATION

4.3.1 DESCRIPTION AND OBJECTIVES

This section covers two designed experiments in order to evaluate the proposed CEGIO-based method. Both of them share the same goal of generating the optimal path from the initial (**I**) to the target (**T**) point. As the robot is modelled as a point defined by two coordinates, the ratio between the unit distance in the context of the path planning algorithm and real-life distance is 14cm, which is the diameter of the robot.

In the first experiment, the map is a 10×10 square and it contains only one obstacle, with the centre positioned at $C(x, y) = (5{,}5)$ and radius $r = 2.5$. The second one involves two obstacles, with $C_1(x_1, y_1) = (3{,}3)$, $C_2(x_2, y_2) = (7, 6)$ and $r_1 = 1, r_2 = 2$. Both experiments have a value of $\sigma = 0.5$.

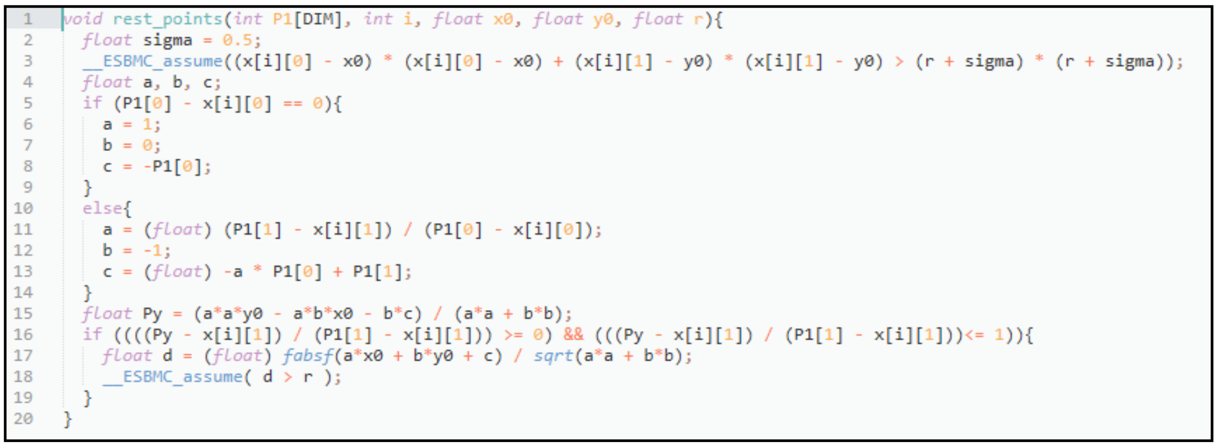
4.3.2 EXPERIMENTAL SETUP

For the cases presented above the employed software verifier is ESBMC v6.1.0 with the Boolector v3.0.0 solver. The experiments were conducted in an idle Intel Core i7-8750H 2.20 GHz processor, with 16 GB of RAM and running Ubuntu 18.04.4 64-bits. The time measuring unit is in seconds based on the CPU time and memory consumption is not restricted. The time presented for the first setting is the average of five executions, however due to the large amount of time, the second experiment was only performed once.

4.3.2 EXPERIMENTAL RESULTS AND CONCLUSION

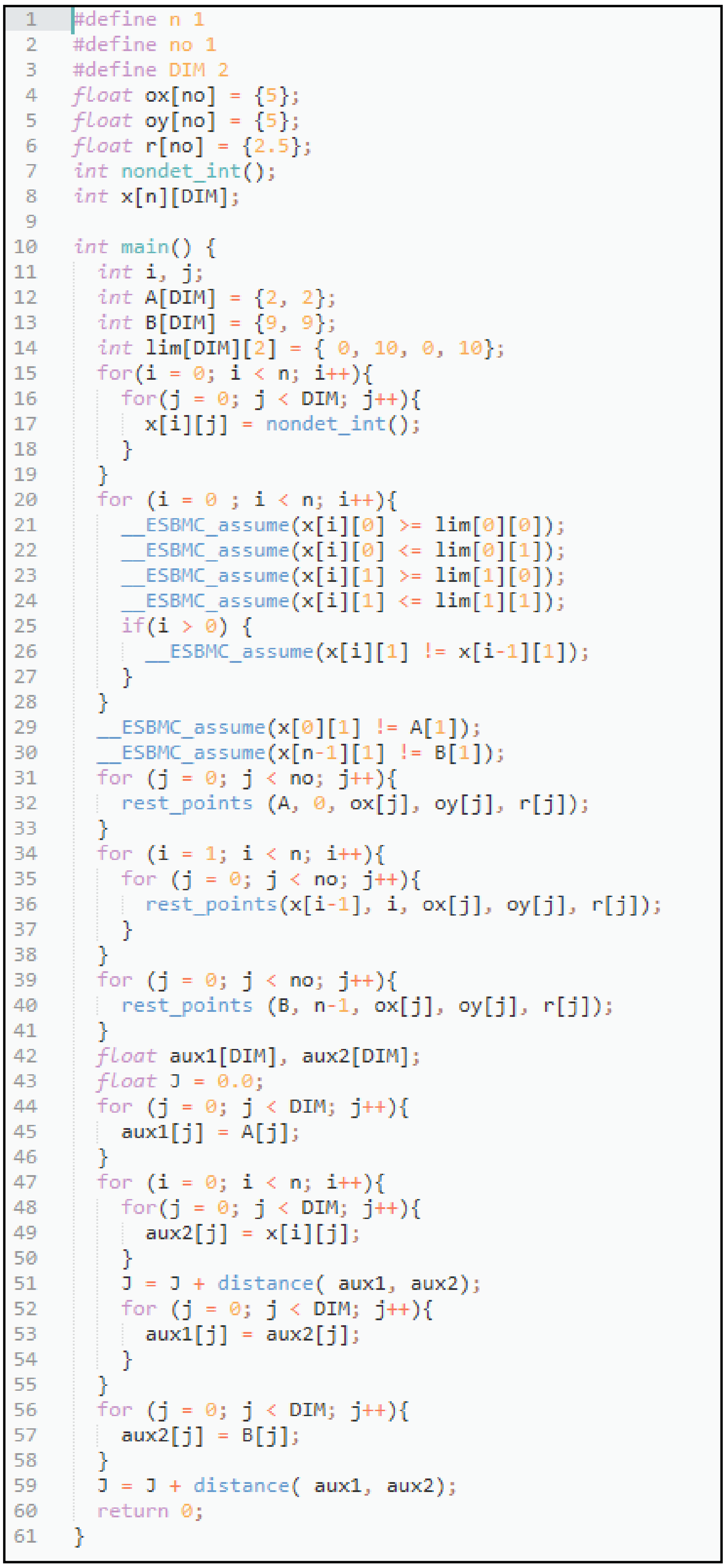
The first experiment took 6 hours to conclude and a path of two points ($n = 2$) was obtained. The second setting ended after 13 hours and found a value for $n = 5$.

Compared to other path finding techniques described in **Section 2.6,** the presented CEGIO-based algorithm finds the global optimal solution, i.e. the shortest path, but comes at the expense of high execution time.



```c
void rest_points(int P1[DIM], int i, float x0, float y0, float r){
  float sigma = 0.5;
  __ESBMC_assume((x[i][0] - x0) * (x[i][0] - x0) + (x[i][1] - y0) * (x[i][1] - y0) > (r + sigma) * (r + sigma));
  float a, b, c;
  if (P1[0] - x[i][0] == 0){
    a = 1;
    b = 0;
    c = -P1[0];
  }
  else{
    a = (float) (P1[1] - x[i][1]) / (P1[0] - x[i][0]);
    b = -1;
    c = (float) -a * P1[0] + P1[1];
  }
  float Py = (a*a*y0 - a*b*x0 - b*c) / (a*a + b*b);
  if ((((Py - x[i][1]) / (P1[1] - x[i][1])) >= 0) && (((Py - x[i][1]) / (P1[1] - x[i][1]))<= 1)){
    float d = (float) fabsf(a*x0 + b*y0 + c) / sqrt(a*a + b*b);
    __ESBMC_assume( d > r );
  }
}
```

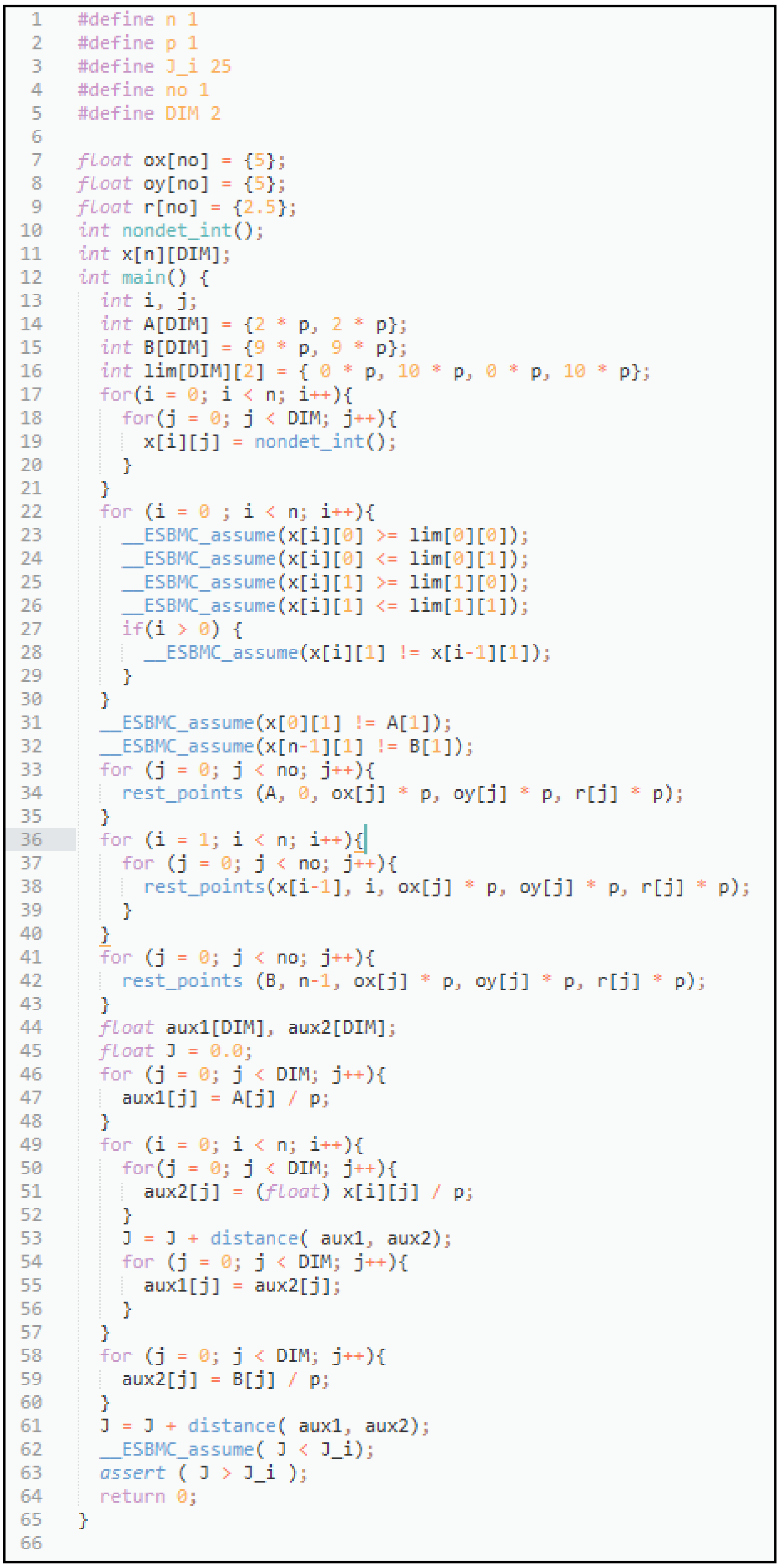
Fig 1. *rest_points* function



```c
#define n 1
#define no 1
#define DIM 2
float ox[no] = {5};
float oy[no] = {5};
float r[no] = {2.5};
int nondet_int();
int x[n][DIM];

int main() {
  int i, j;
  int A[DIM] = {2, 2};
  int B[DIM] = {9, 9};
  int lim[DIM][2] = { 0, 10, 0, 10};
  for(i = 0; i < n; i++){
    for(j = 0; j < DIM; j++){
      x[i][j] = nondet_int();
    }
  }
  for (i = 0 ; i < n; i++){
    __ESBMC_assume(x[i][0] >= lim[0][0]);
    __ESBMC_assume(x[i][0] <= lim[0][1]);
    __ESBMC_assume(x[i][1] >= lim[1][0]);
    __ESBMC_assume(x[i][1] <= lim[1][1]);
    if(i > 0) {
      __ESBMC_assume(x[i][1] != x[i-1][1]);
    }
  }
  __ESBMC_assume(x[0][1] != A[1]);
  __ESBMC_assume(x[n-1][1] != B[1]);
  for (j = 0; j < no; j++){
    rest_points (A, 0, ox[j], oy[j], r[j]);
  }
  for (i = 1; i < n; i++){
    for (j = 0; j < no; j++){
      rest_points(x[i-1], i, ox[j], oy[j], r[j]);
    }
  }
  for (j = 0; j < no; j++){
    rest_points (B, n-1, ox[j], oy[j], r[j]);
  }
  float aux1[DIM], aux2[DIM];
  float J = 0.0;
  for (j = 0; j < DIM; j++){
    aux1[j] = A[j];
  }
  for (i = 0; i < n; i++){
    for(j = 0; j < DIM; j++){
      aux2[j] = x[i][j];
    }
    J = J + distance( aux1, aux2);
    for (j = 0; j < DIM; j++){
      aux1[j] = aux2[j];
    }
  }
  for (j = 0; j < DIM; j++){
    aux2[j] = B[j];
  }
  J = J + distance( aux1, aux2);
  return 0;
}
```

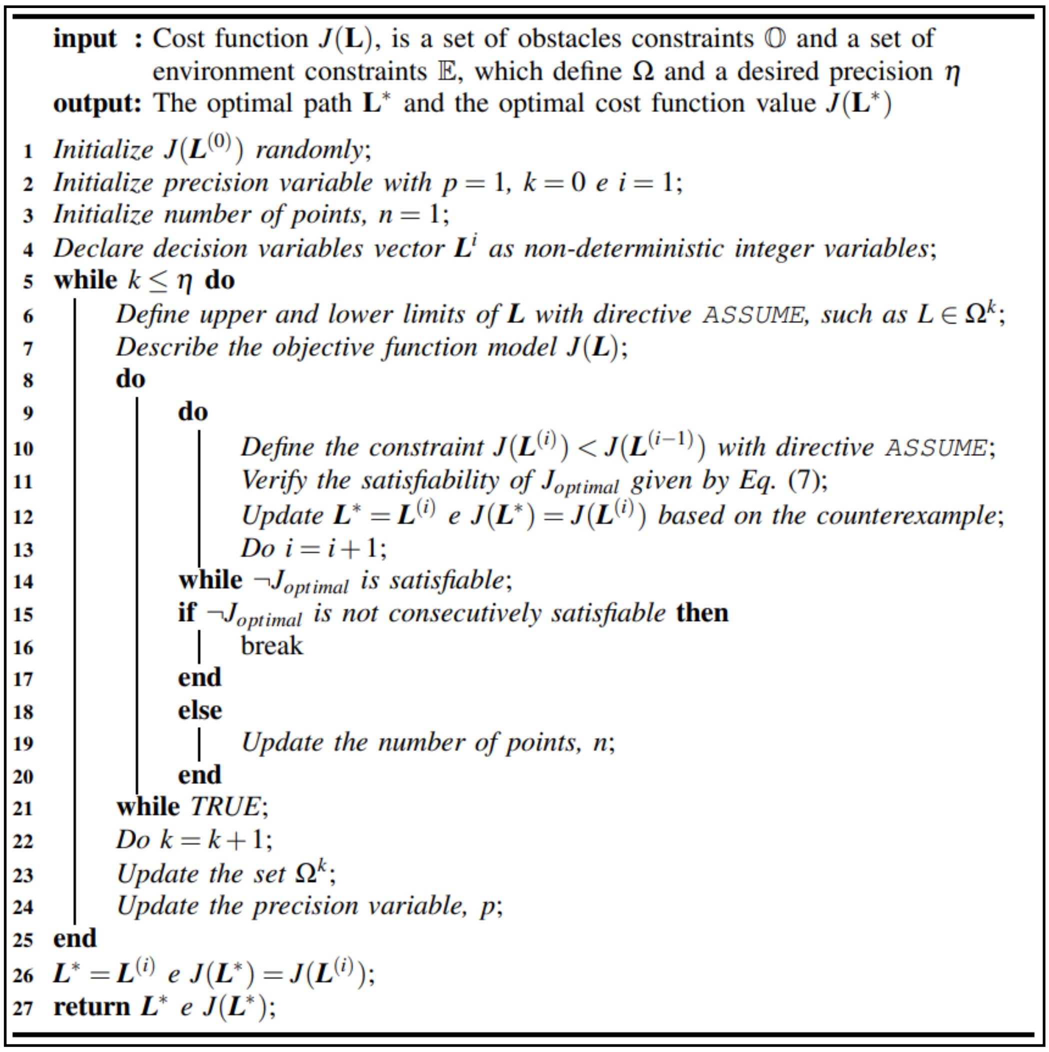
Figure 2. C code after modelling step



```c
#define n 1
#define p 1
#define J_i 25
#define no 1
#define DIM 2

float ox[no] = {5};
float oy[no] = {5};
float r[no] = {2.5};
int nondet_int();
int x[n][DIM];
int main() {
  int i, j;
  int A[DIM] = {2 * p, 2 * p};
  int B[DIM] = {9 * p, 9 * p};
  int lim[DIM][2] = { 0 * p, 10 * p, 0 * p, 10 * p};
  for(i = 0; i < n; i++){
    for(j = 0; j < DIM; j++){
      x[i][j] = nondet_int();
    }
  }
  for (i = 0 ; i < n; i++){
    __ESBMC_assume(x[i][0] >= lim[0][0]);
    __ESBMC_assume(x[i][0] <= lim[0][1]);
    __ESBMC_assume(x[i][1] >= lim[1][0]);
    __ESBMC_assume(x[i][1] <= lim[1][1]);
    if(i > 0) {
      __ESBMC_assume(x[i][1] != x[i-1][1]);
    }
  }
  __ESBMC_assume(x[0][1] != A[1]);
  __ESBMC_assume(x[n-1][1] != B[1]);
  for (j = 0; j < no; j++){
    rest_points (A, 0, ox[j] * p, oy[j] * p, r[j] * p);
  }
  for (i = 1; i < n; i++){
    for (j = 0; j < no; j++){
      rest_points(x[i-1], i, ox[j] * p, oy[j] * p, r[j] * p);
    }
  }
  for (j = 0; j < no; j++){
    rest_points (B, n-1, ox[j] * p, oy[j] * p, r[j] * p);
  }
  float aux1[DIM], aux2[DIM];
  float J = 0.0;
  for (j = 0; j < DIM; j++){
    aux1[j] = A[j] / p;
  }
  for (i = 0; i < n; i++){
    for(j = 0; j < DIM; j++){
      aux2[j] = (float) x[i][j] / p;
    }
    J = J + distance( aux1, aux2);
    for (j = 0; j < DIM; j++){
      aux1[j] = aux2[j];
    }
  }
  for (j = 0; j < DIM; j++){
    aux2[j] = B[j] / p;
  }
  J = J + distance( aux1, aux2);
  __ESBMC_assume( J < J_i);
  assert ( J > J_i );
  return 0;
}
```

Figure 3. C code after specification step



```
input  : Cost function $J(\mathbf{L})$, is a set of obstacles constraints $\mathbb{O}$ and a set of
         environment constraints $\mathbb{E}$, which define $\Omega$ and a desired precision $\eta$
output: The optimal path $\mathbf{L}^*$ and the optimal cost function value $J(\mathbf{L}^*)$
Initialize $J(\boldsymbol{L}^{(0)})$ randomly;
Initialize precision variable with $p = 1$, $k = 0$ e $i = 1$;
Initialize number of points, $n = 1$;
Declare decision variables vector $\boldsymbol{L}^i$ as non-deterministic integer variables;
while $k \leq \eta$ do
    Define upper and lower limits of $\mathbf{L}$ with directive ASSUME, such as $L \in \Omega^k$;
    Describe the objective function model $J(\mathbf{L})$;
    do
        do
            Define the constraint $J(\mathbf{L}^{(i)}) < J(\mathbf{L}^{(i-1)})$ with directive ASSUME;
            Verify the satisfiability of $J_{optimal}$ given by Eq. (7);
            Update $\boldsymbol{L}^* = \mathbf{L}^{(i)}$ e $J(\boldsymbol{L}^*) = J(\mathbf{L}^{(i)})$ based on the counterexample;
            Do $i = i + 1$;
        while $\neg J_{optimal}$ is satisfiable;
        if $\neg J_{optimal}$ is not consecutively satisfiable then
            break
        end
        else
            Update the number of points, $n$;
        end
    while TRUE;
    Do $k = k + 1$;
    Update the set $\Omega^k$;
    Update the precision variable, $p$;
end
$\boldsymbol{L}^* = \boldsymbol{L}^{(i)}$ e $J(\boldsymbol{L}^*) = J(\boldsymbol{L}^{(i)})$;
return $\boldsymbol{L}^*$ e $J(\boldsymbol{L}^*)$;
```

Algorithm 2. Path planning algorithm proposed by Araújo *et al.*

CHAPTER 5 – TRAJECTORY PLANNING

5.1 TRAJECTORY PLANNING IMPLEMENTATION

The presented trajectory planning is implemented on a Lego Mindstorms EV3 robot (Picture. 1) using the ROBOTC IDE. It uses basic geometrical formulas and the built-in motors encoders in order to guide the agent in a predefined environment. As mentioned in **Chapter 3**, the following method needs to receive as input the coordinates of the path points resulted from the CEGIO-based path planning algorithm, along with the start and target coordinates. The first experiment's results from Section 4.3 are chosen as an example. Consequently, the robot assumes that the given environment is obstacle-free, i.e. has no knowledge of obstacle information.

The designed model has the following parameters: the radius of the robot is 7 cm and the two front wheels have a radius of 4 cm. The robot is represented as a point coordinate and the starting position is conventionally oriented to the North of the map. The variable representing the orientation holds values from 0 to 359, with 0 representing the North direction. After each move, the new orientation of the robot is computed in order to guide its next step.

This process can be seen in Fig. 4. The *beginNextMove* function takes as input the coordinates for the current and target positions and the current robot orientation. The distance between the two input points is calculated based on the Euclidian distance formula. In order for the robot to turn in the right direction before moving forward, the rotation angle in computed with the help of the *computeRotationAngle* function.

To call this function, it is firstly needed to calculate the rotation angle relative to the Y-axis of the map as seen in Fig. 5. The *computeAngleOy* function is based on the formula for calculating the slope of a line.

Definition 9. *Given any two points on a line, the slope of the line is given by the formula:*

$$slope = (y_A - y_B)/(x_A - x_B),$$

where (x_A, y_A) are the coordinates of the first point A and (x_B, y_B) are the coordinates of the second point B.

The if statements adjust the output of the function with regards to the resulted value of the slope. This adjustment is made based on which of the four circle quadrants includes the aforementioned value. The output of this function is an integer ranging from 0 to 359, which represent the angle value in degrees.

This result is given as input along with the current orientation of the robot for the *computeRotationAngle* function. The function returns the difference between the input angles, thus obtaining the rotation angle needed before the start of the next move.

The for loop in the main function (Fig. 4) iteratively computes the necessary robot movement for the set of points. The *beginNextMove* function (Fig. 6) decides the complete movement of the robot based on the given input including the rotation (if needed) and moving forward commands. The return value is an integer representing the resulted orientation after the movement. It can have two different implementations, depending on the way the rotation execution is wanted. More specifically, two functions are provided for this aspect: *rotateRight* and *moveRight*.

The *moveForwardEncoder* function aims to minimise the displacement error of the robot by adjusting the power given to each motor such that they run synchronously. Two corresponding functions are created for backwards movement without the balancing of power, as they are called only for small distances which do not result in any displacement errors.

The first rotation function, *rotateRight*, provides a basic rotation to the right around the centre point of the robot; however, this results in a displacement of a few centimetres (maximum 2 cm) compared to the ideal rotation. The cause for this is concluded to be the lack of power for the back wheels, which would have helped by providing the necessary traction. In order to solve this problem, a *moveRight* function has been created.

After the *moveRight* call, the robot will be rotated, however the left front wheel will remain in the same physical place. This function uses the *moveBackwardsCM* function, which is called in order to move the robot backwards for a distance equal to the front wheel's diameter (4 cm). However, by using *moveRight*, the time taken to complete the path following is greater compared to using the more straightforward function, *rotateRight*. This is due to the extra steps taken to align the robot in the perfect position. Additional display outputs have been added for each movement function, the scope being an easier verification during testing.

Note that the calculation of necessary encoder ticks is different for forward and rotation movement. In the case of forward movement, the function shown in Fig. 7 is used, whereas for rotation *tickCountRotate* (Fig. 8) needs to be called. The integer output of both functions is used as the upper limit in while statements for all movement methods. The employed technique assures a correct movement of the robot without the use of a gyroscopic sensor.

As this report is focused with optimisation, is it naturally recommended to use the *rotateRight* and respective *rotateLeft* functions for time efficiency, especially since the displacement error is very small. If physical place precision is considered more important, then the version which implements *moveRight* should be employed. A solution that balances both location precision and time optimisation cannot be created with the current design of the robot.

5.2 EXPERIMENTAL EVALUATION

5.2.1 DESCRIPTION AND OBJECTIVES

This section covers the designed experiments in order to evaluate the proposed trajectory planning method, based on the results of the first experiment from Section 3.3. They all share the same goal of reaching the target destination by following a predefined path. Since the robot is modelled as a point, the ratio presented in Section 3.3 is maintained.

The purpose of this experimentation is to see which of the two rotating functions discussed above generates the best result, as the forward command gives the expected constant output. The robot is tested on three different types of surfaces: hard with zero inclination terrain, hard with low inclination and soft terrain. Setting 1 uses the *rotateRight* function and Setting 2 uses the *moveRight* function. All the experiments are performed with a power of 30 for the motors and the results presented are the average of ten executions.

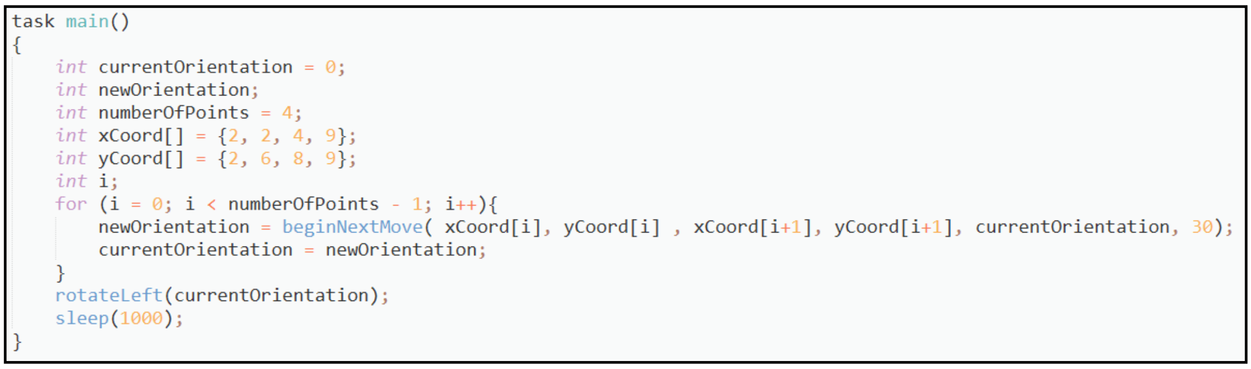
5.2.2 EXPERIMENTAL RESULTS AND CONCLUSION

All the experiments took an average time of 10 seconds to complete for the *rotateRight* case, and for *moveRight* the average time was 18 seconds.

In the first scenario, the robot moves on a hard terrain with zero inclination and uses the *rotateRight* function. This naturally gives the best time results, and the distance between the final position of the robot and the actual target position is 1.6 cm. In the case of using *moveRight*,, the difference between positions is of 0.9 cm, which is a significant improvement but comes at the cost of longer execution time.

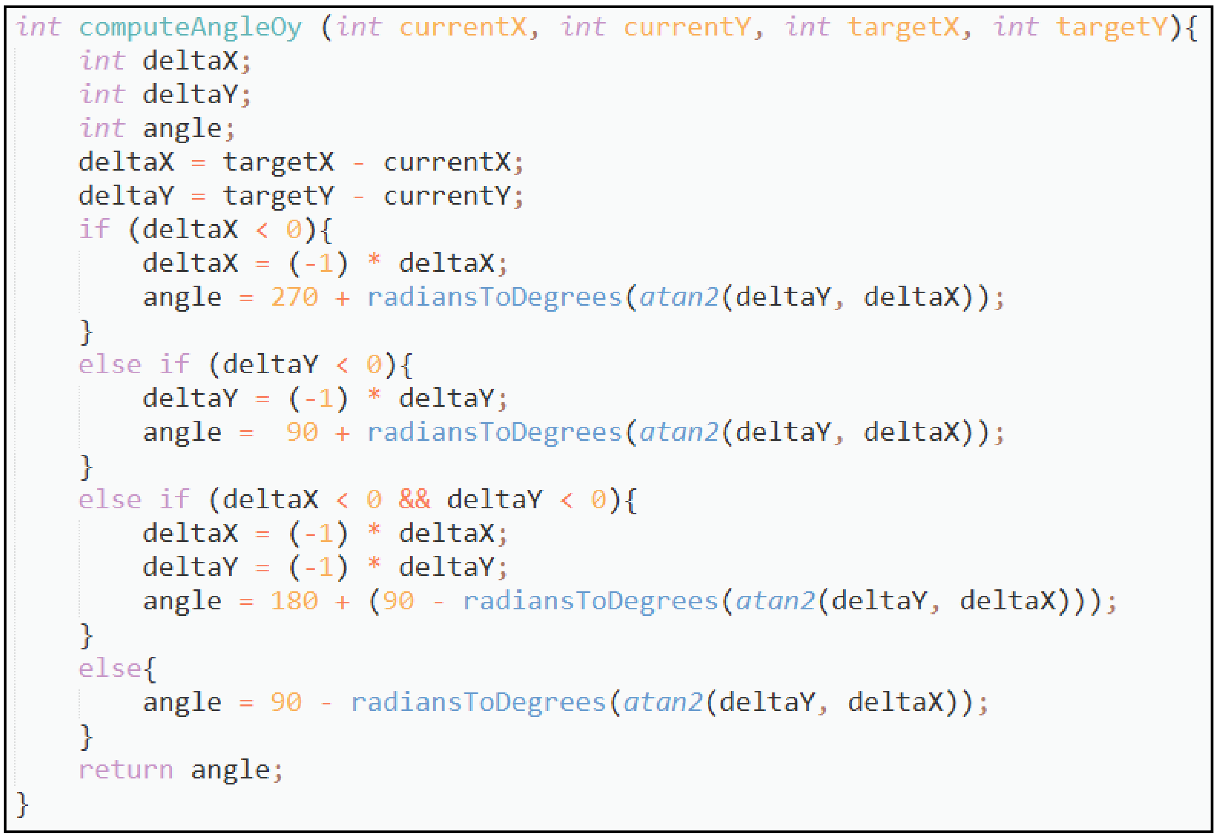
The second scenario consists of hard terrain with a very low inclination (smaller than 5 degrees) and the results are as follows: a difference of 2.3 cm for *rotateRight* and 1.9 cm for *moveRight*. The last scenario offers the most disappointing results. Due to the lack of friction between the robot's wheels and fuzzy surface, the robot barely changes its initial position.

Therefore, it can be concluded that the presented method is a feasible solution for trajectory planning, however improvements must be made with more consideration to terrain characteristics.



```
task main()
{
    int currentOrientation = 0;
    int newOrientation;
    int numberOfPoints = 4;
    int xCoord[] = {2, 2, 4, 9};
    int yCoord[] = {2, 6, 8, 9};
    int i;
    for (i = 0; i < numberOfPoints - 1; i++){
        newOrientation = beginNextMove( xCoord[i], yCoord[i] , xCoord[i+1], yCoord[i+1], currentOrientation, 30);
        currentOrientation = newOrientation;
    }
    rotateLeft(currentOrientation);
    sleep(1000);
}
```

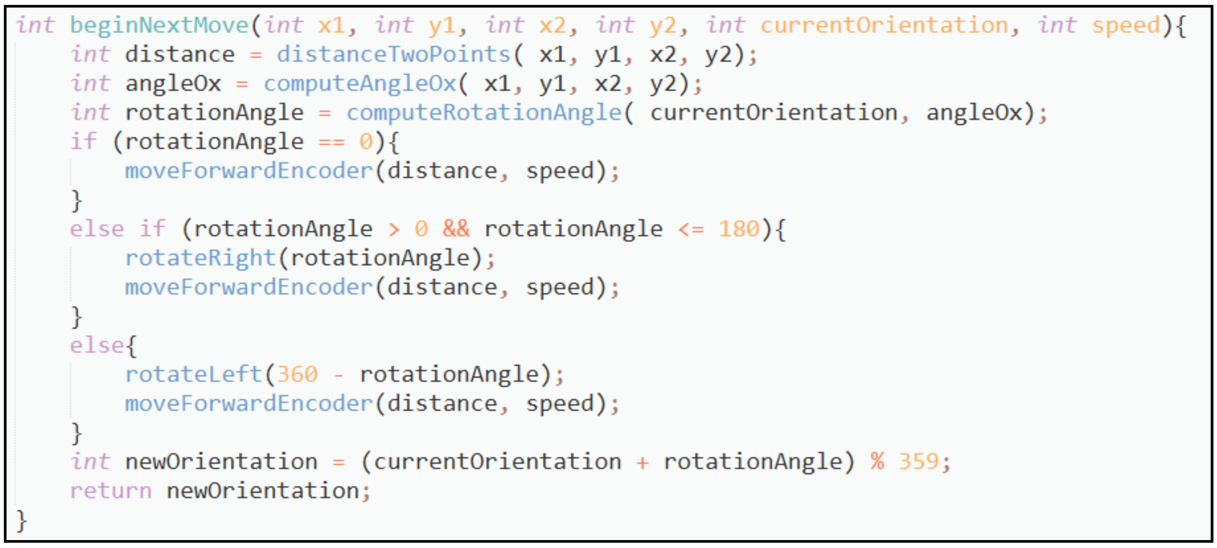
Figure 4. Main function of trajectory planning



```
int computeAngleOy (int currentX, int currentY, int targetX, int targetY){
    int deltaX;
    int deltaY;
    int angle;
    deltaX = targetX - currentX;
    deltaY = targetY - currentY;
    if (deltaX < 0){
        deltaX = (-1) * deltaX;
        angle = 270 + radiansToDegrees(atan2(deltaY, deltaX));
    }
    else if (deltaY < 0){
        deltaY = (-1) * deltaY;
        angle =  90 + radiansToDegrees(atan2(deltaY, deltaX));
    }
    else if (deltaX < 0 && deltaY < 0){
        deltaX = (-1) * deltaX;
        deltaY = (-1) * deltaY;
        angle = 180 + (90 - radiansToDegrees(atan2(deltaY, deltaX)));
    }
    else{
        angle = 90 - radiansToDegrees(atan2(deltaY, deltaX));
    }
    return angle;
}
```

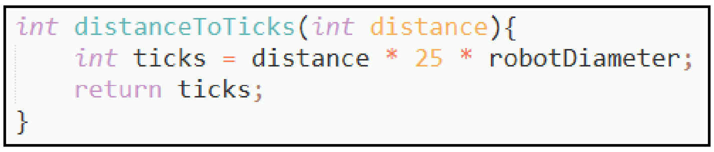
Figure 5. Function that computer the robot's orientation

with respect to the Y-axis of the map



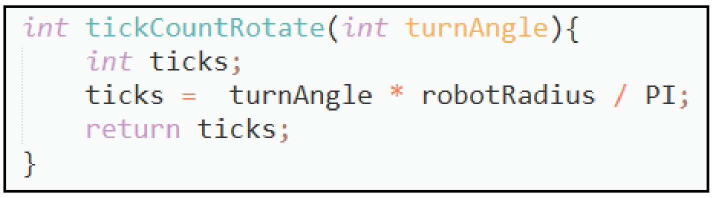
```
int beginNextMove(int x1, int y1, int x2, int y2, int currentOrientation, int speed){
    int distance = distanceTwoPoints( x1, y1, x2, y2);
    int angleOx = computeAngleOx( x1, y1, x2, y2);
    int rotationAngle = computeRotationAngle( currentOrientation, angleOx);
    if (rotationAngle == 0){
        moveForwardEncoder(distance, speed);
    }
    else if (rotationAngle > 0 && rotationAngle <= 180){
        rotateRight(rotationAngle);
        moveForwardEncoder(distance, speed);
    }
    else{
        rotateLeft(360 - rotationAngle);
        moveForwardEncoder(distance, speed);
    }
    int newOrientation = (currentOrientation + rotationAngle) % 359;
    return newOrientation;
}
```

Figure 6. Function that controls the next step of the robot



```
int distanceToTicks(int distance){
    int ticks = distance * 25 * robotDiameter;
    return ticks;
}
```

Figure 7. Function that computes the amount of ticks needed in order to move a given distance



```
int tickCountRotate(int turnAngle){
    int ticks;
    ticks =  turnAngle * robotRadius / PI;
    return ticks;
}
```

Figure 8. Function that computes the amount of ticks needed for a rotation of given angle

CHAPTER 6 – CONCLUSION

6.1 REFLECTIONS

The main reason of choosing this project has been a desire to work on something that involved robotics, an area completely unfamiliar which I was eager to dive into. Before the start of the year, I had no previous knowledge of model checking or what it entailed, and because of that I naively thought I could grasp the topic within a short amount of time. The Gantt chart proposed at the middle of the first semester was deemed completely inaccurate, as all the original self-imposed deadlines were not respected by approximately one month.

The reason for this was due to the prolonged time that had to be dedicated to research in order to contextualise the original paper. Being familiar only with node-based path planning techniques from the second year of studies, the jump to the CEGIO-based approach was refreshing, but the technique was harder to comprehend. Without the appropriate level of background, the technique presented by Araújo *et al.* is an unsolvable puzzle.

After reading the recommended materials, I was not confident in my understanding of BMC, so further research on its history has been made. Along with acquiring new information, this has also led me to strengthen some previously learnt notions that were not precise enough. As for the software aspect, learning how the ASSUME and ASSERT work was the most challenging part, several try-out functions unrelated to path planning being tested.

Regarding the trajectory planning part of the project, originally my work began on a Arduino.CC robot, which was unusable at the beginning of the second semester due to technical failures. At the recommendation of my tutor, I switched to a Lego Mindstorms EV3 robot which involved building the actual robot unlike the Arduino one.

The different versions of the robot have helped me see how much the problem changes and how to adjust the code accordingly, an aspect I did not think I had to consider. The final version of EV3 balances its weight almost evenly on its wheels, since this has shown to generate the smallest displacement error while moving. The task of programming the robot required a surprising amount knowledge of geometry, which was used in the movement functions and parameter generation.

6.2 FURTHER WORK

The following improvements and ideas are presented as possible further work:

1. Autonomous robot adjustment depending on environmental characteristics
2. Applying the CEGIO-based algorithm for multi-agent path finding

References

[1] Anonymous.(4 March 2020). Autonomous Navigation Market Growth Analysis [online], MarketWatch: Stock Market News
Available from: https://www.marketwatch.com/press-release/autonomous-navigation-market-growth-analysis-by-size-share-news-demand-opportunity-during-2019-2025-2020-03-04
Accessed 28 April 2020.
[2] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 3-6. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[3] R. F. Araújo, A. Ribeiro, I. V. Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. p. 1
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[4] R. F. Araújo, A. Ribeiro, I. V. Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 1-2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[5] R. F. Araújo, A. Ribeiro, I. V. Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 1-2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[6] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 1-2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[7] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 2
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[8] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 3
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[9] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. pp. 1-2
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.

[10] S. S. Rao, Engineering optimization: theory and practice – 4th ed. (2009). p. 1
[11] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 1-2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[12] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 1-2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[13] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 1-2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[14] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[15] A.H.G. Rinnooy Kan, G.T. Timmer, Chapter IX Global optimization, Handbooks in Operations Research and Management Science, Elsevier, Volume 1, (1989), pp. 631-662
Available from: https://doi.org/10.1016/S0927-0507(89)01010-8
Accessed 15 February 2020
[16] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 6
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[17] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 6
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[18] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 6
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[19] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[20] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.

[21] A. Biere, "Bounded model checking," in Handbook of Satisfiability. IOS Press, 2009, p. 458.
[22] A. Biere, "Bounded model checking," in Handbook of Satisfiability. IOS Press, 2009, p. 458 - 460.
[23] A. Biere, "Bounded model checking," in Handbook of Satisfiability. IOS Press, 2009, p. 462.
[24] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. (2003) Bounded Model Checking. 2003. pp. 2 - 4
Available from:
http://www.cs.cmu.edu/~emc/papers/Books%20and%20Edited%20Volumes/Bounded%20Model%20Checking.pdf
Accessed 20 October 2020
[25] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[26] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[27] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[28] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. (2003) Bounded Model Checking. 2003. p. 9
Available from:
http://www.cs.cmu.edu/~emc/papers/Books%20and%20Edited%20Volumes/Bounded%20Model%20Checking.pdf
Accessed 20 October 2020
[29] A. Biere, A. Cimatti, E. M. Clarke, O. Strichman, Y. Zhu. (2003) Bounded Model Checking. 2003. p. 9
Available from:
http://www.cs.cmu.edu/~emc/papers/Books%20and%20Edited%20Volumes/Bounded%20Model%20Checking.pdf
Accessed 20 October 2020
[30] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[31] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 7
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.

[32] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[33] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[34] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[35] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 10
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[36] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 8
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[37] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 1
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[38] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 1
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[39] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 11
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[40] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 10
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[41] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. p. 10
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.

[42] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. pp. 9 - 10
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[43] R. F. Araújo, H. F. Albuquerque, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimization based on Satisfiability Modulo Theories. 2017. pp. 9 - 10
Available from: https://ssvlab.github.io/lucasccordeiro/papers/jscp2017_2.pdf
Accessed 28 September 2019.
[44] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[45] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 3-10. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[46] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 3. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[47] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 3-18. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[48] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 5-11. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[49] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 5-11. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.

[50] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 11 -13. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[51] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 11 - 13. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[52] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 13-14. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[53] Yang, Liang & Qi, Juntong & Song, Dalei & Xiao, Jizhong & Han, Jianda & Xia, Yong. (2016). Survey of Robot 3D Path Planning Algorithms. Journal of Control Science and Engineering. 2016. pp. 15-17. 10.1155/2016/7426913.
Available from:
https://www.researchgate.net/publication/304813875_Survey_of_Robot_3D_Path_Planning_Algorithms
Accessed 10 October 2019.
[54] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2 - 4
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[55] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 2 - 4
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[56] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 3
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[57] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 3
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.

[58] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 4
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.
[59] R. F. Araújo, A. Ribeiro, I. V. de Bessa, L. C. Cordeiro, J. E. C. Filho. (2017). Counterexample Guided Inductive Optimisation Applied to Mobile Robots Path Planning (Extended Version). 2016. pp. 4
Available from: https://arxiv.org/pdf/1708.04028.pdf
Accessed 10 September 2019.