LIVERPOOL JOHN MOORES UNIVERSITY

Department of **Electronics & Electrical Engineering**

Webots-Based Implementation and Simulation of Robotics Algorithms

Dissertation

Pablo Suárez Sánchez

Supervisor: **Princy Johnson**

2nd Marker: **Ronan McMahon**

6455ELE 2023-24

Acknowledgements

This project could not have been accomplished without the support and encouragement of various people, whose academic and moral contributions have been invaluable in its completion.

First and foremost, I am deeply grateful to my family for their understanding, encouragement, and patience during all this years. Their unwavering belief as kept me always motivated.

I am equally grateful for the support of my friends, whose understanding and appreciation have been invaluable.

Special recognitions and sincere appreciations go to Antonio M. López and Princy Johnson, my supervisors, for their contributions and valuable guidance. Their expertise and mentorship have been essential in shaping the direction and quality of this project.

Lastly, I also extend my gratitude to myself for the dedication, perseverance and hard work that have been instrumental during this last four years.

Pablo

Abstract

In the vast realm of robotics, the implementation of different behaviours relies on sophisticated and well-stablished techniques. This project will embark on the review, implementation, and simulation of various algorithms and techniques used in general robotics.

The main focus of the project will be the simulation, analysis, and validation of these, using the Webots open-source software. More specifically, the project will focus on the development of navigation algorithms like trajectory following (Pure Pursuit), sensor-based approaches like Bug algorithms or Artificial Potential Fields (APF), as well as map-based navigation including Grassfire, Dijkstra, and A*. These last algorithms will be supported by map generation techniques such as Probabilistic Roadmaps generation or RRT (Rapidly Exploring Random Trees) algorithms.

By employing a professional simulation software like Webots, this endeavour seeks to explore a platform for in-depth examination and experimentation on robotics algorithms.

This work does not aim to develop or research into new or innovative solution, as it's mainly focused onto reviewing and acquiring a greater understanding of these kind of technologies, widely demanded on the actual context of implementing automation and robotics in nearly any field.

Resumen

Introducción

En el contexto tecnológico actual, la robótica y otros sistemas autónomos han adquirido una gran relevancia. Esta realidad ha producido una gran demanda en el desarrollo e integración de soluciones innovadoras referidas a los algoritmos y técnicas que controlan y permiten el correcto funcionamiento de estos nuevos sistemas, aumentando su capacidad de navegación, planificación, localización y mapeado tan necesarias para este tipo de aplicaciones.

Este proyecto explorará los entresijos de los sistemas que controlan el funcionamiento de algunos sistemas robóticos, centrándose especialmente en el desarrollo de estos algoritmos y su simulación en Webots, un software que proporciona un entorno ideal para el análisis y desarrollo de este tipo de técnicas.

Motivación

Dado el contexto industrial y comercial actual, existe una gran demanda de sistemas robóticos y autónomos, y en consecuencia, del desarrollo de las tecnologías de control de estos. La capacidad de estos sistemas para navegar, mapear o interactuar con su entorno es crucial en ciertas aplicaciones, por lo que resulta esencial explorar la gran variedad de algoritmos y otras técnicas disponibles para abordar esto.

También es estrictamente necesario analizar las diferencias entre los cálculos y el funcionamiento teórico, y una aplicación real. Aquí es donde los recursos de simulación entran en juego, proporcionando un entorno virtual en el que trasladar los conceptos teóricos a un funcionamiento realista y fiel a las imprecisiones del mundo real.

Por último, como una motivación más personal, este proyecto combina mi gran interés por la robótica y las nuevas tecnologías, con su aplicación en el contexto ingenieril actual. Resulta esencial adquirir conocimientos en esta área, a la hora de adentrarse en el contexto laboral de la ingeniería moderna.

Objetivos

El objetivo principal de este proyecto es la implementación y simulación de algoritmos fundamentales, y su simulación en la plataforma Webots. Estos algoritmos incluyen técnicas de navegación, planificación, localización y mapeado. Llevando a cabo diferentes simulaciones de cada uno de estos, el proyecto trata de evaluar el rendimiento y validez de estos algoritmos en un entorno realista.

El objetivo principal puede desglosarse en una serie de metas más específicas:

a. Revisión y análisis de las técnicas existentes para navegación, evasión o esquiva de obstáculos, planificación de rutas y mapeado
b. Diseño e implementación de algoritmos de navegación para adoptar diferentes posiciones, seguimiento de trayectorias, y mecanismos de evasión de obstáculos basados en estrategias “Bug” o campos potenciales
c. Diseño e implementación de técnicas de planificación de rutas (como Dijkstra y A*), adaptado para entornos de simulación en Webots
d. Diseño e implementación de algoritmos de generación de mapas probabilísticos o mapas RRT
e. Evaluación de los algoritmos implementados y resultados obtenidos

Contribución

Este proyecto no es realmente una solución innovadora para la aplicación de todas estas tecnologías. Su valor reside en la investigación, revisión, comparación y contextualización de los diferentes algoritmos de robótica existentes, y su posterior contribución a comprender sus fortalezas y debilidades, además de cómo varía su comportamiento en diferentes escenarios.

La simulación de estos algoritmos proporcionará datos válidos a la hora de evaluar el rendimiento y viabilidad de cada solución para, en un futuro, poder implementarla sobre un robot real.

Por último, este trabajo define un buen marco inicial para futuro desarrollo o investigaciones, ya que explora una gran variedad de algoritmos, estableciendo una sólida base sobre la que desarrollar trabajos futuros.

Robot e-puck

El robot utilizado para implementar y simular todos los algoritmos es el robot “e-puck”. Se trata de un pequeño robot diferencial, de software y hardware abiertos, y muy extendido entre la comunidad educativa y científica. A su vez se trata de un robot muy económico y robusto, lo que lo hace muy adecuado para el ámbito educacional.

Algunas de sus características principales son:

- Pequeño tamaño y peso
- Diseño basado en ruedas diferenciales, permitiendo un control preciso y sencillo
- Variedad de sensores y posibilidad de expansión
- Potencia de procesamiento y capacidad de comunicación
- Uso y programación sencillos
- Amplia documentación y soporte de la comunidad
- Muy asequible y fácil de encontrar
- Muy extendido entre numerosos simuladores de robótica, siendo Webots uno de ellos

Software de simulación Webots

Webots es un software profesional de simulación de robótica de código abierto, desarrollado por Cyberbotics. Este software ofrece un ecosistema de simulación relativamente “potente”, versátil y amigable para un usuario principiante, a la hora de simular distintos robots y diseñar los entornos de prueba. Esto lo convierte en una elección idónea para un proyecto como este.

Este simulador admite la programación de controladores para el robot en numerosos lenguajes como C/C++, Python, Java o MATLAB. En este caso, debido a la facilidad para exportar los datos a la propia suite de MATLAB, y la cantidad de funciones y operadores matemáticos útiles que éste tiene integrados, el lenguaje utilizado será MATLAB.

El programa ofrece algunas características muy interesantes como:

- Motor de físicas avanzado
- Extensas librerías de robots, sensores y otros
- Editor intuitivo del entorno de simulación
- Integración de ROS
- Simulación en tiempo real

Algunas de las ventajas de este simulador frente a otros como CoppeliaSim o Gazebo, residen en su naturaleza de software de código abierto (gratuito), funcionalidad combinada con la facilidad de uso, eficiencia de las simulaciones en tiempo real y su fidelidad con la realidad.

Algoritmos de movimiento simple

La implementación de los algoritmos de movimiento fundamentales de un robot, son esenciales para permitir que un robot terrestre se desplace de manera efectiva y precisa. El modelo básico utilizado habitualmente para modelar esto es el modelo de la bicicleta. Con este modelo, el robot posee 2 ruedas: una rueda trasera fija, y una delantera que permite el giro sobre el eje vertical para dirigir el movimiento. Este modelo resulta muy útil, al simplificar el movimiento del robot y con ello facilitar la planificación de la trayectoria a seguir.

El algoritmo de movimiento más simple es el de moverse a un punto específico del plano. Este tipo de algoritmo, implementa dos controladores proporcionales que ajustan la velocidad y la dirección del robot hacia el punto objetivo.

Una vez conseguida la implementación de esto, se puede avanzar hacia movimientos algo más complejos, como el de seguir una línea o trayectoria.

El movimiento más general será entonces el de seguir una trayectoria, siendo el algoritmo más clásico utilizado con este fin el “Pure Pursuit”. Este algoritmo se basa en mover un punto objetivo ficticio a lo largo de la trayectoria a seguir, utilizando una velocidad constante; a su vez, se indica al robot que debe seguir este punto. En este caso, los controladores a utilizar son uno proporcional, que controla la velocidad de las ruedas del robot en relación con la distancia de este con el punto objetivo; y uno

integral, que controla la orientación y ángulo de giro de este, en relación con este punto virtual.

Se presentan y evalúan esto algoritmos, “tuneando” los controladores y probando distintos valores para las variables de control, analizando cómo varían los resultados obtenidos con cada uno de estos, y cuáles generan un movimiento más preciso y eficiente.

Navegación reactiva (basada en sensores)

Los algoritmos de navegación reactiva son una de las técnicas más utilizadas en navegación robótica. Esto permite a un robot ir desde un punto inicial A hacia un objetivo B sin necesidad de saber cómo se distribuyen los obstáculos a su alrededor. Estos algoritmos aprovechan la capacidad de sus sensores para adaptar la dinámica de su movimiento en tiempo real y evitar colisionar con algún obstáculo, permitiendo así navegar en espacios relativamente complejos y brindando también a robot la capacidad de moverse a través de un entorno en constante evolución (con obstáculos móviles, por ejemplo). En este trabajo se revisan algunas de las técnicas más clásicas, aunque también ofreciendo un pequeño contexto sobre los algoritmos más complejos.

Uno de los casos a tratar, y entre los más utilizados, son los algoritmos “Bug”, en los que se asume la manera más sencilla de movimiento. El robot se mueve en línea recta hacia el punto objetivo hasta que un obstáculo es detectado en el camino, en cuyo caso, el robot, dependiendo del tipo de algoritmo implementado, circunnavegará el obstáculo evitándolo, hasta encontrar de nuevo una trayectoria viable hacia el objetivo. Existen distintos tipos de algoritmos “Bug”, dependiendo de las decisiones y posibilidades que baraja el robot, y la complejidad de las soluciones que sea capaz de encontrar. En este caso, el trabajo se centra sobre el algoritmo “Bug2”.

Por otro lado, existen otras soluciones más complejas en cuanto a navegación reactiva, como la navegación basada en campos potenciales. En este caso, la idea es la de simular el comportamiento de una partícula cruzando un campo potencial, de

manera que los obstáculos simulen un comportamiento repulsivo, y el punto objetivo como el potencial atractivo del campo. Esta solución es compleja en cuanto a su adaptación al entorno donde se utilice, ya que los valores necesarios para su correcto funcionamiento son muy dependientes del problema, además de la necesidad de que estos se ajusten muy precisamente (el mínimo cambio erróneo puede desequilibrar completamente el funcionamiento del robot).

De nuevo se presentarán y evaluarán estos algoritmos, cambiando en cada caso las variables de control y analizando cada uno de los resultados obtenidos. Además, en el caso de los algoritmos “Bug”, la decisión de circunnavegar un obstáculo por el lado derecho o izquierdo puede determinar la capacidad del robot de navegar por el entorno dado o impedirlo, por lo que se evaluará el efecto de la toma de una decisión u otra en entornos y disposiciones de obstáculos distintas.

Navegación basada en mapas

La navegación basada en mapas es una pieza fundamental en muchas aplicaciones de robótica móvil. Estos algoritmos se apoyan en diferentes tipos de mapas y representaciones del entorno, para calcular las rutas óptimas, evitando obstáculos y alcanzando el destino específico de manera eficiente.

Es crucial entender los tipos de mapas a utilizar al aplicar estas técnicas de búsqueda. Habitualmente, los mapas métricos son demasiado complejos y demandantes computacionalmente, por lo que suelen transformarse en otro tipo de mapas. Los mapas de rejilla reducen la capacidad computacional requerida, dividiendo el entorno en una cuadrícula de celdas. También existen los mapas topológicos, que ofrecen una representación más abstracta, centrándose en la conectividad y relación entre ubicaciones clave del mapa, y dejando de lado los detalles geométricos precisos.

En casos como los de los mapas de rejilla, encontrar una ruta implica explorar las celdas adyacentes, utilizando algoritmos sencillos como el “Grassfire” para determinar el camino más eficiente. Esto se puede aplicar a problemas sin mucha complejidad, como la resolución de un laberinto.

Por otro lado, los mapas topológicos permiten implementar algoritmos más complejos y utilizados en muchas disciplinas, como el “Dijkstra” o el “A*”, que combina la eficiencia del primero con la implementación de una función heurística que guía la búsqueda de la ruta. Estos algoritmos se aplican en problemas de redes u optimización de la logística y entregas, entre otros.

La implementación de dichos algoritmos en este proyecto, irá estrechamente ligada con la técnicas de generación de mapas. Combinando ambas, con los algoritmos de movimiento presentados anteriormente, el robot será capaz de recibir la información sobre el entorno, generar un mapa adecuado de este, y con ellos buscar la ruta óptima entre su posición y la localización del punto de destino. Una vez planificada la ruta, se ejecutarán los correspondientes algoritmos de movimiento o seguimiento de la trayectoria, para así lograr recorrer el camino calculado.

Estos algoritmos se probarán tanto sobre un mapa sencillo, como el caso de un laberinto con una sola solución; como en otros más complejos, donde existen varios caminos para llegar a una solución, y cada algoritmo deberá decidir cuál es el camino más eficiente. En el caso de “A*”, se evalúa el funcionamiento de distintos heurísticos y cómo afecta la naturaleza de cada uno de ellos al resultado calculado.

Generación de mapas

Los algoritmos de generación de mapas son esenciales en la navegación robótica, ya que las representaciones o mapas que conocemos habitualmente no siempre son adecuadas ni fácilmente computables a la hora de aplicarlas en conjunto con algoritmos de planificación. Para remediar esto, existen diferentes métodos para transformar o discretizar los mapas, haciéndolos más adecuados para los cálculos a realizar.

Habitualmente las representaciones en mapas de rejilla tampoco son muy eficientes, excepto en algunos casos puntuales, especialmente en aquellos donde el mapa es muy detallado, ya que se requerirían infinidad de celdas. La forma más eficiente de

representar un espacio, es mediante un mapa topológico, que permite almacenar la suficiente información utilizando poco espacio y potencia computacional.

A la hora de generar estos mapas topológicos, el objetivo es definir un “mapa de carreteras” del espacio representado, evitando los obstáculos. Estos mapas consisten en una serie de nodos y conexiones, donde cada nodo representa un punto significativo del entorno, y cada conexión es una línea recta que representa un posible camino entre estos puntos.

Existen varios tipos de estos mapas, entre ellos, los mapas probabilísticos o de carreteras probabilísticas. A partir de una configuración vacía, se genera un número de puntos distribuidos aleatoriamente sobre el mapa, descartando aquellos que se encuentran en posiciones no transitables de este. Una vez hecho esto, se generan las trayectorias viables que unen cada par de puntos.

Una vez definido el mapa, es válido para generar trayectorias entre dos puntos cualquiera del mismo, sólo necesitando los puntos de inicio y fin, y utilizando un algoritmo de búsqueda como los mencionados anteriormente (“Dijkstra” o “A*”). El algoritmo es probabilísticamente completo, por lo que si existe una solución para el mapa de carreteras, la encontrará si el número de puntos generados establecido es adecuado.

Por otro lado, es posible generar otro tipo de representaciones, en caso de no ser necesaria la generación de un mapa de carreteras útiles. En este caso la estrategia consiste en crear una ruta directamente desde el nodo inicial, generando un árbol de caminos, hasta encontrar el punto final u objetivo. Este algoritmo se conoce como “RRT” (Rapidly-Exploring Random Tree). También existen otras variantes más eficientes de éste, como el “RRT-2”, que genera dos árboles al mismo tiempo, uno desde el punto de inicio, y otro desde el punto final, hasta que ambos se encuentran.

Estos mapas serán generados en los entornos de prueba de los algoritmos de navegación basada en mapas o planificación de rutas. Como se ha mencionado anteriormente, el objetivo es el de combinar ambas técnicas, proporcionando al robot la capacidad de analizar el mapa, y decidir cuál es el camino a tomar para llegar a un objetivo determinado.

Análisis y presentación de los resultados

Todos los datos y resultados de la simulación de cada algoritmo y técnica implementados en Webots, son exportados a la suite de MATLAB, donde el análisis de los datos obtenidos, y la posibilidad de crear gráficos y representaciones de los resultados es mucho más sencillo y vistoso.

Para cada uno de los algoritmos, se seleccionarán una serie de estadísticas y parámetros relevantes, que permitan clasificar y comparar el rendimiento de cada uno de ellos.

Conclusiones y trabajo futuro

En el desarrollo de este proyecto se ha logrado implementar y simular una significativa variedad de algoritmos fundamentales para la navegación, planificación, localización y mapeado utilizado en robots autónomos, en la plataforma Webots. A través de estas simulaciones, se ha podido evaluar el rendimiento y la validez de cada una de estas soluciones, en distintos entornos controlados. Las principales conclusiones obtenidas son:

- Revisión y comparación de los algoritmos, identificando algunas de sus fortalezas y debilidades, así como su aplicabilidad en diferentes escenarios
- Implementación y simulación en Webots, que permitió la programación y ejecución de cada algoritmo en un entorno personalizado, para evaluar correctamente el rendimiento y comportamiento de cada uno
- Resultados de la simulación, que proporcionan una visión clara sobre la eficiencia de cada uno de ellos. Se utilizó MATLAB para analizar y visualizar estos datos, permitiendo una comparación basada en distintas métricas de rendimiento
- Investigación en el marco de la robótica y base para trabajos futuros, aportando un pequeño marco de referencia para el futuro desarrollo de alguno de estos algoritmos, y estableciendo una base sólida de datos, que podrían ser utilizados para una posterior implementación en un robot real

Por otro lado, surgen otros puntos sobre los que trabajar en un futuro como:

- La mencionada implementación en un robot real
- La optimización del funcionamiento de algunos de los algoritmos
- Mejoras y correcciones en el errático funcionamiento de algunas de las implementaciones
- Integración de algunas de las nuevas tecnologías de IA o “machine learning”
- Ampliación del entorno de simulación, con nuevos y más complejos escenarios de prueba

Siguiendo estas directrices futuras, no sólo se podría conseguir mejorar los resultados obtenidos con este proyecto, sino que se contribuiría al avance general del campo de la robótica en estos ámbitos, buscando soluciones más eficientes y efectivas para aplicaciones en el mundo de la ingeniería real.

Table of contents

1. Introduction

1.1. Introduction

In the actual context of ever-evolving technology; robotics and autonomous systems have been given a huge relevance. This landscape has demanded the design, development and integration of innovative algorithms and techniques to power these new technologies, in order to elevate the understanding of navigation, planning, localization and mapping needed for these applications.

This project will embark on a journey into the heart of robotics algorithms, exploring the different techniques used, and focusing on its development in the Webots simulation platform.

The ability of a robot to be able to navigate and interact with static or dynamic environments is essential. The Webots platform as a simulation tool, provides us with the right canvas to develop these technologies and test theoretical algorithms into real-world applications, including realistic and 3D visual attractive simulations with a user-friendly interface, as it already has many built-in features as shapes, robots and sensors which will make the simulation tasks much manageable.

1.2. Motivation

The projects core and main objective is to explore the insights and complexities of robotics and autonomous systems. In the actual market, there's a big demand for this kind of products, and the development of its control technologies.

Given the actual context, the ability of robotics systems to navigate, plan, map and interact with a dynamic environment is crucial. It is essential to explore the variety of algorithms and techniques available to overcome these, but also the differences between these theoretical maths and real applications.

To bridge this gap and to be able to develop robust algorithms, it's crucial to simulate them on a proper environment, which the Webots platform provides us with. This kind

of virtual environments help us translating theoretical concepts into realistic simulations, while providing us with the data and statistics needed to evaluate these systems.

As a more personal motivation, this work involves my deep interest in robotics and its applications in the actual engineering. I can recognize the significant value of acquiring knowledge in this field, especially within my university, which actively researches in robotics and automation applications.

1.3. Aims and objectives

Aim:

Implement and simulate fundamental robotics algorithms on the Webots platform. These algorithms will include navigation, planning, localization, and mapping techniques. By conducting simulations in various complex scenarios, the project aims to evaluate the performance of these algorithms, generate detailed maps of simulated environments, and analyse the accuracy of the localization methods, providing valuable insights into the effectiveness of these algorithms in realistic navigation situations.

Objectives:

a. Review of techniques for navigation, obstacle avoidance, route planning, and mapping in an indoor environment.
b. Design and implementation of navigation algorithms for reaching predetermined poses, path tracking, mobile object tracking (pure pursuit), and obstacle avoidance mechanisms based on "Bug" strategies and Potential Fields.
c. Design and implementation of route-planning algorithms like Dijkstra and A*, adapted for simulated environments in Webots.
d. Design and implementation of Algorithms for generating probabilistic maps or RRT paths.
e. Evaluate the mapping and navigation system using benchmark parameters.

1.4. Contribution

This project is not really an innovative solution for these technologies, its value relies more into the review, comparison, and contextualization of different existing robotics algorithms. The real contribution stands in understanding the strengths and weaknesses of each solution, and how can these perform on different scenarios.

On the other hand, Webots based simulations will provide realistic data, in order to assess each solution's performance and suitability in different scenarios.

Finally, this project is a good framework for future research, as it will cover a good range of robotics planning, mapping, and navigation algorithms, setting up a solid basis upon which to develop future work.

1.5. Report layout

1. Introduction
   - 1.1. Introduction

     Provides a general overview of the project.
   - 1.2. Motivation

     Exploring the reasons behind undertaking this project, highlighting its relevance.
   - 1.3. Aim and Objectives

     States the goals and objectives that this project aims to achieve.
   - 1.4. Contribution

     Highlights the potential contribution of this project to robotics and engineering.
   - 1.5. Report layout

     Preview of the structure of the report.
2. Literature review
   - 2.1. Algorithms

Survey of existent literature and giving theoretical context to each algorithm reviewed.

2.2. Software platforms

Review of the software selected for the project, and comparison with other options.

2.3. Performance metrics

Discussion about some of the metrics used to evaluate each implementation's performance.

3. System Design

   Design considerations and decisions made to develop the system.

4. Software implementation

   Detailed description of the practical implementation of each technique.

5. Results and discussion

   Presenting the results obtained and discussion about them.

6. Conclusion and future work

   Summarize key findings, results, and potential future work on the topic.

2. Literature review

Providing a proper context for the project, as well as reviewing previous work done on the same topic is one of the main objectives of this project. This review will be structured in three main sections.

Firstly, emphasis will be placed into the theoretical framework of the project, the robotics algorithms themselves. These algorithms will include navigation, obstacle avoidance, mapping, and route planning. Given the already extensive work done on this topic by various researchers, the intention will be getting a proper context from previous work done.

Following this, the system design and simulation platform chosen will be reviewed, the Webots software, where all these mentioned theoretical algorithms will be tested further on. This software will be evaluated against some other potential alternatives,

giving a proper analysis on its features, user-friendly interface, and accuracy of its simulations.

Finally, in order to perform accurate tests for each algorithm, some benchmark metrics need to be set. By reviewing some similar works to this project, some accurate and significant performance metrics will be set, in order to evaluate our simulations further on the project.

2.1. Algorithms

There a wide range of algorithms and techniques being reviewed on this project. This will be classified in four main types, depending on the function they perform.

2.1.1. Simple movement control

When talking about robot movement, the first thing is to define how a robot moves with some equations, and then introduce the simplest movements a robot can achieve.

The simplest archetype for most ground robots is the wheeled car. The car's movement is usually represented with the bicycle model in which a bicycle has a rear wheel fixed to the body, and the front wheel rotates about the vertical axis to steer the vehicle. The position of a robot on Cartesian planes can be illustrated by Figure 1. The world coordinate is shown in blue, and the vehicle's coordinate axis in red. γ is the steering angle of the car, and v is the velocity of the back when in the x-direction (Corke, 2017).

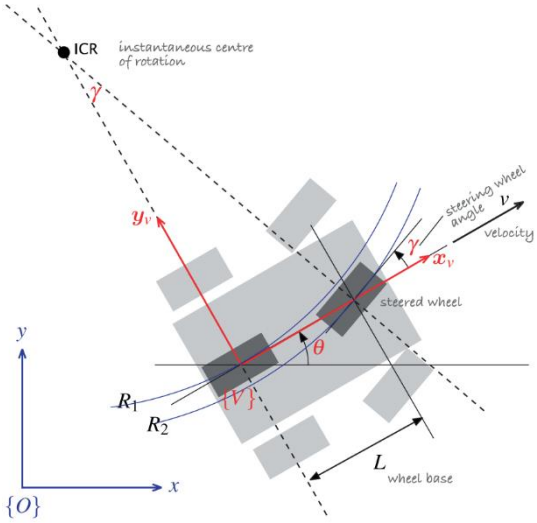

Figure 1 - Bicycle-based model of a vehicle's motion. From Peter Corke, Robotics, Vision & Control, Springer, 2nd Edition, 2017.

With all of this, the motion equations for the robot in the world frame can be written as shown in Equation 1.

$$\dot{x} = v\ \cos\theta$$
$$\dot{y} = v\ \sin\theta$$
$$\dot{\theta} = \frac{v}{L}\ \tan\gamma$$

Equation 1. Robot's basic motion equations (Corke, 2017).

Once defined the robot's motion equations and characteristics, the simplest moving algorithm can be introduced.

Moving to a specific point on the plane is the simplest navigation algorithm we can implement. Considering a goal point in the plane:

$$P^* = [x^*, y^*]^T$$

Equation 2 - Robot's goal point in the plane (Corke, 2017).

The robot's velocity is controlled by applying a proportional action to its distance to the goal:

$$v^* = K_v\sqrt{(x^* - x)^2 + (y^* - y)^2}$$

Equation 3 - Proportional control algorithm for the robot's velocity (Corke, 2017).

Then to steer towards the goal, which is at the vehicle-relative angle:

$$\theta^* = \tan^{-1}\frac{y^* - y}{x^* - x}$$

Equation 4 – Vehicle's relative angle to steer towards the goal (Corke, 2017).

Another proportional control algorithm is used:

$$\gamma = K_h(\theta^* - \theta)$$

Equation 5 - Proportional control algorithm for the vehicle's steering angle (Corke, 2017).

Once moving to a point on the plane is achieved, we can model more complex movements in the plane, being the next in nature, following a line.

A line on the plane is defined by:

$$ax + by + c = 0$$

Equation 6 - General equation for a line in the plane (Corke, 2017).

Following this line will require two controllers to be able to adjust the steering. The first one steers the robot to minimize its normal distance from de line, which is:

$$d = \frac{(a, b, c) \cdot (x, y, 1)}{\sqrt{a^2 + b^2}}$$

Equation 7 - Normal distance between the robot and the line (Corke, 2017).

The proportional controller turning the robot towards the line would be:

$$\alpha_d = -K_d d$$

Equation 8 - Proportional control algorithm to turn the robot towards the line (Corke, 2017).

The second controller would adjust the heading angle (orientation) of the vehicle, in order to be parallel to the line followed:

$$\theta^* = \tan^{-1}\frac{-a}{b}$$
$$\alpha_h = K_h(\theta^* - \theta)$$

Equation 9 - Steering angle needed and control law to adjust the robot's orientation (Corke, 2017).

Finally, the combined control law would be:

$$\gamma = \alpha_d + \alpha_h = -K_d d + K_h(\theta^* - \theta)$$

Equation 10 - Combined control algorithm to follow a line (Corke, 2017).

Finally, having already reviewed the algorithms and equations needed to move a robot to a specific point, or follow a line, there's the final basic movement a robot should be capable of doing.

Following a path or trajectory is a more general and common movement for a robot, instead of following a line, for example, a trajectory could be generated by a motion or path planned and sent to the motion control algorithm of the robot, which should be capable of execute this movement precisely. The classic algorithm used to implement this is named pure pursuit, which is based on moving the goal point to reach along the path in a constant speed, and then make the robot follow that point, achieving this movement through a specified path.
The approach to this is similar to moving to a point, despite the fact that the point is moving this time (Universidad de Oviedo, s.f.). An error is calculated with the distance maintained between the robot and the pursuit point:

$$e = \sqrt{(x^* - x)^2 + (y^* - y)^2} - d^*$$

Equation 11 - Distance error between the pursuit point and the robot (Corke, 2017).

This error is regulated to zero by controlling the robot's velocity, this is done by designing a proportional-integral controller (PI):

$$v^* = K_v e + K_i \int e \, dt$$

Equation 12 - PI controller for the robot's velocity (Corke, 2017).

The second controller now steers the robot towards the target at the relative angle θ*, implementing the following:

$$\theta^* = \tan^{-1} \frac{y^* - y}{x^* - x}$$
$$\gamma = K_h(\theta^* - \theta)$$

Equation 13 - Relative angle and steering angle control algorithm (Corke, 2017).

2.1.2. Reactive Navigation (Sensor-Based)

Reactive navigation algorithms are among the most used techniques, in order to make a robot go from A to B without knowing how obstacles are distributed in his path. This kind of algorithms are able to adapt to a real-time dynamic environment, responding to obstacles and changes. The sensors equipped on robots managed like this, make them able to navigate through complex and dynamic spaces with some agility and responsiveness, depending also on the complexity and capacity of the algorithm used. This project will review some of these techniques, the most basic and classic ones, but also giving some context to more advanced ones.

Bug strategies

Bug strategies are among the earliest and simplest sensor-based algorithms with probable guarantees (Yufka & Parlaktuna, 2009). These algorithms assume that the robot is a point in the plane and use its sensors (usually a contact or zero range one) to try finding the path from the initial point to the desired destination.

Bug strategies are based on the simplest assumption made when thinking about moving from A to B. Its navigation approach consists of moving towards the destination point until an obstacle is encountered, in which case, the robot would circumnavigate it until motion towards the goal is again possible.

The Bug0 algorithm is the most basic one of these. It consists of getting the robot orientated to the objective point, move towards it until an obstacle is found; then just circumnavigate it until the robot is capable of reorientating towards the objective again, and continue until reaching the objective point.

A pseudocode for getting this algorithm implemented would be like this:

```
1:   While not at goal location do
2:         If hit an obstacle then
3:               Follow obstacle turning to the left
4:         Else
5:               Drive towards goal location
6:         End if
7:   End while
```

This would lead to a result similar to this:

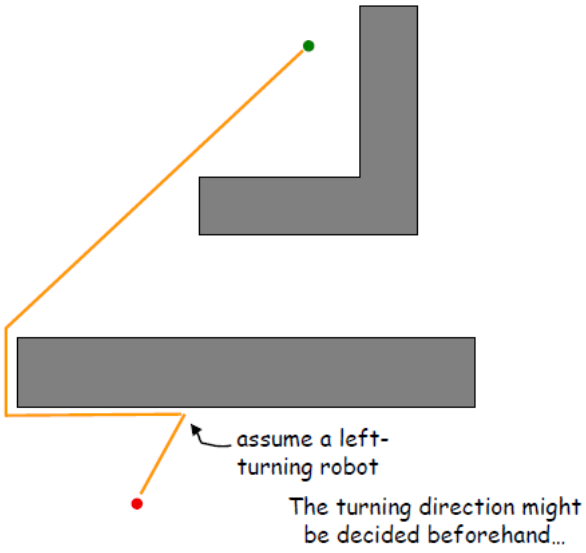

Figure 2 - Result of the movement using the Bug0 algorithm. From Howie Choset, G.D. Hager, and Z. Dodds, Robotic Motion Planning: Bug Algorithms slides, CMU School of Computer Science, 2010.

In the example given, the left-turning robot option has been selected. Navigating obstacles in the opposite direction would be completely valid, but depending on the starting point, or how the obstacles are displayed, right or left turning robots would give different results and efficiencies, but there's no "written rule" on which one to choose.

Obtaining a much more efficient bug algorithm which improves the last one is quite simple, and consists of adding some memory, remembering the path taken.

The Bug1 algorithm is essentially, an improvement of his little brother Bug0. It consists in the robot heading towards the goal, until an obstacle is encountered. Once this happens, the robot circumnavigates the complete obstacle, remembering the start or first "hit" point of the obstacle, and also logging the distance from each point of the obstacle's perimeter to the goal location. Once the robot gets again to the hit point, it navigates wall-following the obstacle again to the point the closest to our objective location. When reaching this point, it heads again towards the goal, repeating this process if another obstacle is encountered.

This can be again implemented following this pseudocode:

```
1:   While not at goal location do
2:         If hit an obstacle then
3:               Log hit point (x)
4:               Follow obstacle turning to the left
5:               While not at x do
6:                     Follow obstacle turning to the left
7:                     Log minimum distance to goal point (n)
8:               End while
9:               Go to n
10:        Else
11:              Drive towards goal location
12:        End if
13:  End while
```

This would result in a movement like this:

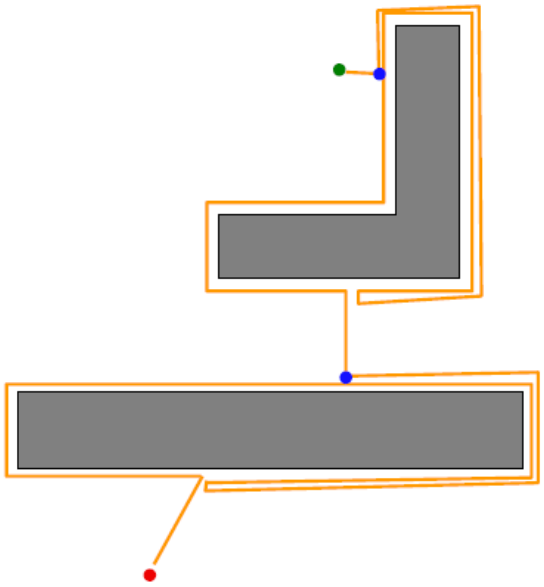

Figure 3 - Result obtained from applying the Bug1 Algorithm. From Howie Choset, G.D. Hager, and Z. Dodds, Robotic Motion Planning: Bug Algorithms slides, CMU School of Computer Science, 2010.

Again, the left-turning robot has been represented, but as before, the other option is completely valid, but in this case the result would be almost the exact same for both turn directions.

There's another evolution for this technique, the Bug2 algorithm. This one starts from the Bug1, improving its performance and efficiency in most of the cases.
This one is based on m-line techniques; it consists of calculating a line (m-line) from the origin to the objective point. The robot moves along this line until an obstacle is reached, and as in the other ones, it starts to wall-follow it, until the m-line is again encountered in a closer point to the objective, starting again to follow the line, repeating this process if another obstacle is found, until reaching the goal.

The pseudocode for this Bug2 algorithm would be:

```
1:   Calculate m-line between origin point and goal
2:   Set state on Follow m-line
3:   While not at goal location do
4:         If current state = Follow m-line and hit an obstacle then
5:               Log distance from hit point to goal (h)
6:               Set state on Avoiding obstacle
7:         If current state = Avoiding obstacle and m-line
           encountered and (distance to goal < h) then
8:               Set state on Follow m-line
9:         End if
10:        Execute current state
11:  End while
```

This would result in a movement like this, on the same obstacle layout example as the other ones:

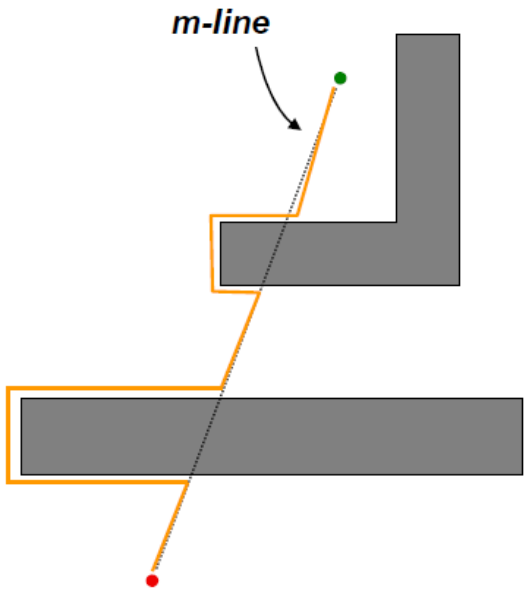

Figure 4 - Result obtained from applying the Bug2 Algorithm. From Howie Choset, G.D. Hager, and Z. Dodds, Robotic Motion Planning: Bug Algorithms slides, CMU School of Computer Science, 2010.

In this case, the Bug2 algorithm outperforms the Bug1, but there are some cases in which this other version works better. This is because both algorithms different nature. Bug1 is an exhaustive search algorithm, what means that it looks at all choices before committing into the best one. On the other hand, Bug2 is a greedy algorithm, as it takes the first choice that looks better than what it had previously. In most of the cases, Bug2 will outperform Bug1, but overall, Bug1 has a more predictable performance.

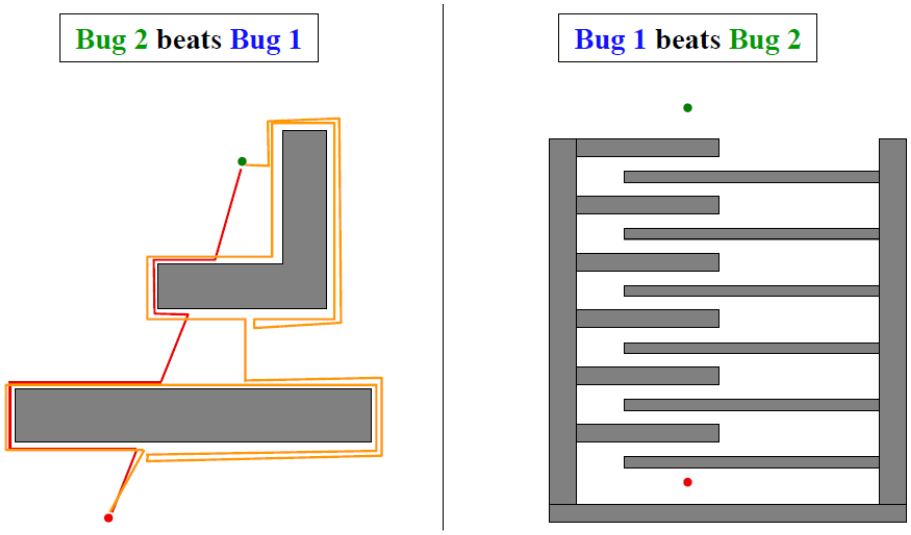

Figure 5 - Comparison between the result obtained in different situations for Bug1 and Bug2 algorithms. From Howie Choset, G.D. Hager, and Z. Dodds, Robotic Motion Planning: Bug Algorithms slides, CMU School of Computer Science, 2010.

Other Bug strategies

There are other similar and realistic but more complex algorithms based on this Bug ones, as this three first ones, are basically contact bug algorithms. Adding a range sensor capable of detecting obstacles from some distance unlocks new more efficient solutions. One of these is the Tangent Bug algorithm.

The Tangent Bug algorithm is another reactive navigation robotics technique, designed to move through unknown environments. Its basic idea is using both motion-to-goal and boundary following behaviours to reach de desired goal. This algorithm dynamically adjusts the movement strategy based on the robots position relative to

the goal and the information given by its range sensors of the obstacles around the machine. It initially moves the robot towards the objective until encountering an obstacle, switching to a boundary-following mode until getting again a line of sight to the goal position.

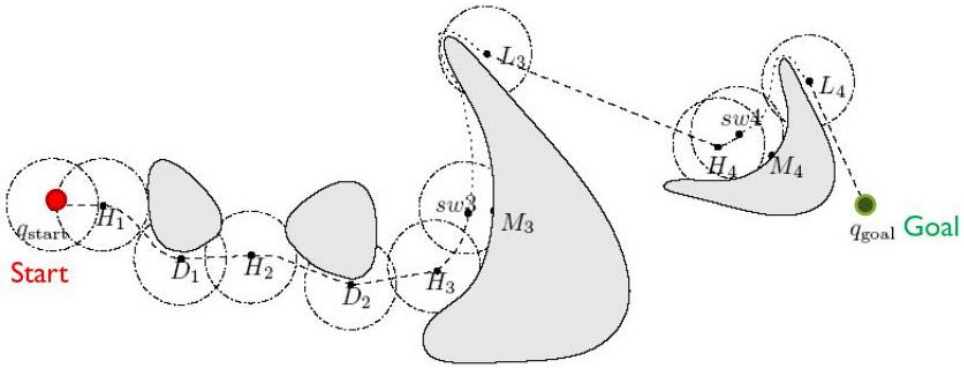

Figure 6 - Example of navigation using a Tangent Bug Algorithm. From Choset, Howie M. Principles of robot motion theory, algorithms, and implementation. Cambridge, Mass. MIT Press, 2005.

There are lots of other more complex algorithms using the same or different strategies to solve the navigation of a robot.

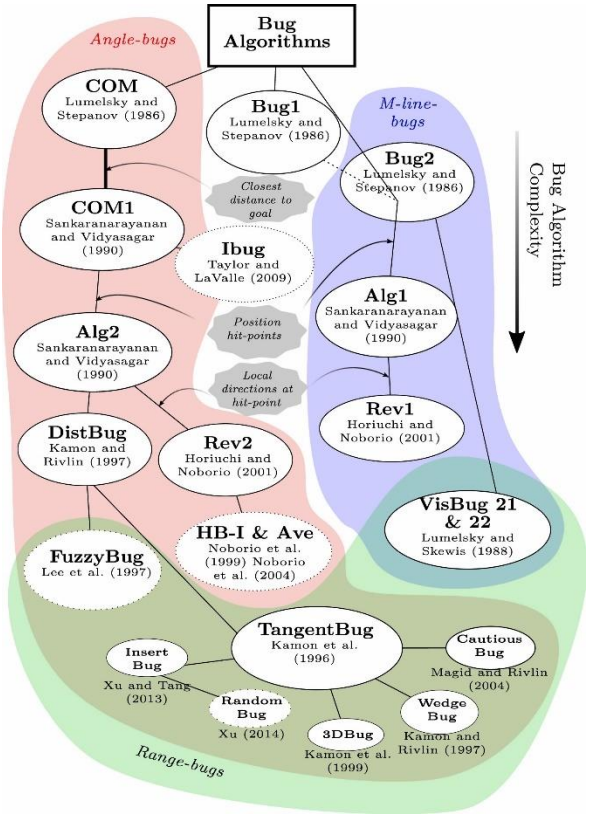

Figure 7 - Overview of Bug algorithms. From K.N. McGuire, G.C.H.E. de Croon, K. Tuyls, Robotics and Autonomous Systems: A comparative study of bug algorithms for robot navigation, Volume 121, 2019.

Artificial Potential Fields

Artificial potential fields are another approach to the reactive navigation algorithms. As its own name says, the idea is to create a function mixing attractive and repulsive "forces", in order to guide the robot to the target position.

First of all, the attractive function. We can generate an attractive force between the robot's position:

$$p^k = [x_k, y_k]^T$$

Equation 14 - Robot's position vector (Universidad de Oviedo, s.f.).

And the objective's location:

$$g = [x^*, y^*]^T$$

Equation 15 - Target point location vector (Universidad de Oviedo, s.f.).

The attractive force model would be modelled following the equations:

$$d(x_k, y_k) = \sqrt{(x^* - x_k)^2 + (y^* - y_k)^2}$$
$$f_{att}(x_k, y_k) = \frac{1}{2} d(x_k, y_k)^2$$

Equation 16 - Distance from the robot's location and the goal, and attractive force model equations (Universidad de Oviedo, s.f.).

Having the robot, the location p^k, this model would return an attractive force to the target location. Looking into a plot of the attractive force function, the robot would be placed on top of the slope, and the force would attract him to the valley, where the goal is located.

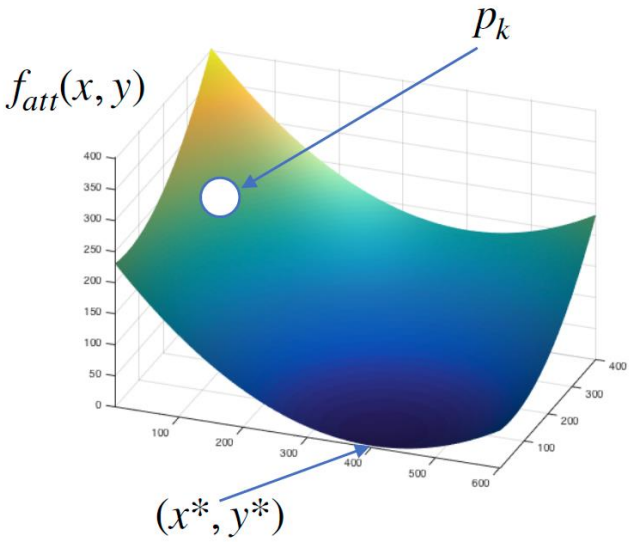

Figure 8 - Example of a potential attractive function. From Intensificación en Sistemas Robóticos slides, Industrial Engineering Master, Universidad de Oviedo.

Each axis component of the gradient function from the potential field would be:

$$\nabla_x f_{att}(x_k, y_k) = \frac{\delta f_{att}(x_k, y_k)}{\delta x_k} = (x_k - x^*)$$

$$\nabla_y f_{att}(x_k, y_k) = \frac{\delta f_{att}(x_k, y_k)}{\delta y_k} = (y_k - y^*)$$

Equation 17 – Attractive potential field gradient function for each axis component (Universidad de Oviedo, s.f.).

Finally obtaining the equations for the control algorithm:

$$x_{k+1} = x_k - \xi \frac{\delta f_{att}(x_k, y_k)}{\delta x_k} = x_k - \xi(x_k - x^*)$$
$$y_{k+1} = y_k - \xi \frac{\delta f_{att}(x_k, y_k)}{\delta y_k} = y_k - \xi(y_k - y^*)$$

Equation 18 - Control algorithm equations for the attractive potential field (Universidad de Oviedo, s.f.).

Where ξ is a scale factor that determines the magnitude of the attractive gradient.

On the other hand, we have the repulsive function, where the idea is to create high repulsive areas on the field, corresponding those to the obstacles the robot can encounter, in order to make the robot stay away from those.
Mathematically, and starting again from the distance equation, this would be modelled as:

$$\rho(x_k, y_k) = \sqrt{(x^* - x_k)^2 + (y^* - y_k)^2}$$
$$f_{rep}(x_k, y_k) = \frac{1}{2}\left(\frac{1}{\rho(x_k, y_k)} - \frac{1}{d_0}\right)^2$$

Equation 19 - Distance to a point and repulsive potential field equations (Universidad de Oviedo, s.f.).

Where d_0 is a parameter that controls the repulsive potential field influence.

Again, calculating each component of the gradient function:

$$\nabla_x f_{rep}(x_k, y_k) = \frac{\delta f_{rep}(x_k, y_k)}{\delta x_k} = \left(\frac{1}{d_0} - \frac{1}{\rho(x_k, y_k)}\right)\frac{1}{\rho(x_k, y_k)^2}\frac{\delta\rho(x_k, y_k)}{\delta x_k}$$
$$\nabla_y f_{rep}(x_k, y_k) = \frac{\delta f_{rep}(x_k, y_k)}{\delta y_k} = \left(\frac{1}{d_0} - \frac{1}{\rho(x_k, y_k)}\right)\frac{1}{\rho(x_k, y_k)^2}\frac{\delta\rho(x_k, y_k)}{\delta y_k}$$

Equation 20 – Repulsive potential field gradient function for each axis component (Universidad de Oviedo, s.f.).

Finally obtaining again:

$$x_{k+1} = x_k - \eta \frac{\delta f_{rep}(x_k, y_k)}{\delta x_k}$$
$$y_{k+1} = y_k - \eta \frac{\delta f_{rep}(x_k, y_k)}{\delta y_k}$$

Equation 21 - Control algorithm equations for the repulsive potential field (Universidad de Oviedo, s.f.).

Where again η is a scale factor to determine the potential field influence.

Combining both potential fields, attractive and repulsive, we obtain the mathematical expression for the complete artificial potential field:

$$x_{k+1} = x_k - \xi \frac{\delta f_{rep}(x_k, y_k)}{\delta x_k} - \eta \frac{\delta f_{rep}(x_k, y_k)}{\delta x_k}$$
$$y_{k+1} = y_k - \xi \frac{\delta f_{rep}(x_k, y_k)}{\delta y_k} - \eta \frac{\delta f_{rep}(x_k, y_k)}{\delta y_k}$$

Equation 22 - Complete equations for the artificial potential field (Universidad de Oviedo, s.f.).

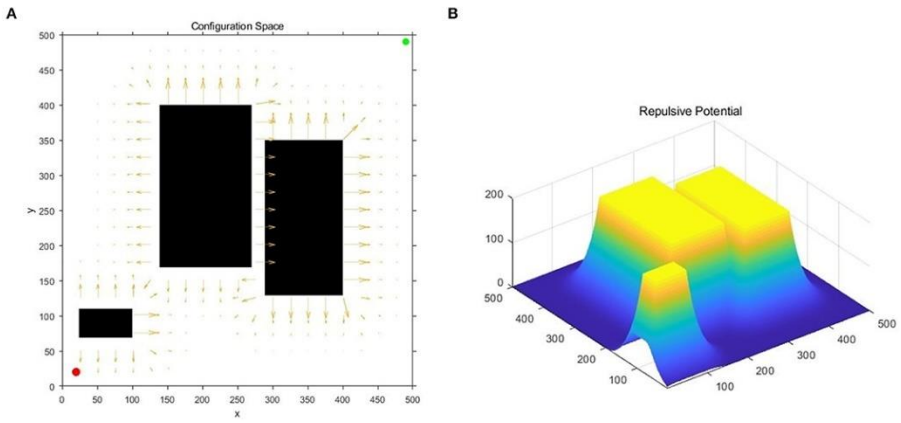

Figure 9 - Artificial potential field example. From Lingjian Ye, Jinbao Chen, Yimin Zhou, Real-Time Path Planning for Robot Using OP-PRM in Complex Dynamic Environment, Shenzhen Institute of Advanced Technology, Chinese Academy of Science, Shenzhen, China, 2022.

This potential field technique has also some weaknesses, as sometimes it does not work as expected. Its main problem is a local minimum on the potential function generated. On these singular points of the function, the effect of the obstacles

(repulsion zones) around the robot, create a local minimum where this repulsive action, counteracts the attractive force from the target, making the robot unable to reach the target.

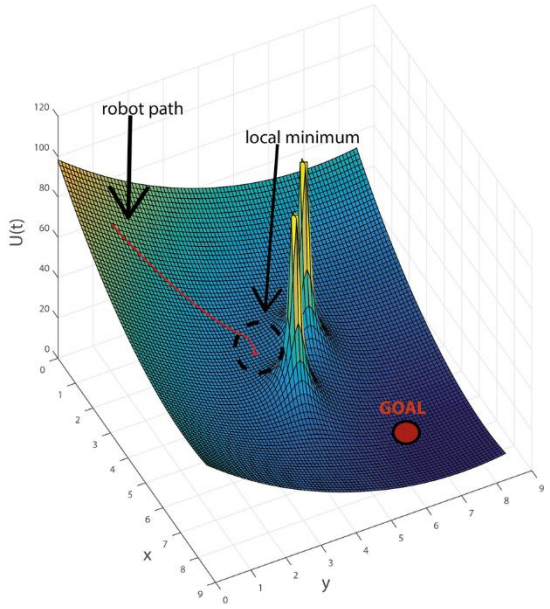

Figure 10 - Example of a local minimum on a potential field function. From Giuseppe Fedele, Luigi D'Alfonso, Obstacles Avoidance Based on Switching Potential Functions, Journal of Intelligent & Robotic Systems, 2018.

There can be some solutions developed to avoid this. The simplest one is to implement a function on the robots control algorithm that detects if the system is on a minimum of the function (looking at the gradient), and it has not reached the objective point, circumnavigate the near obstacle, in order to get the potential function working again. There are also more complex solutions, for example, the one developed by Barraquand and Latombe, called Randomized Path Planner (RPP), that when stuck on a local minimum, the robot initiates a series of random walks that eventually allow the robot to escape this conflictive point.

2.1.3. Map-Based Navigation

Map-based navigation algorithms play an essential role in enabling robots to move around spaces efficiently, avoiding the obstacles represented and reaching a target

location. These algorithms operate on the framework of distinct types of maps and representations of the environment.

After getting to work with map-based navigation, the first thing to do, is to define what type of maps will be used with this control techniques.

Metric and grid maps

This is one of the most common maps we can encounter. This is basically a plain continuous map where each point represented has its own coordinates, giving this an exact localization on this map.
However, when it comes to practical implementation and computational efficiency, these metric maps are not good enough. Transforming these metric maps into grid maps, becomes a crucial step into simplifying the complex geometrical information a metric map gives, into a simpler grid of cells, resulting in a much more computationally and memory manageable map.

Figure 11 - Comparison between metric and grid maps. Pablo Suárez, 2023.

Topological maps

Topological maps are a more abstract and powerful representations of a space. Unlike the ones mentioned before, this kind of representations emphasize into the connectivity and topology between the key locations on the map. These maps are basically a network of interconnected nodes, where each node represents a key point

of interest, and there are some connections between nodes, which are also sometimes tagged with a “cost” of taking that route. This high-level abstraction maps provide a simpler form to compute and calculate navigation and efficient routes between locations, without getting into unnecessary details.

Grid maps navigation

This is one of the most common maps used for this kind of applications. There are some key points needed in order to develop these algorithms:

- The robot can only occupy one of the cells in the map, and it’s able to know or measure in which cell it is.
- The robot can move to any unoccupied adjacent cells.

The most common representation of occupied cells, is to use a Boolean variable meaning:

- `True` if the cell is unoccupied, so the robot can move into it.
- `False` if this cell is occupied by an obstacle.

Navigation based on this kind of maps is simple, the objective is to find a route or path across the grid, having a starting cell and an ending one. There is usually more than one path possible, so the challenge is not only to find one of them, but also to calculate the shortest or most efficient one (that may not be unique).

Grassfire algorithm

The most common algorithm to develop for grid maps is named Grassfire algorithm. This technique is based on searching the most efficient path starting from the target cell and finishing on the start point. The algorithm starts tagging the target cell with a zero value, and immediately starts to iterate tagging each adjacent node with the adequate “cost”. After doing this, this algorithm shows the minimum cost of travelling

from any cell on the grid, to the target. To travel to de end point from one cell, it is sufficient to move to an adjacent node with a lower cost.

This method can easily be understood with its implementation in pseudocode:

```
1:   For each node (n) from the graph:
2:         n.distance = infinite
3:   objective.node.distance = 0
4:   open_list = empty
5:   Add objective.node to open_list
6:   While open_list is not empty do
7:         current_node = 1st element in open_list
8:         Eliminate current_node from open_list
9:         For each node (n) adjacent to current_node do:
10:              If n.distance > (current_node.distance+1)
11:                    Add n at the end of open_list
```

This algorithm is pretty simple to explore the most efficient paths in a grid map, but it also has some problems, as if “cost” of moving from one node to another are not constant, this method does not work at all.

Topological maps navigation

Topological maps navigation algorithms usually permit implementing more complex algorithms, as they are usually more detailed and complex than grid representations. There are much more topological-map exploration algorithms like Dijkstra, A*, Kruskal, Prim and many more.
On the other hand, it’s sometimes also possible to create a topological map based on a grid one, in order to solve grid-map complex problems.

Dijkstra’s algorithm

This algorithm is fundamental in graph theory for finding the shortest paths between nodes in topological maps. This method, systematically explores the graph, assigning

distances to nodes and updating them continually while it navigates thought the map. Finally, the algorithm prioritizes nodes with the shortest distances, outputting the optimal and efficient path between two locations (start and target points).
The algorithm can be implemented with this pseudocode:

```
1:   For every node (n) on the graph do:
2:         n.cost = infinite
3:         n.origin = indefinite
4:   initial_node.cost = 0
5:   open_list = empty
6:   Add initial_node to open_list
7:   While open_list is not empty do:
8:         current_node = open_list element with the least cost
9:         Eliminate current_node from open_list
10:        If (current_node = target_node)
11:              Return OK
12:        For each node (n) adjacent to current_node do:
13:              If n.cost > (current_node.cost +
                 from_current_node_to_n.cost)
14:                    n.cost = current_node.cost +
                       from_current_node_to_n.cost
15:                    n.origin = current_node
16:                    Add n to open_list
```

This algorithm is quite simple and is able to find the most efficient path to go from A to B on a topographic map.

A* algorithm

When exploring grid or topological maps in which each movement or connection has the same cost, Dijkstra's and Grassfire algorithms have similar behaviours and

performance, and give valid and efficient outputs. Both explore each node next to the target, in order to follow the most efficient path. But when it comes to more complex maps or movements, they sometimes fail, giving wrong paths, or just don't work at all. Here is when the A* algorithm appears, as a more complete and efficient solution for this kind of navigations. This method combines the performance of both, adding a heuristic component which guides the planner on its way to the target.
A heuristic function is used to give each node of the map a non-negative value following these equations:

$$\mathrm{h}(\mathrm{objective}) = 0$$

Equation 23 - Heuristic function value for the objective point (Corke, 2017).

Then for each two adjacent nodes x and y:

$$h(x) \leq h(y) + d(x, y)$$

Equation 24 - Heuristic function condition for two adjacent node x and y (Corke, 2017).

Where $\mathrm{d}(\mathrm{x}, \mathrm{y})$ would be the "cost" of going from x to y.

If this heuristic function is consistent, the A* algorithm is optimal and complete. Completion for algorithms, means that it will be able to find the optimal solution if there is one, or in other words, that it does work properly.

The most common heuristics in this kind of planning algorithms usually are:

- Euclidean distance:

$$h(x, y) = \sqrt{(x - x^*)^2 + (y - y^*)^2}$$

Equation 25 - Euclidean distance equation (Corke, 2017).

- Manhattan distance:

$$h(x, y) = |x - x^*| + |y - y^*|$$

Equation 26 - Manhattan distance equation (Corke, 2017).

Being (x, y) the actual point and (x^*, y^*) the target point.

Once reviewed heuristics, the implementation would be based on the following pseudocode:

```
1:   For every node (n) on the graph do:
2:         n.f = infinite
3:         n.cost = infinite
4:         n.origin = indefinite
5:   initial_node.cost = 0
6:   initial_node.f = h(initial_node)
7:   open_list = empty
8:   Add initial_node to open_list
9:   While open_list is not empty do:
10:        current_node = open_list element with the least f value
11:        Eliminate current_node from open_list
12:        If (current_node = target_node)
13:              Return OK
14:        For each node (n) adjacent to current_node do:
15:              If n.cost > (current_node.cost +
                 from_current_node_to_n.cost)
16:                    n.cost = current_node.cost +
                       from_current_node_to_n.cost
17:                    n.f = n.cost + h(n)
18:                    n.origin = current_node
19:                    Add n to open_list
```

This implements the A* algorithm, providing an optimal solution for most of the navigation problems that could be faced.

Replanning

There could be times where the route planned is being executed, but there's some anomaly encountered on the path, as the map could have an error, or there may be some new obstacles not represented before. On this point, the solution is to make a replanning, using the A* replanning algorithm, or a better version of this, the D* algorithm.

The A* replanning algorithm is quite simple, it just uses the A* method to construct a path, then if a discrepancy is encountered, it simply replans the movement from where the robot is, without taking into the algorithm the new obstacle encountered.
The algorithm would follow this flowchart:

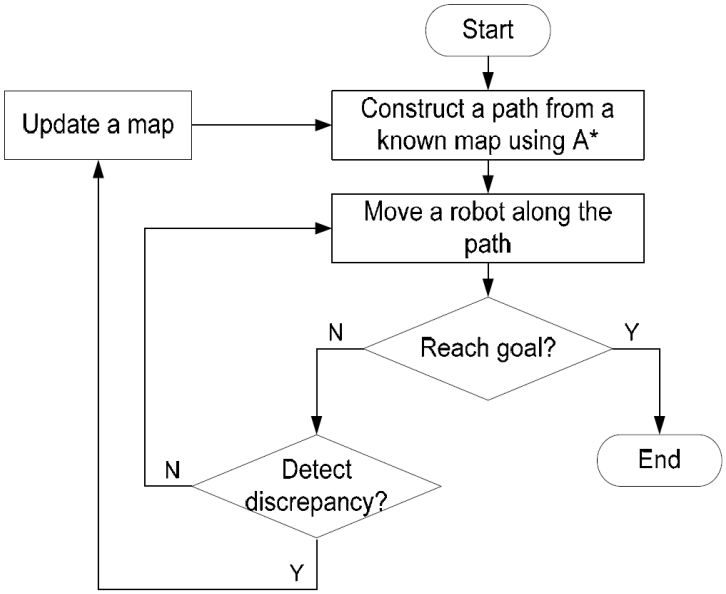

Figure 12 - A replanning algorithm flowchart. From Intensificación en Sistemas Robóticos slides, Industrial Engineering Master, Universidad de Oviedo.*

The other well-known algorithm is a bit more complex. The D* replanning method fixes the A* problem, as it includes the information acquired about the new obstacles and discrepancies with the map, providing a more efficient algorithm:

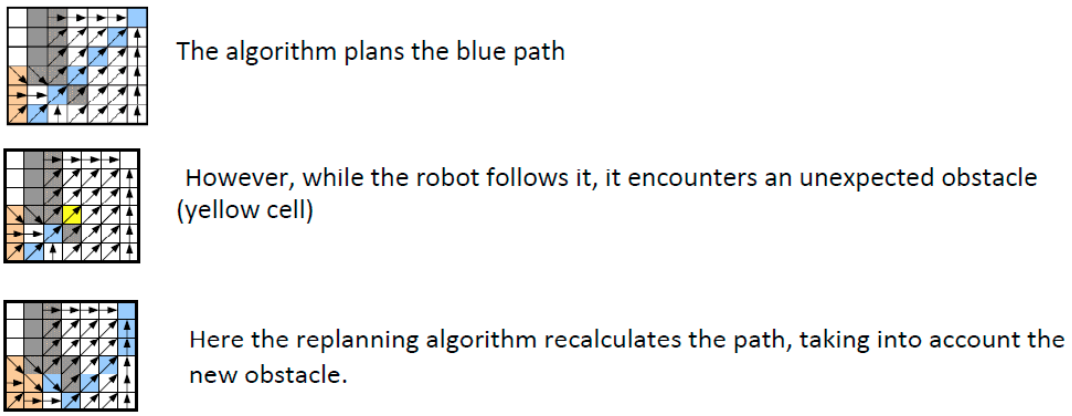

Figure 13 - D replanning algorithm representation. From Intensificación en Sistemas Robóticos slides, Industrial Engineering Master, Universidad de Oviedo.*

2.1.4. Map generation algorithms

Usually, maps and other environment's representations are not adequate and easily computable for its direct use in navigation. To overcome this, there are some methods to transform and discretize maps, in order to make them computationally efficient for searching paths and trajectories, what is the same, to enable path planning.
Creating grid map representations is not efficient at all, mainly because if the map is detailed, lots of grids would be needed. The same happens with usual space configurations, as usual maps represent mainly three-dimensional spaces, if not more, depending on what we want to map. In conclusion, the most efficient way is to create topologic maps, as these ones can store more information with less space and computation-power.

Roadmap generation

To create these topologic maps, the objective is to define an obstacle-avoiding roadmap of the space represented. This roadmaps, as topologic maps, consist of

nodes and edges, with nodes representing significant points in the environment, and edges denote possible paths and connections between these points.
There are many types of roadmaps including visibility graphs, Voronoi diagrams, or probabilistic roadmaps.

Visibility graphs

These are graphs based on transitable positions between obstacles, being these created from the vertices of each obstacle. This visibility graphs are complete roadmaps, what's the same, they find a solution path if it exists, or they output an error if not possible. They also find the most efficient path, mainly made of straight lines between each obstacle's vertices.
Its construction is simple, the simplest way is to create an empty graph, calculate the vertices of each obstacle, and for each couple of vertexes, explore I there's an obstacle between them, and if not, create a road that connects them. Iterating on this process, the roadmap is generated.
These graphs are useful for search algorithms previously stated, as Dijkstra, A*, D*...
This method also sometimes creates useless roads, between vertices that don't mean to be connected. There are some techniques to eliminate those and improve the graphs performance. They have other problems, like some roads being too close to obstacles; as close as if any little error is made in the robot's motion control, it could crash with an obstacle.

Probabilistic roadmap

Starting from an empty graph "G", the algorithm generates a random roadmap point configuration "q". Then if the points are possible (on the map's free space), its added to "G", if not they are discarded. The algorithm keeps on iterating until "G" contains a number "N" specified configurations. For each configuration "q" in "G", some "k"

neighbour points are selected, and if a path is found between “q” and “q*” (q* contains the points named k), the graph connects “q” and “q*”.
Once defined it’s valid and useful to generate trajectories between any two points, only needing start and end points, connect them as done with the other graphs nodes; and use a search algorithm like Dijkstra or A*.
This algorithm is probabilistically complete, so if there’s a solution for the roadmap, I will find it if an adequate number of points is set for the graph.

Rapidly exploring Random Tree (RRT)

A graph’s generation, for adding finally the start and end is useful in case of needing to illustrate a roadmap with multiple paths and options. Sometimes this is not necessary, so the representation can be generated starting directly on the initial node.

In this case the strategy used is based on a tree-generation, in where each node has only one origin.

This algorithm is commonly known as Rapidly exploring Random Tree, it can be implemented with the following pseudocode:

```
1:  Add initial node to the tree
2:      Repeat n times:
3:      Generate a random configuration X until X is on completely free space.
4:      y = tree's node the nearest to x
5:      If distance(x, y) > delta
6:          z = configuration between x and y & distance (z, y) <= delta
7:          x = z
8:      If it's possible to go from x to y
9:          Add x to the tree, and add y as its origin
```

This algorithm also has another more efficient variant, in which two trees are generated at the same time, one from each start and ending point, until both come together. This variation is named RRT-2.

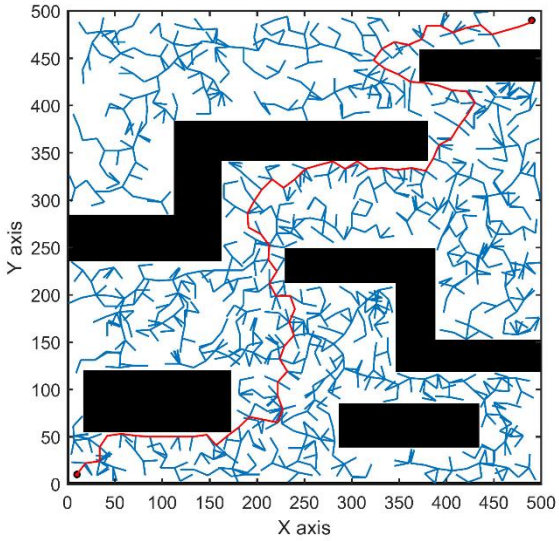

Figure 14 - RRT algorithm example. From Xinda Wang, Xiao Luo, Baoling Han, Collision-Free Path Planning Method for Robots Based on an Improved Rapidly-Exploring Random Tree Algorithm, Beijing Institute of Technology, 2020.

2.2. Software platforms

In the realm of robotics and autonomous systems the development and integration of algorithms demands a rigorous and tuneable simulation and testing environment. This simulation software platforms play an essential role in this process, as they provide a virtual space where this techniques ca be developing without facing any risk (material or human), and with fully accurate tests.

The choice of a proper simulation platform is crucial, so in this section, some of these platforms will be analysed, focusing more into Webots as the software where the algorithm explained on this project will be evaluated.

2.2.1. Webots simulation environment.

Webots is a professional robotics simulation suite, developed by Cyberbotics. It serves as a powerful, versatile, and user-friendly ecosystem to simulate and test robotics algorithms and techniques. This makes an ideal choice for this project, as it involves complete control and customization on simulations, as well as being easy to learn and use.

Some of its key features would be:

- **Physics engine:** This software features a complete physics engine that is able to reproduce realistic interactions between robots and their environment. This feature includes an accurate modelling of dynamics and cinematics of the systems simulated.
- **Robot and sensor library:** Webots includes a wide range of different pre-built robots and systems, assorted into various categories, depending on their

features or use. It also features a variety of sensor models, such as cameras, proximity or contact sensors, lidars, and many other useful ones.

- **Environment editor:** An intuitive environment editor is implemented in this program, which enables users to create and edit their own worlds according to their needs. This includes different terrains, fluids, obstacles and many more.
- **Integration with ROS:** Webots also supports ROS ecosystem, providing a great integration, allowing users to implement their simulations into real robots.
- **Real-Time simulations:** It can also perform real-time simulations, allowing the developer to analyse the behaviour of their robots or look at some statistics in real-time.

Overall, this makes Webots a great option for this project, as it's a comprehensive and complete software package aimed to design and test this kind of algorithms.

2.2.2. Webots and other robotics simulators comparison

There are many other software options in the market. Some of them offer incredibly accurate physics and simulations, or big robot prototypes libraries. But some of them also involve a much more detailed knowledge and training to get used to it.

In this section, some of these options will be compared and evaluated, mainly referencing into an article named "How to pick a mobile robot simulator: A quantitative comparison of CoppeliaSim, Gazebo, MORSE and Webots with a focus on accuracy of motion" (Farley, et al., 2022).

The simulators compared will be CoppeliaSim, Gazebo, MORSE and Webots, analysing qualitative and quantitative features.

CoppeliaSim (formerly known as V-REP, from Virtual Robot Experimentation Platform) is a powerful robotics simulation software, developed by Coppelia Robotics. It provides the user with a user-friendly interface, an extensive library of robot models and support

for different other sensors and actuators. It's widely used in education research and some industrial applications.

Gazebo is an open-source robot simulation suite that offers a realistic and flexible environment to simulate robotic systems, developed by OSRF (Open Source Robotics Foundation). This software is one of the most popular, mainly because of its high-fidelity physics engine, integration with ROS and ability to simulate complex scenarios.

MORSE is an open-source robotics simulation platform designed for simulating large robots, developed by the Institute for Systems and Robotic in the Instituto Superior Tecnico in Lisbon. This software provides a great scalable and realistic environment to simulate multi-robot scenarios or testing their algorithms. This is used in research projects and educational purposes.

Comparison

In order to compare this software platforms, the article previously mentioned provides some benchmark parameters, with qualitative and quantitative features detailed in Table 1.

Table 1 - Qualitative metrics comparation for each software (Farley, et al., 2022).

Metric name	CoppeliaSim	Gazebo	MORSE	Webots
Free to use	True	True	True	True
Open source	False	True	True	True
ROS Compatibility	Built-in plugin provided	Yes	Yes	A built-in plugin provided
Programming languages	C/C++, Python, Lua, MATLAB, Java, Octave	C/C++, Python	Python	C/C++, Python, Java, MATLAB
UI functionality	Full functionality	Full functionality	Visualization only	Full functionality
Model format support	URDF, SDF, Stl, Obj, Dxf, Collada	URDF, SDF, Stl, Obj, Collada	Blend	Proto Nodes
Physics engine support	Bullet, ODE, Vortex, Newton	Bullet, ODE, DART, Simbody	Bullet	ODE

On the other hand, there are some measurable quantitative metrics, essentially about simulation's efficiency, CPU loads and real-world fidelity. These are shown in Table 2.

Table 2 - Quantitative metrics comparation for each software (Farley, et al., 2022).

Metric name	CoppeliaSim	Gazebo	MORSE	Webots
Real time factor	0.973	1.064	0.839	0.903
Average load CPU efficiency	11%	23%	12%	4%
Intense load CPU efficiency	12%	23%	12%	7%
IMU angular velocity error	21.46 rad/s	18.76 rad/s	24.42 rad/s	22.30 rad/s
IMU linear acceleration error	247.36 m/s^2	340.39 m/s^2	624.23 m/s^2	359.66 m/s^2

Real time factor is calculated by taking the ratio of the sum of simulated time steps to the sum of desired real time steps. The closer to 1, the most accurate.

IMU accuracy is basically the accuracy of the data measured in the simulator, compared to real data. The numbers measured look so large, this is because the researchers decided to add up all angular velocity and linear acceleration error,

obtaining larger numbers than if made with means or other similar approaches. Obviously, for this metric, the smaller the number is, the best.

These results are all summarized on Table 3, where each metric has been given a weight (out of 10), depending on how crucial that measurement is.

Table 3 - Software's benchmark metrics summary and result (Farley, et al., 2022).

Metric name	Weight	CoppeliaSim	Gazebo	MORSE	Webots
Free to use	4	1.000	1.000	1.000	1.000
Open source	2	0	1.000	1.000	1.000
ROS Integration	6	0.800	1.000	1.000	0.800
Programming languages	3	1.000	0.667	0.333	1.000
UI functionality	6	1.000	1.000	0.333	1.000
Model format support	4	1.000	1.000	0	0.250
Physics engine support	3	1.000	1.000	0	0
Real time factor	4	0.914	1.000	0.789	0.849
Average load CPU efficiency	2	0.364	0.174	0.333	1.000
Intense load CPU efficiency	2	0.583	0.304	0.583	1.000
IMU angular velocity error	10	0.875	1.000	0.792	0.867
IMU linear acceleration error	10	1.000	0.713	0.424	0.736
Total	**56**	**49.100**	**49.087**	**32.148**	**44.226**

The final result gives us two remarkably close simulators, CoppeliaSim and Gazebo. These are the most accurate ones, and the most compatible in terms of programming languages, ROS integration and formats supported.

Conclusion

The comparison gives us two software that are quantitatively better than the other, however, the Webots platform is indeed not that far from them.
According to the data, the Webots software is the most efficient of the four, giving also great marks on compatibility, real time factor and simulated vs real data errors. This added to the user-friendly interface, the software being open source, and the wide range of pre-built robots and sensors, make it an excellent choice for this project.

2.3. Performance metrics

In order to judge if the implementation of the algorithms explained before, some performance and efficiency metrics should be stated. For navigation, planning or obstacle avoidance algorithms, we can state some performance criteria:

- Mission success: number of successful missions.
- Path length: distance travelled to accomplish the task.
- Time: time taken to accomplish the task.
- Collisions: number of collisions per task, per distance and per time
- Obstacle clearance: minimum and mean distance to the obstacles
- Robustness in narrow spaces: number of narrow passages traversed.
- Smoothness of the trajectory: relative to control effort

This performance metrics have been stated by an article named “Quantitative metrics for Mobile Robots Navigation” (Muñoz & Valencia, 210). These provide a great approach to similar algorithms and the benchmarks used to judge each implementation.

These metrics can also be sorted in three different importance orders:

- Safety metrics: metrics expressing the relationship between the data and the safeness of the trajectory followed. This includes the mean distance between

the robot and the obstacles through the hole mission, and the minimum mean distance to obstacles too.

- Dimensional metrics: the trajectory to a goal is considered in space and time means. This metrics include length of the covered trajectory, mean distance to goal or time taken to accomplish the mission.
- Smoothness metrics: smoothness in a trajectory shows the consistency between de decision and action that the navigation control system makes. These me be more qualitative or subjective metrics, meaning that the supervisor manipulating the robot, can judge if the robot's movement and actions are smooth and accurate.

3. System design

In this section, I will delve into some important aspects of the system's design, where the foundation of the project will be led. By making these careful considerations and setting up the simulation environment appropriately, we aim to create a, accurate environment to successfully evaluate and validate each algorithm's performance.

3.1. E-puck robot

The e-puck robot is an open-hardware and onboard open-source-software developed by Michael Bonani and Francesco Mondadada at the EPFL (Switzerland). It is a small differential wheeled mobile robot really spread within the scientific community research ambit and also orientated to educational purposes, due to its open hardware

characteristic, being built and sold by several companies and making its price really competitive and suitable for these ambits (Gonçalves, et al., 2009).

Its main characteristics are highlighted as:

- Size and weight: it's a really compact and lightweight robot, making it easy to transport and manipulate
- Differential wheeled design: this kind of design enables programming smooth and precise movements
- Sensors and expandability: it is equipped with a nice variety of sensors allowing to scan its surroundings. It also features expansion slots to add additional sensors or other modules, making it flexible in that aspect.
- Processing power and communications: The robot is powered by a microcontroller which provides sufficient processing power to run algorithms

and commands. It also features wireless communications support, making this robot suitable for communicating with other robots or a central control system.

A more detailed specifications table is given:

Table 4 - E-puck robot specifications (Gonçalves, et al., 2009).

Diameter	70 mm	**RAM**	8 kB	
Height	50 mm	**Flash memory**	144 kB	
Weight	200 g	**Sensors**	8 infrared proximity and light sensors	·3D accelerometer
Max speed	13 cm/s	**Camera**	VGA Colour camera (640x480)	
Autonomy	2 h moving	**LEDs**	8 LEDs in ring + 1 body + 1 front	
Microcontroller	dsPiC 30F6014A @60MHz	**Comms**	Standard Serial Port, Bluetooth	

On the other hand, there are some other interesting points or features of this kind of robots:

- Ease of use: it features user-friendly software interfaces and development tools.
- Documentation and Community Support: lots of comprehensive documentation, tutorials, or sample code to use. It also has a wide online

community of users and developers, providing other ideas, resources, and troubleshooting.

- Affordability: It's open-hardware nature allows companies to develop these on competitive prices, allowing it to be used on educational and research purposes with limited budgets.
- Research and educational impact: thanks to much of the above-mentioned points, this kind of robot is widely spread in these areas, as a significant research and educational tool.
- Simulation: it's also a widespread robot implemented in numerous simulators as, especially for this project, the Webots platform.

3.2. Simulation environment set-up

All robot movement simulations will be carried out on Webots, using the MATLAB plugin and coding for the controller's implementations. The Webots environment is a really customisable and flexible environment, permitting precise bit also graphically accurate simulations. Depending on the algorithm being tested the set up will be a bit

different, but there are some common main characters on this play:

- World setup: each world created on Webots consists of some basic parameters needed to perform the simulation. These are defined as “WorldInfo” and contain the basis for the simulation. Some of the most important include:
    - Basic world info: includes title, description, or window management.
    - Gravity: sets up the gravitational force acting on the objects on the simulation. Set as 9.81 m/s^2 as default.
    - CFM (Constraint Force Mixing): controls the force of the physics engine, in order to manage joints and constraints.
    - Physics: to enable, disable or change the physics engine parameters
    - Basic time step: specifies the timestep used for calculations on the simulation. Will be set to 64 ms.
    - FPS: defines the desired framerate for rendering the simulation’s graphical environment. Set to 30 fps for performance.
    - Coordinate system: defines the coordinate system type.
    - GPS coordinate system and reference: specifies the coordinate system type used by GPS sensors and sets up the origin point for these. Set to local and (0,0,0) by default.
    - Force and torque scales: these parameters affect the drag and resistance forces and torques of the simulation.
    - Contact properties: specifies properties related to object collisions and contact physics
    - Other performance parameters
- A viewpoint and background light will also be needed. These need to be adjusted in order to have a visual good-looking simulation.
- The e-puck robot is the main essential on the simulation:
    - Translation and rotation: which will define the starting point and orientation of the robot
    - Controller: specifies the controller loaded into the robot
    - Supervisor: enables the robot controller to access the “administrator” functions, as accessing some world parameters, or other robot’s functions and control.

- Camera settings: camera resolution, noise and FOV adjustments
- Sensor slots: this enables the user to add other sensors on the robot's expansion slots. In this work this will be used to include the GPS and compass modules. A LiDAR will also be included to simulate sensor-based navigation algorithms (Bug2 algorithm and Artificial Potential Fields).
- Other settings: battery drain simulation, emitter and receiver channels for communications, and others.

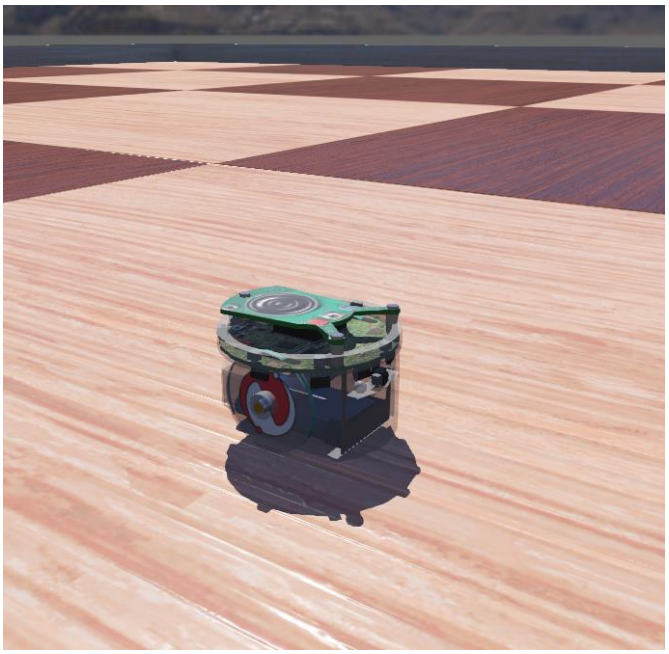

Figure 15 - E-puck robot on Webots simulation.

- The plain rectangle arena will also be present on all the simulations. This node crates a rectangular plain tiled arena, surrounded by walls. This will allow to

test the robot on a plain terrain, as well as being the basis to place other objects and obstacles.

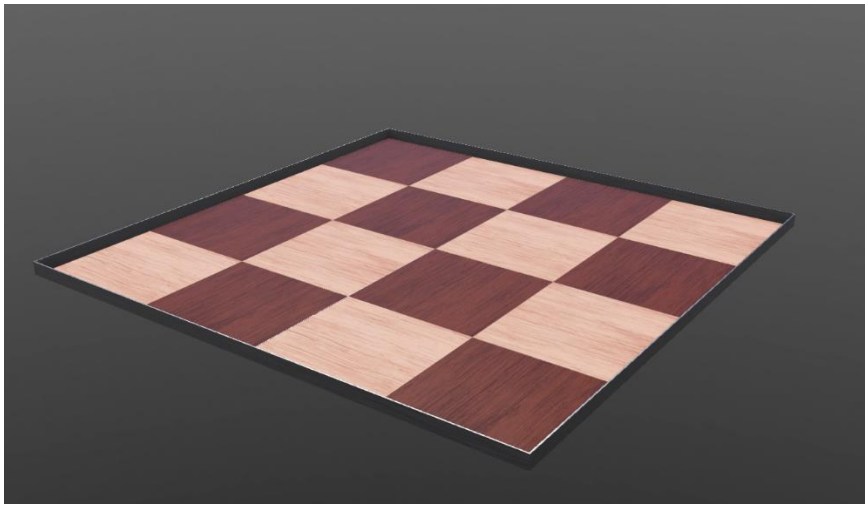

Figure 16 - Webots rectangle arena node.

3.2.1. Basic navigation algorithms test arena

These simple algorithms will just be tested on the environment just mentioned. There's no need to add anything else to the simulation, as the only need on this test is to allow

the robot to move along the arena and collect this movement data in order to analyse it later on.

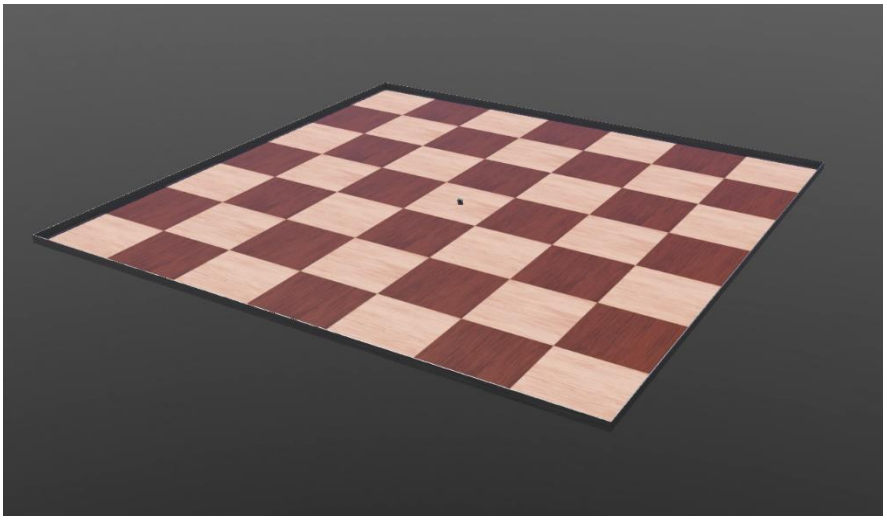

Figure 17 - Simulation visual set-up for go to a point algorithm.

3.2.2. Sensor-based Navigation Algorithms

In this case, these algorithms will need a more complex test arena, as some obstacles will be needed to test the performance of these. As mentioned before, in this case a

LiDAR sensor will be included on the sensor's slot of the e-puck robot. The first obstacle test arena used for the Bug2 algorithm will be the following:

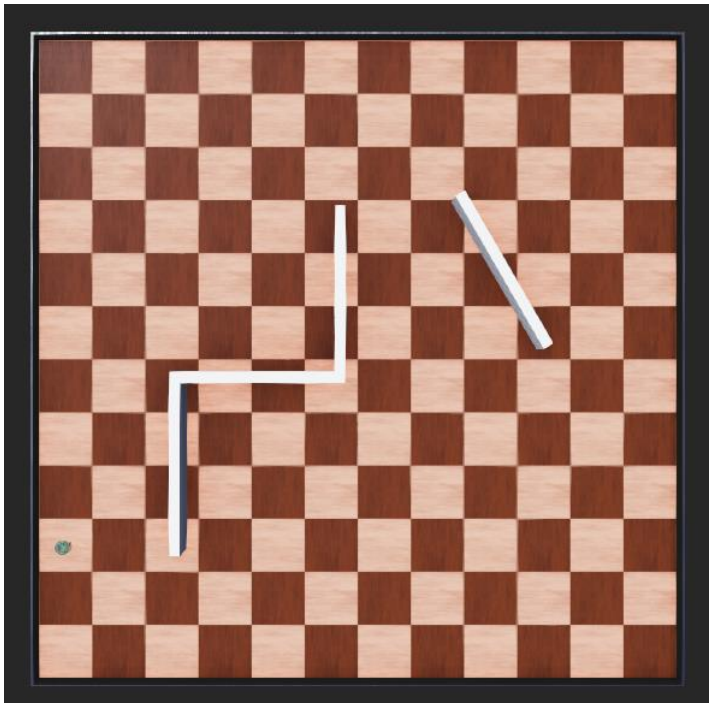

Figure 18 - Set-up for Bug2 algorithm testing.

To test this kind of algorithm's performance, other kind of obstacle display will be tested, in order to test its strengths and weaknesses:

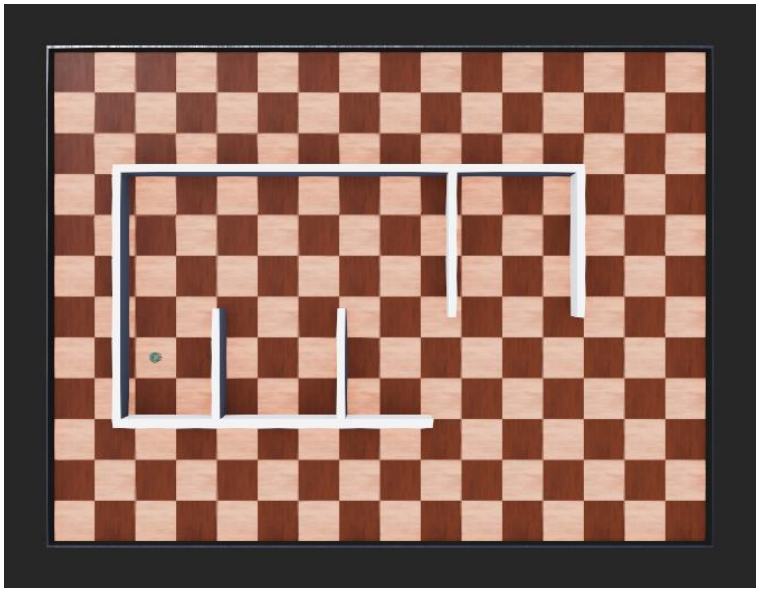

Figure 19 - Bug2 algorithm second test arena.

To finish with the sensor-based navigation algorithms being implemented, we have the Artificial Potential Fields algorithm, which due to its complexity and environment-depending nature, it's difficult to define the constants that define its operation, this algorithm will be tested in a really simple obstacle map:

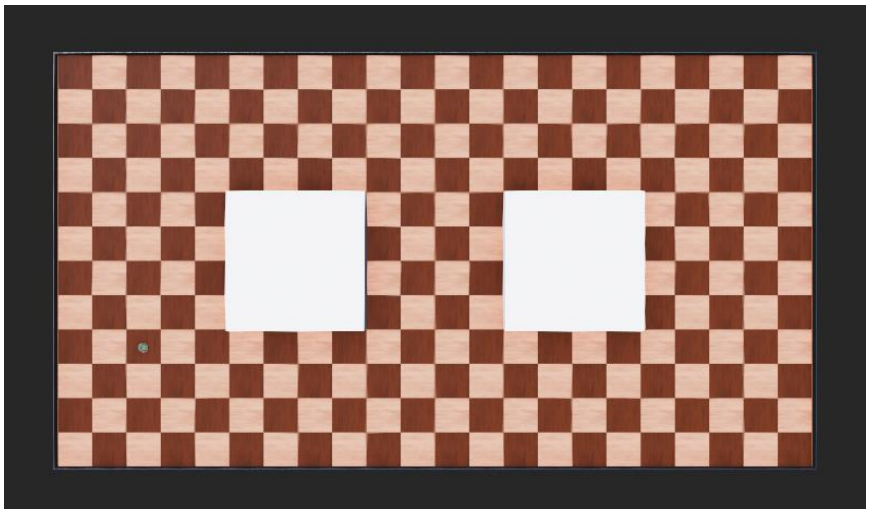

Figure 20 - APF algorithm test arena set-up.

3.2.3. Map-based Navigation Algorithms

In this case, defining the map and environment of the simulation is crucial, as these algorithms will be tested against these designs. In case of Grassfire algorithm or map generation ones, these will be tested on a maze:

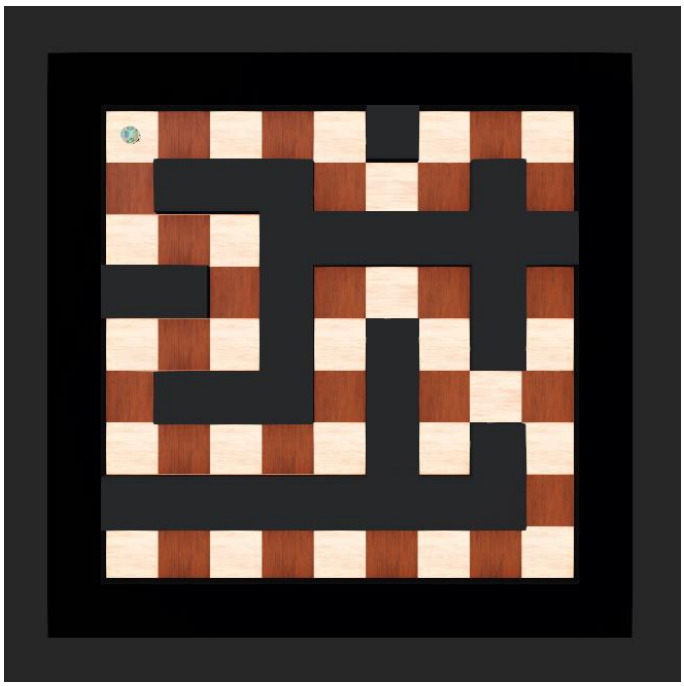

Figure 21 - Map-based algorithms maze test arena.

For more complex algorithms, this maze can also be used, but it's more interesting to analyse those in a different map or scenario, which includes different paths or

solutions to get to the same target point. These search algorithms like Dijkstra or A*, will be tested on the following map:

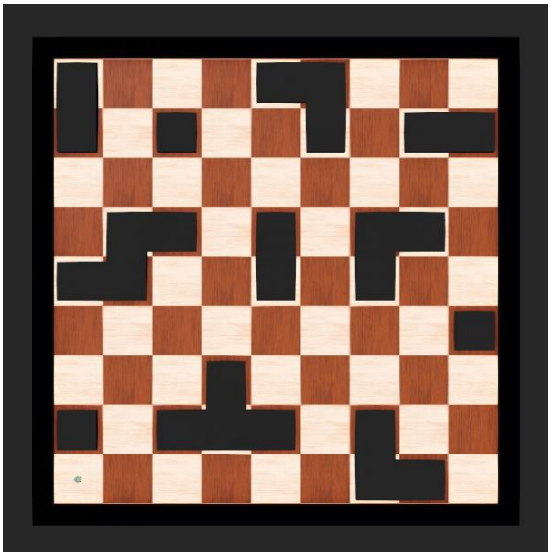

Figure 22 - Search algorithms test arena obstacle layout.

3.2.4. Map generation algorithms

Finally, the map generation algorithms considered for this project, such as probabilistic maps and RRT map generations, will directly be implemented on the MATLAB suite. These algorithms don't generate something suitable to simulate on Webots directly, as it does not control a robot's movement or navigation. In these cases, simulations will be implemented on MATLAB, providing graphs and data, in order to understand their functioning and estimate their performance in different cases. In case of probabilistic map generation, this algorithm will serve as main foundation to implement search algorithms (Dijkstra and A*), as it is capable of generating a visual graph of one of the simulation arenas showed above, in which these search algorithms will be tested then.

3.3. Data collection and analysis

These kinds of precise and realistic simulations generate extensive datasets, for which some powerful processing tools to undergo their analysis. MATLAB stands

out as an ideal platform for this task, as it's really capable of handling large volumes of data and still work efficiently. Since all the implementations are already coded on MATLAB language, transferring these data from the Webots platform to MATLAB ensures an easy and smooth workflow.

4. Implementation

In this section, we delve into the implementation details of each algorithm, aiming to provide a clear understanding of its functionality and operation within our system. While the code implementations are provided in the annexes for reference, our focus here is on elucidating the underlying principles and mechanisms behind each algorithm's behaviour. Through illustrative explanations and conceptual breakdowns, we aim to demystify the implementation process and foster a deeper comprehension of how these algorithms are applied in practice. Let's explore each algorithm's functioning and discuss the key considerations in its implementation.

4.1. Simple navigation algorithms – Move to a point

The first and simplest but necessary algorithm consists of making the robot go from its initial position to a target point in a precise and efficient way. This algorithm implements two simple proportional controllers, one used to control the robot's linear speed towards the goal; and the second one used to steer the robot and

keep it always facing our target. The main algorithm is easily explained with the following flowchart:

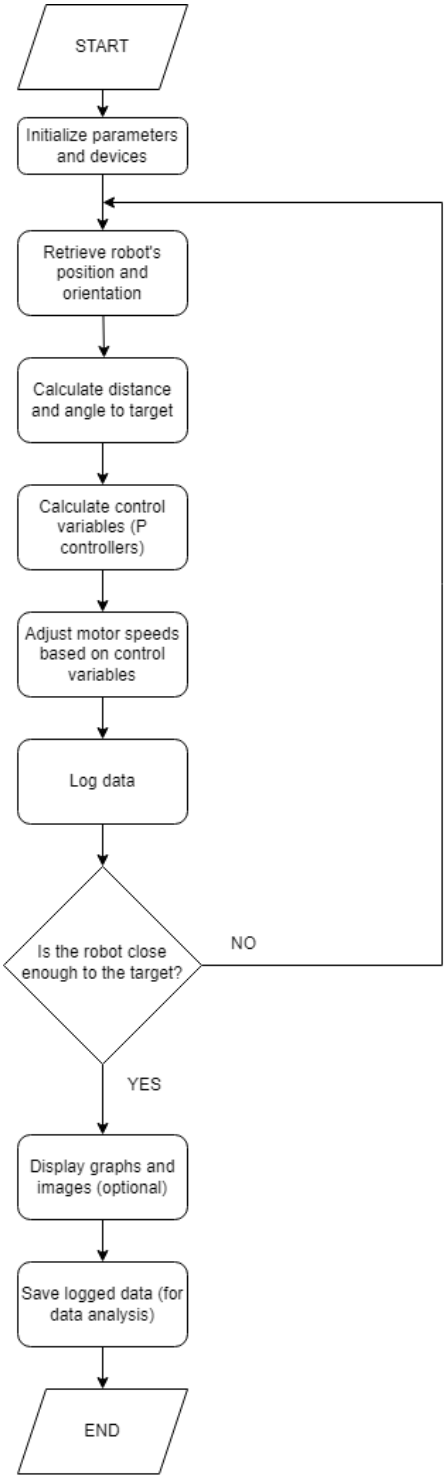

Figure 23 - Go to a point algorithm flowchart.

Once designed and implemented, the controllers' variables must be tuned in order to find an optimal functioning of this one. The variables for this tunning are the proportional gains of both linear speed and steering of the robot. These will be the main analysis for this simple algorithm, which will be discussed in next section.

4.2. Simple navigation algorithms – Follow a trajectory

On the other hand, it comes the second but really important simple move algorithm, the trajectory following. Being able to design an accurate and efficient trajectory following algorithm is a must in these cases, as then this will serve as foundation for other "trajectory calculating algorithms", as those won't implement the robot's movement itself. The strategy used to address this part, is known as Pure Pursuit, as it involves calculating a moving point close to our robot, but

always faced towards our goal trajectory, making this point closely go before the robot's movement.

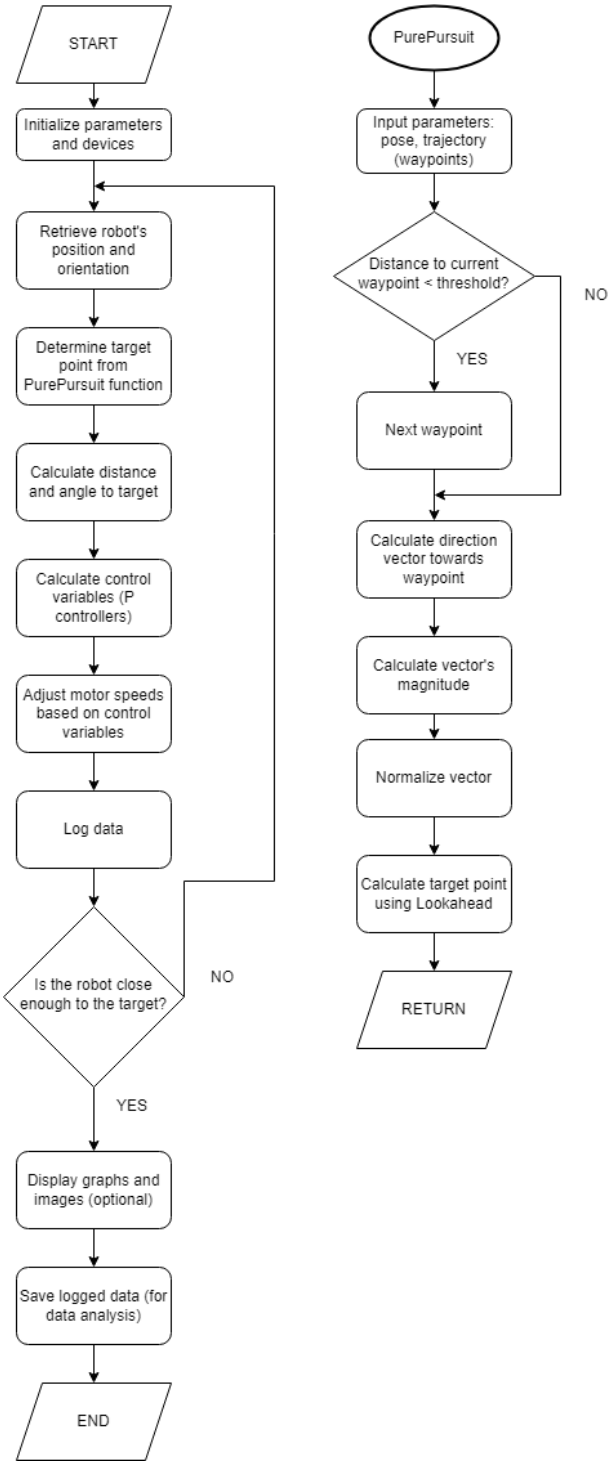

Figure 24 - Trajectory following and Pure Pursuit algorithm flowchart.

In this case, variables to be tuned are P controller's gains for speed and steering, but there's other really important parameter, the Lookahead distance, which determines the point along the trajectory which the robots aim to pursue. This parameter governs how far ahead does the robot anticipate the trajectory, making it a critical factor as it directly affects the robot's ability to follow the desired trajectory accurately. Adjusting this parameter can lead to a responsive, smooth, and accurate trajectory following algorithm.

4.3. Reactive Navigation – Bug2 Algorithm

Starting with reactive navigation algorithms, bug algorithms are some of the most basic but effective ones. In this case, I have designed the improved Bug2 algorithm

version, as it stands as the evolution and combination of Bug0 and Bug1 but obtaining much better overall performance in most of the scenarios.

This kind of algorithm can be easily implemented as a Finite State Machine (FSM):

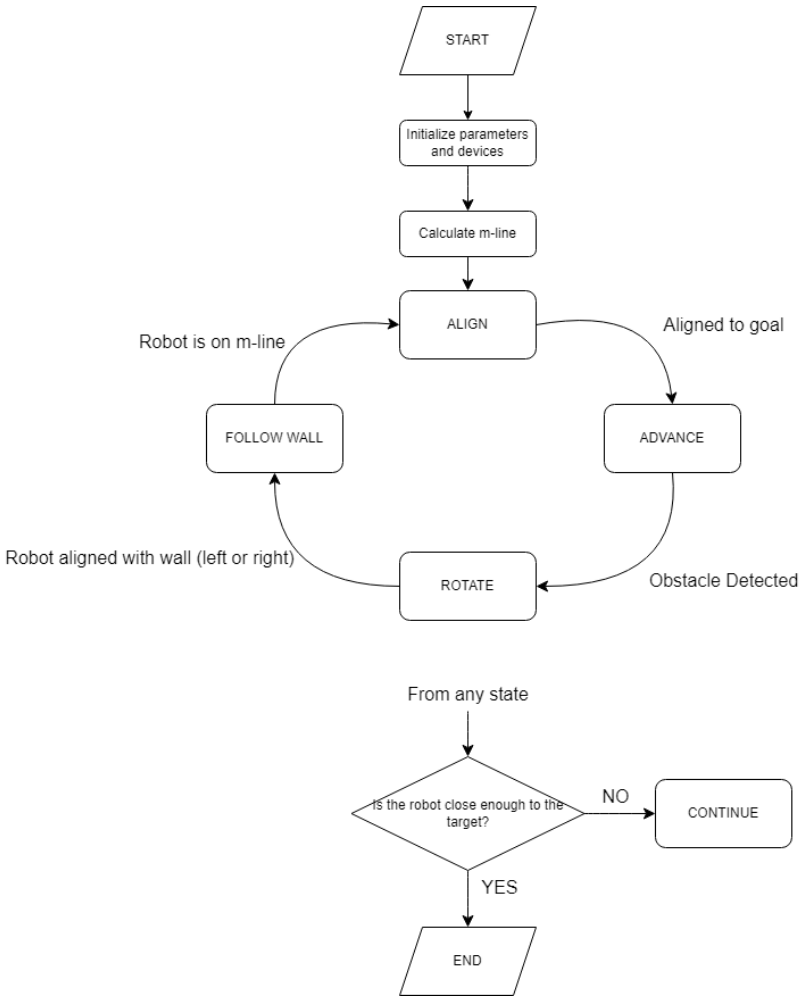

Figure 25 - Bug2 algorithm state machine implementation.

This diagram contains only the basic statements to make the algorithm work, but each state and transition have their own internal variables, conditional statements and actions, which implementation can be consulted on the code appendixes of this project.

As other comments, in this case we get again two proportional controllers and their gains to tune, one controls the speed at which the robot advances towards

the goal (really similar to the cases studied before), and the other one monitors the distance between wall and robot during the wall-follow state. Tuning this second proportional gain correctly, should lead to a precise wall following algorithm.

4.4. Reactive Navigation – Artificial Potential Fields

This Artificial Potential Fields approach for reactive navigation techniques requires an accurate and well-defined workflow. The core of these algorithms is to mimic the behaviour of a physical system where the robot is seen as a particle navigating through a field of forces. The goal point acts as an attractive force that pulls the

robot towards it, while all obstacles generate a repulsive force in order to push the robot away from them.

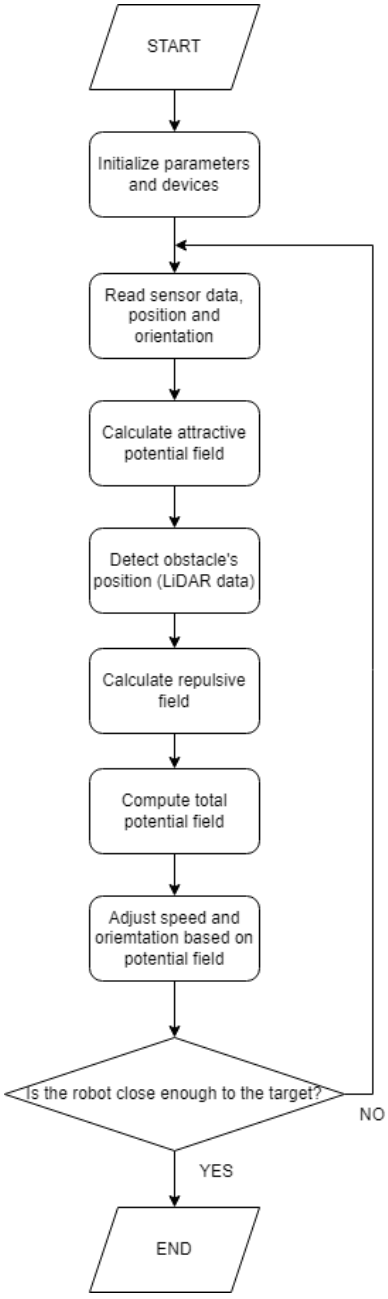

Figure 26 - AFP algorithm flowchart.

The key components of this algorithm are the forces and their calculations. This kind of techniques are highly constant dependant, as for each obstacles display or

robot's position, some values for the potential fields constants will make the robot work its way to the goal point or not. The main focus here will be to try to find the suitable values for these, in our simulations case.

4.5. Map-Based Navigation – Dijkstra's algorithm

Dijkstra's search algorithm is a fundamental algorithm in robotics and graph theory, used to find the shortest path between two nodes in a graph. There are various important concepts to take into account before implementing this kind of algorithm.

First of all, Dijkstra's algorithm operates on a graph, a collection of nodes connected by edges. Each of these edges or links have an associated weight or cost, typically representing distance, time, or another relevant metric. In this case, for simplicity and efficiency, distance cost functions will be the metric chosen. In this part, graphs will be assumed to be already implemented.

This algorithm always aims to find the shortest path from a designated node to all of the other nodes of the graph. The shortest path to each will be the one with the minimum total cost. Dijkstra implements a greedy approach, which means that it

makes locally optimal choices on each step with the hope of finding a globally optimal solution.

In this case once the path is calculated through the graph, the trajectory following algorithm (Pure Pursuit) will be used, in order to achieve the robot's navigation through the map.

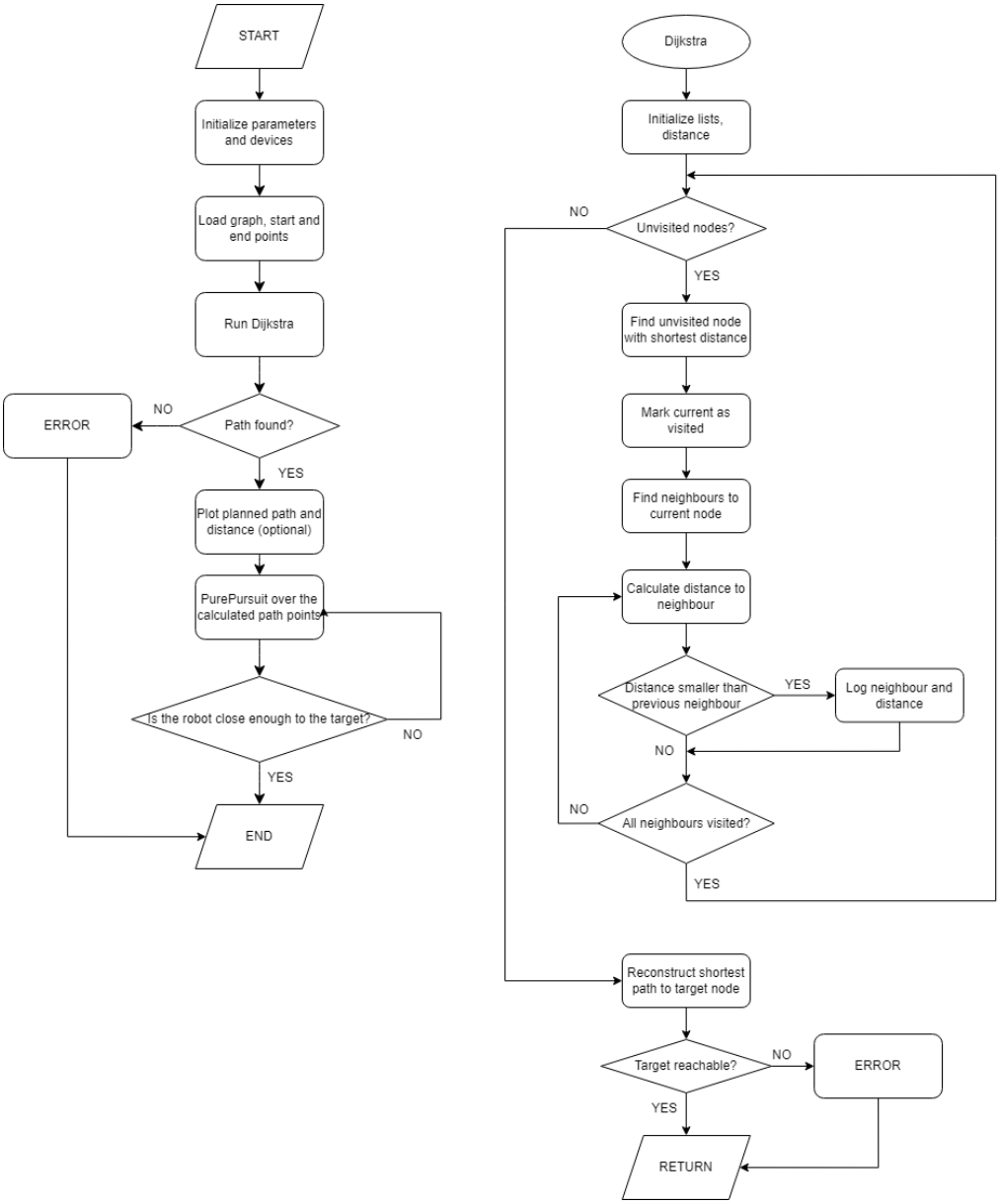

Figure 27 - Dijkstra's search algorithm and path following implementation.

4.6. Map-Based Navigation – A* algorithm

The A* algorithm is a widely used search algorithm, especially in robotics or the latest artificial intelligence applications, particularly used in pathfinding and graph exploration. It introduces an “extension” or improvement in some cases of the Dijkstra’s algorithm, as it adds the benefit of applying a heuristic function to guide the search process.

A* (A star) operates over a graph constructed by nodes connected by edges, with again each edge having an associated cost representing some relevant metric. What sets this algorithm apart is the inclusion of a heuristic function, which helps guiding the search towards the goal node more efficiently.

This approach implements a best-first search strategy, prioritizing nodes with lower costs for exploration. By doing this, it effectively balances the greedy approach of Dijkstra’s algorithm, and the informed search provided by the heuristic functions.

Again, in this project’s context, implementing the A* algorithm for map-based navigation, provides the robot with an intelligent pathfinding capability, as once the algorithm calculates the shortest path through the graph, it can be again integrated

with the Pure Pursuit algorithm implemented before to enable smooth and efficient navigation through the computed waypoints.

The main algorithm, will be the same as Dijkstra's but obviously changing the main searching function for:

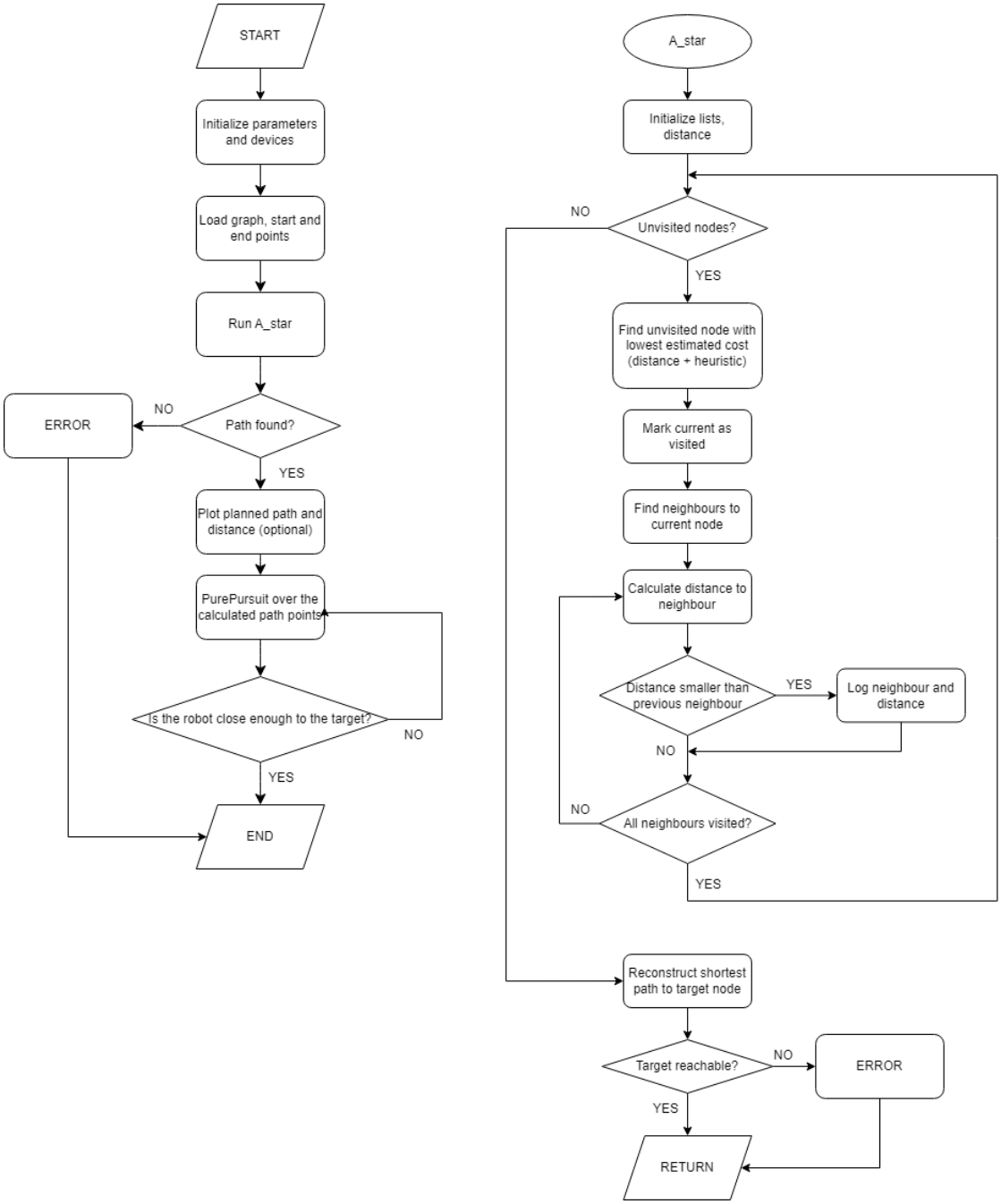

Figure 28 - A algorithm and trajectory follow flowchart.*

4.7. Map generation algorithms – Probabilistic roadmaps

Probabilistic roadmaps algorithms (PRM) are one of the fundamental approaches to map generation in robotics and motion planning. This algorithm generates a roadmap (graph), by randomly sampling points in the map, assessing if they're transitable or not, and then stablishing connections between valid configurations for paths. Unlike other graph-based approaches or grid maps, this kind of algorithm is capable of handling complex obstacle geometries and spaces configuration.

The process typically involves different steps. First of all, handling the space representation, having many ways to do so, it allows to recognise the obstacles position, and to determine which points would be transitable or which paths would be viable or not. Then the map is randomly sampled, placing points all over the space representation, making sure that each of these is in an unoccupied position and discarding the ones that are not. Then a connection is stablished between close nodes (how close they are is usually determined by a control parameter),

that are possible to link, creating a network of paths through which the robot can move along.

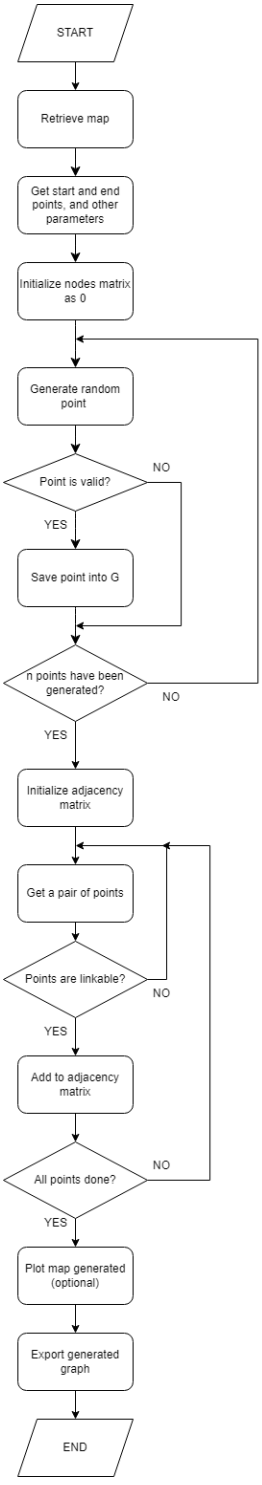

Figure 29 - Probability Roadmaps generation algorithm flowchart.

To determine if two points are linkable, the Bresenham algorithm has been implemented. The Bresenham algorithm is a line-drawing technique used to drag lines between 2 points in a discrete grid (in this case the map given). It's commonly used on computer graphics and image processing because of its efficiency and simplicity. Determining the line between two generated points and checking if it goes through any of the grid cells occupied by a wall, determines if two points are linkable or not.

4.8. Map generation algorithms – RRT (Rapidly-Exploring Random Tree)

The Rapidly-Exploring Random Tree (RRT) algorithm is a popular and simple motion planning algorithm widely used in robotics to efficiently find possible paths for robots operating in complex spaces.

This algorithm starts randomly creates points on the map, and connects them to the existing point tree, continuing with this process until the maximum iterations or

the goal point are reached. It also checks for collisions when creating a new point, in order to create a viable path through the map.

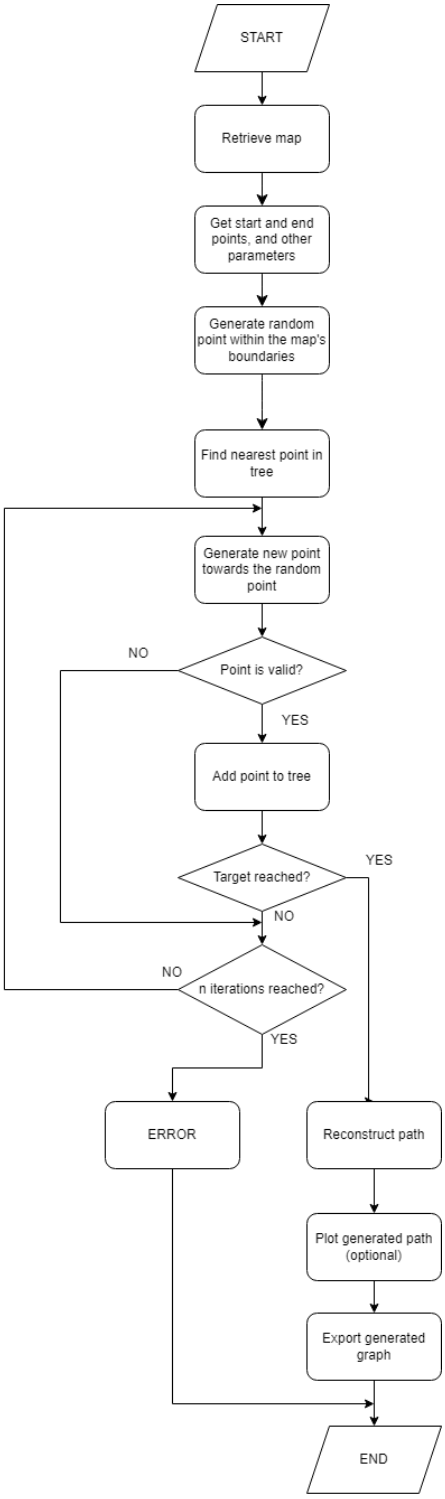

Figure 30 - RRT flowchart.

5. Results and discussion

Having explored the implementation and functioning of each algorithm within these systems, it's time to turn the attention into the results obtained from our simulations, testing each of them for different situations and analyse and interpret all the data collected during these experiments, shedding some light into each algorithms performance, effectiveness, and behaviour in each case.

5.1. Move to a point

In this first section, results from the experimentation with the move to a point algorithm will be presented and evaluated. This analysis will be done under varying configurations of both steering and speed proportional controller's gain. By systematically changing these values, the aim is to assess their impact on the algorithms effectiveness and precision in guiding the e-puck robot towards the target point.

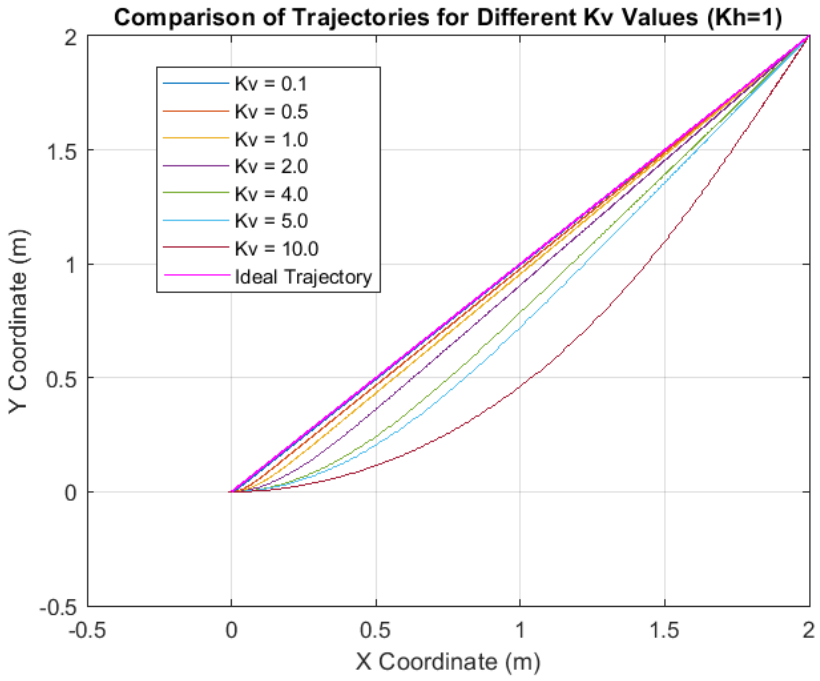

Figure 31 - Move to a point algorithm trajectories for Kv comparison.

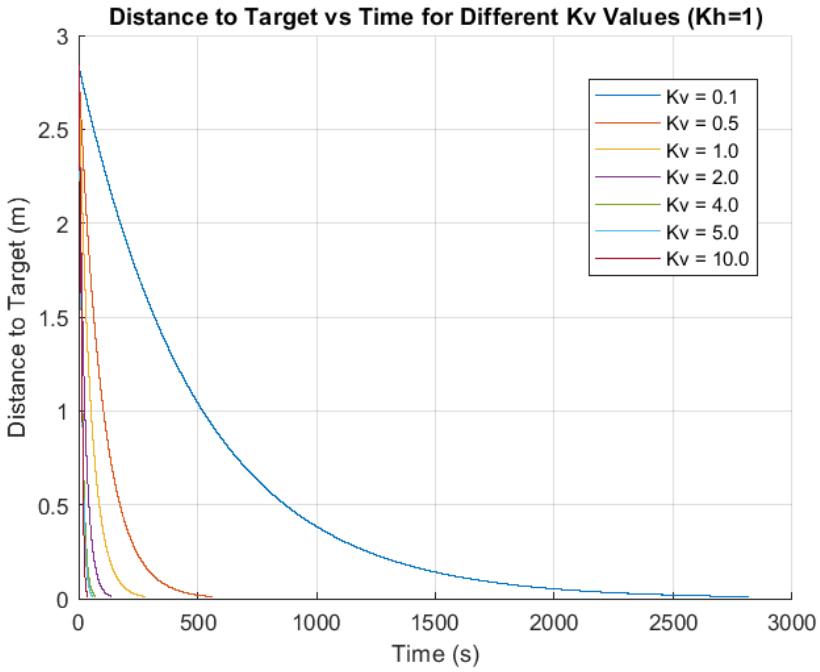

Figure 32 - Move to a point algorithm distances to target for Kv comparison.

Table 5 - Kv values and statistics comparison for move to a point algorithm.

Kv values (Kh=1)	0.1	0.5	1	2	4	5	10
Mean error (m)	1.32e-5	1.29e-5	4.28e-5	1.41e-3	6.00e-3	9.80e-3	4.13e-2
Standard deviation (m)	2.65e-4	5.30e-4	1.28e-3	3.33e-3	1.05e-2	1.51e-2	4.07e-2
Ideal distance (m)	2.8284	2.8284	2.8284	2.8284	2.8284	2.8284	2.8284
Real distance travelled (m)	3.1789	2.8569	2.8443	2.8524	2.8792	2.8933	2.9707
Mean Speed (cm/s)	0.11269	0.5062	1.0066	2.0176	3.9018	4.6993	7.4988
Time to Target (s)	2820.928	564.416	282.624	141.440	73.856	61.632	39.680

Starting with the tuning for the proportional controller gain for the speed (Kv). The steering gain has been set to a neutral unit value in this case.

Statistic values such as mean error and standard deviation have been computed by comparing the trajectory followed by each case, against an ideal trajectory to the objective point, in this case, a straight line from start to target points.

It's easy to realise how does this value affect the robot's performance. This controller gives the wheels speed an input of Kv times the distance to the objective. This makes high Kv values to achieve much higher mean speeds, and lower times to target values. On the other hand, this makes the robot make much larger curves, as the robot advances quickly towards the goal, while the steering controller hasn't already achieved to match the target point orientation completely. We can clearly observe this

in Figure 31, where high valued Kv controllers make much larger radius turns, while lower ones get similar results to the ideal straight line.
On the other hand, Figure 32 shows the distance to target vs time graphic, where we can see how higher values achieve the objective point in an acceptable amount of time, while low values spend too much time to do so.
In this case, the objective would be to achieve a precise but fast result, so intermediate values should be fine, for example a value of Kv=5, gives a fast navigation, while maintaining an error of less than 1 cm, which is perfectly acceptable.

Next would be analysing the effect of changing the controller gain for steering (Kh), again assigning a neutral or good value for Kv. Using the results of last section, a value of Kv=4 has been selected.

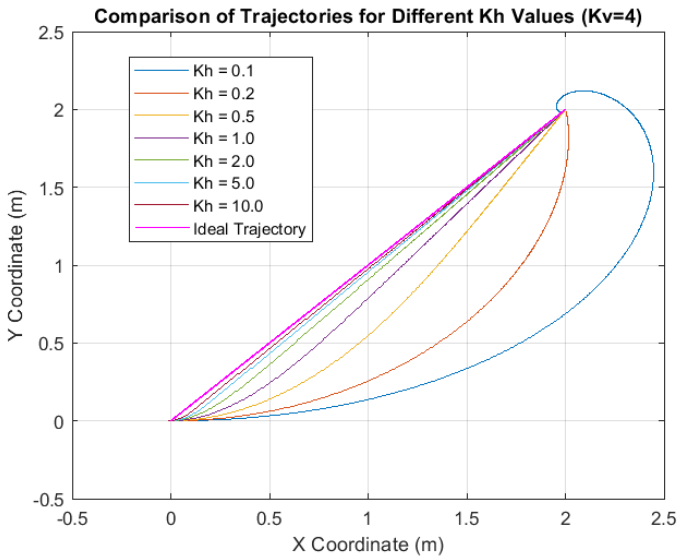

Figure 33 - Move to a point algorithm trajectories for Kh comparison.

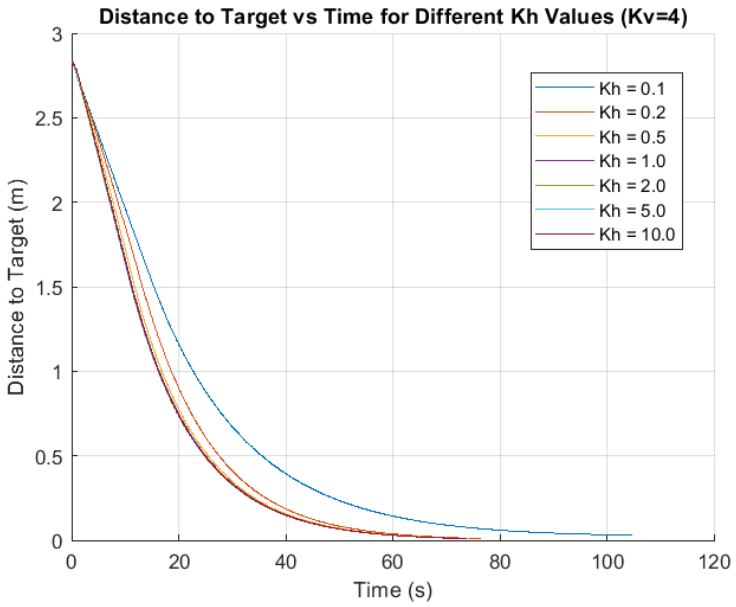

Figure 34 - Move to a point algorithm distances to target for Kh comparison.

Table 6 - Kv values and statistics comparison for move to a point algorithm.

Kh values (Kv=4)	0.1	0.2	0.5	1	2	5	10
Mean error (m)	0.3155	9.76e-2	1.90e-2	6.00e-3	1.85e-3	6.36e-4	2.50e-4
Standard deviation (m)	0.2545	8.51e-2	2.62e-2	1.05e-2	4.04e-3	1.77e-3	9.16e-4
Ideal distance (m)	2.8284	2.8284	2.8284	2.8284	2.8284	2.8284	2.8284
Real distance travelled (m)	4.1565	3.1766	2.9381	2.8792	2.8522	2.8414	2.8356
Mean Speed (cm/s)	3.9697	4.1501	3.9576	3.9018	3.8752	3.8573	3.8527
Time to Target (s)	104.768	76.608	74.304	73.856	73.664	73.728	73.664

In this case, lower values clearly achieve poorer results. This controller applies a difference of speed between the two differential wheels in order to make the robot rotate. The value of this difference is determined by Kh times the difference in orientation (angle) between the robot's actual heading and the point's position.

Lower values mean much slower and smooth turns, but affecting this to the time taken, the big curve covered to achieve the objective and the precision of the

trajectory compared to the ideal one. Even for the lowest value, the proximity threshold value to consider the objective reached, had to be incremented, as the controller wasn't able to achieve that precision.

In this case, the conclusion is easy to get to. A high value outputs much lower mean errors, distance covered and time to target. The highest value does not mean to be the best, as in some cases, it would make the robot oscillate too much, so a value of Kh=2, would be perfect, achieving the fastest result and still maintaining a mean error of less than 1 cm.

5.2. Follow a trajectory – Pure Pursuit

Moving onto the results of the trajectory following algorithm implementation, the data from the Pure Pursuit algorithm will be analysed. Again, this analysis will be based on tuning the controller gains (Kv, Kh), and also understanding the Lookahead (L) factor, and how do these impact the robot's behaviour while following a trajectory. For cases like straight lines, this does not have much interest, as it only applies a move to a point strategy, obtaining similar results to the section before. In this case, the algorithms will be tested against a sinusoidal trajectory,

which involves some straight-like sections and sharp turns, perfect to test the algorithms and movement possibilities and limitations.

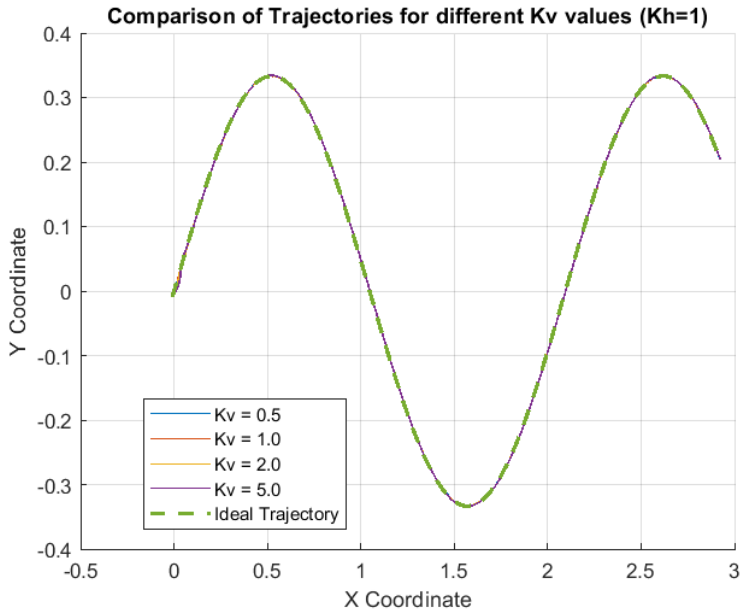

Figure 35 - Pure Pursuit algorithm trajectories for Kv comparison.

Table 7 - Kv values and statistics comparison for Pure Pursuit.

Kv values (Kh=1, L=0.1)	0.5	1	2	5
Mean error (m)	1.44e-4	1.30e-4	1.57e-4	5.49e-4
Standard deviation (m)	2.07e-4	3.26e-4	5.80e-4	1.28e-3
Ideal distance (m)	3.5363	3.5363	3.5364	3.5368
Real distance travelled (m)	3.5409	3.5378	3.5369	3.5412
Mean Speed (cm/s)	0.1002	0.2002	0.4003	1.0007
Time to Target (s)	3532.352	1766.464	883.52	353.92

In the case of Kv values, there's not much change or variation in the trajectory error itself, as it only seems to affect the time taken to accomplish the goal or the

mean speeds. So, in this case, the optimal value should be between 2 and 5, as depending on the Kh value selected, these results may vary.

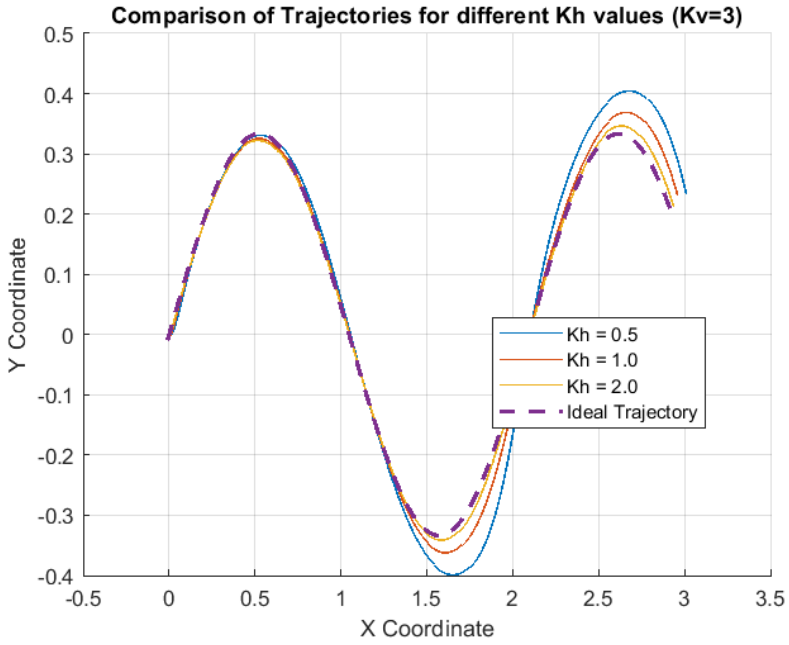

Figure 36 - Pure Pursuit algorithm trajectories for Kh comparison.

Table 8 - Kh values and statistics comparison for Pure Pursuit.

Kh values (Kv=3, L=0.1)	0.5	1	2
Mean error (m)	1.73e-2	9.28e-3	6.00e-3
Standard deviation (m)	2.61e-2	1.13e-2	4.01e-3
Ideal distance (m)	3.6498	3.5832	3.5532
Real distance travelled (m)	3.7903	3.6285	3.5565
Mean Speed (cm/s)	3.5378	3.4613	3.4281
Time to Target (s)	107.2	104.869	103.808

The Kh value selection is more important here, for high Kv values such as the chosen one (Kv=3), the robot moves quite fast, so it will be important to make the steering responsive and its gain should be capable of handling the sharp turns. Low values such as 0.5, don't give high mean error outputs, but as we can see on

the graph, when the robot is moving faster, it doesn't respond well to sharp turns, and gets a bit deviated from the ideal trajectory.

On the other hand, higher values like 2, get the least error and deviation, similar distance travelled and the least time to target. So again, A higher value for the steering controller is the right choice.

Once the effects of speed and steering controllers have been analysed, it's time to sample the Lookahead factor (L), which determines the distance from the robot, in which the pursuit point is generated. Higher values usually provide smooth trajectories, but the movement described will be too different to the one desired.

On the other hand, low values provide trajectories similar to the ideal ones, but the movement is usually more unstable, erratic, or oscillating.

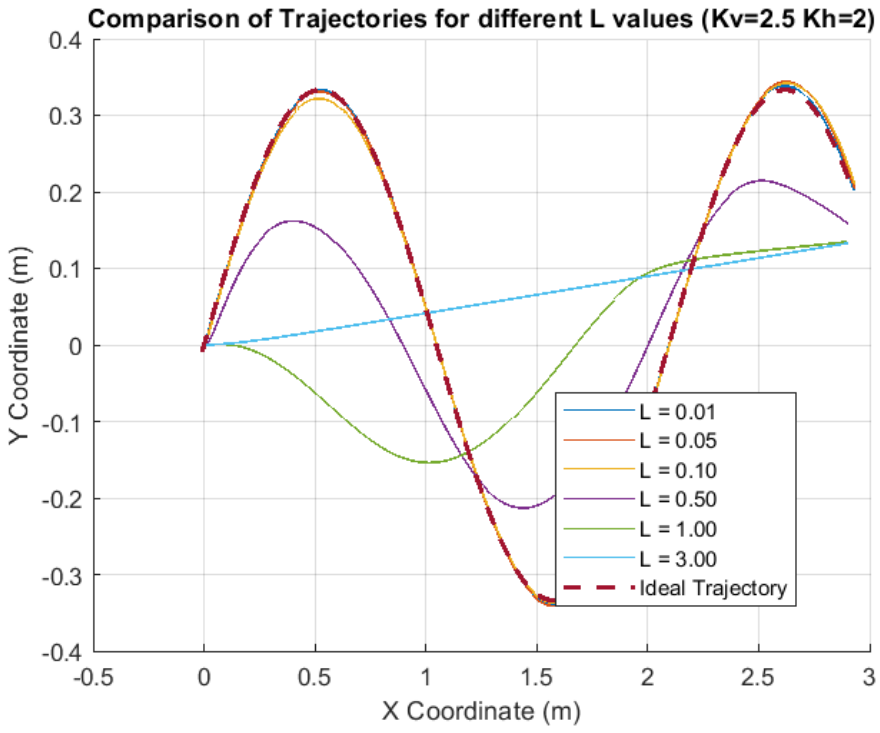

Figure 37 - Pure Pursuit algorithm trajectories for L comparison.

Table 9 - L values and statistics comparison for Pure Pursuit.

Kh values (Kv=2.5, Kh=2)	0.01	0.05	0.1	0.5	1	3
Mean error (m)	3.97e-4	1.93e-3	5.53e-3	0.10	0.21	0.20
Standard deviation (m)	8.31e-4	1.92e-3	3.42e-3	5.69e-2	0.11	0.11
Ideal distance (m)	3.5410	3.5473	3.5478	3.5121	3.5099	3.5108
Real distance travelled (m)	3.5508	3.5618	3.5478	3.1383	2.9649	2.9165
Mean Speed (cm/s)	1.04959	2.0055	2.8882	7.5555	11.0301	12.5540
Time to Target (s)	338.368	177.664	122.752	41.600	26.944	23.296

As just described, the trajectories vary a lot depending on this factor. Values such as L=3, provide almost a straight line to an endpoint (it turns out that the ideal trajectory being sampled ends on x=3, so this lookahead factor, just drives the robot to the end point of the trajectory. Values higher than 3, will provide the same output).

On the other hand, lower values present a really precise and ideal trajectory, but providing high travel times and low speeds.

L=0.1 seems like the optimal value, as it provides a really accurate trajectory, with low mean error and deviation, a similar travelled distance, and an acceptable time to target (meaning also medium-high values for speed).

It's also important to mention that the ideal trajectory has been sampled over the same X values than the trajectory they were being compared with, in order to get the error calculations done properly (that's why each of the ideal distances is a bit different from each other).

Finally, we can conclude that the best values for this kind of controller, in this simulation case, will be Kv=[3, 5], Kh=2 and L=0.1. Which will be useful in further

implementations, where some algorithm-generated trajectories will be needed to follow accurately.

5.3. Bug2 algorithm

Once these simple movement algorithms have been analysed, it's time to get into sensor-based navigation, starting with Bug2 algorithm.

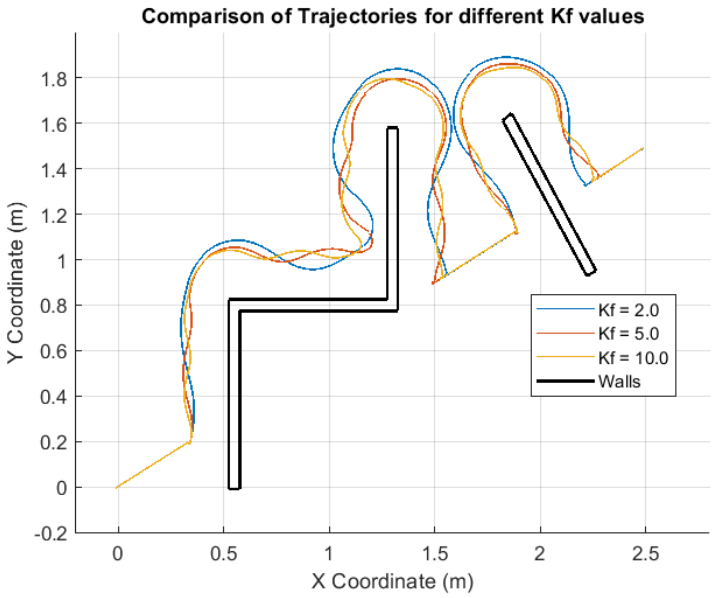

Figure 38 - Trajectory comparison for Kf values in Bug2 algorithm.

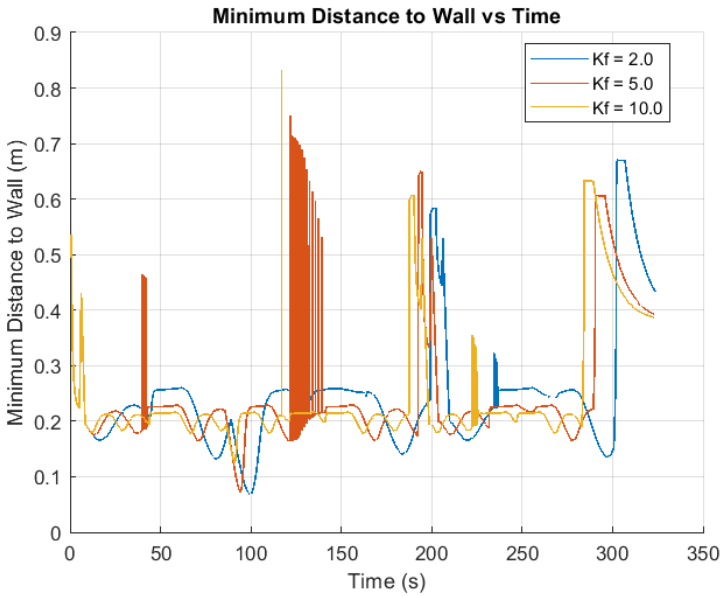

Figure 39 - Minimum distance to wall vs time graph for different Kf values.

Table 10 - Kf values and statistics comparison for Bug2 algorithm.

Kf values	2	5	10
Mean speeds (cm/s)	1.9641	1.9757	1.9639
Time to Target (s)	342.336	328.576	323.392
Minimum distance to wall (cm)	6.890	7.303	12.589
Mean distance to wall (cm)	24.656	24.464	24.749
Collisions	0	0	0
Target achieved	yes	yes	yes

These are the outputs for the Bug2 algorithm tested over the first map. The graphs and table show the results for different values of Kf, which is the proportional gain for the controller which maintains the distance between robot and wall, while wall following an obstacle.

On Figure 38 we can see the different trajectories described depending on the value of the Kf controller. Each of the state machine's state is also visible, as the straight line to the objective, wall following and the re-encounter with m-line can be easily recognised. We can also see how each value changes the oscillation of the

robot while following the wall, although it's true that the walls are not too long, and the wall following controller can't really stabilize in a little matter of time.

On Figure 39 the distance to the closest wall is plotted. There are some fuzzy values for Kf=5 specially, those result from the turns at the edges of the walls, where the LiDAR sometimes gets the wall out of range, getting a peak on the distance. The "sinusoidal-like" parts represent the wall follow state, then the peaks and distortions usually plot the turns at the wall's edges.

Over the statistics plotted, we can see that the output for each value is quite similar, there's no collisions and target is achieved for all of this values, while for

example, lower values return some collisions, even some of them make the target unreachable, getting the robot stuck in some wall corner.

So, the conclusion, or value with the best result here, should be between 5 and 10, as they are faster to achieve the goal, oscillation is not excessive, and minimum distances to walls are large enough to be acceptable and safe.

On the other hand, we have the second map implemented. This one aim to show and test the nature of this algorithm, and in which obstacle configurations it does work well.

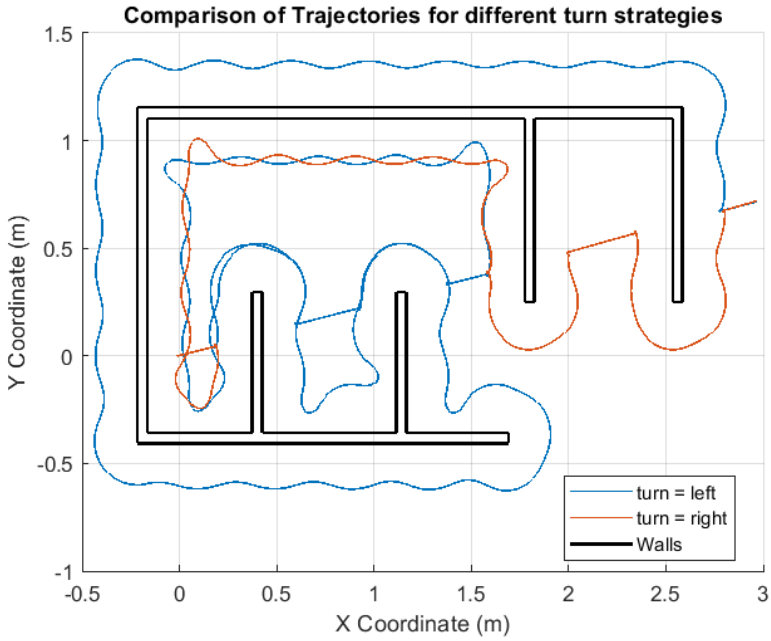

Figure 40 - Turn side comparison for Bug2 algorithm.

Table 11 - Turn side statistics comparison.

Turn	left	right
Mean speeds (cm/s)	1.9726	1.8709
Time to Target (s)	989.312	390.656
Collisions	0	0
Target achieved	yes	yes

In this case, depending on which side turning is coded on the algorithm, it will get to the objective on an efficient way, or not, as it would have to go all around the obstacle in order to get too the target point.

Here as the walls are larger, we can appreciate the oscillation of the wall following controller (in this case Kf=7). This could be reduced without affecting the speed too much, implementing an integrator, or even also a derivative action, controlling the system with a PID controller.

5.4. Artificial Potential Fields

This algorithm was one of the main challenges in this project, as it involves a difficult tuning task with attractive and repulsive field constants, in order to achieve an optimal performance. Despite having done meticulous and exhaustive

experimentation and adjustments, a lot of problems were encountered in fine-tuning these.

The importance of this lies in finding the adequate balance between both fields, achieving smooth navigation towards the goal, while ensuring a safe and robust avoidance behaviour.

Despite the efforts, the implementation of this APF algorithm did not output the desired results, as we can see on Figure 41. The robot kept on colliding with the obstacles, with all values inputted during experimentation.

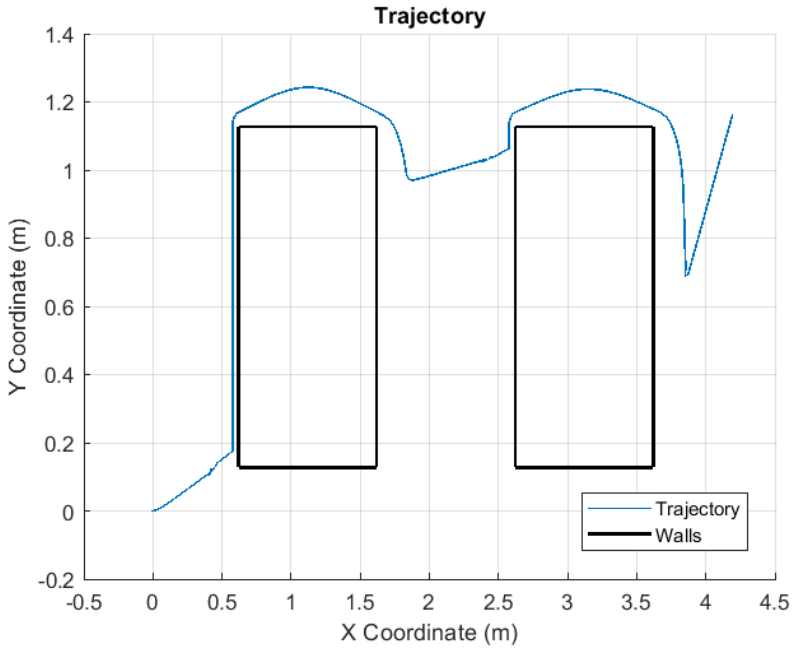

Figure 41 - APF algorithm failed trajectory.

As displayed on the graph, the robot orientates towards the goal point, and when getting close to the obstacle, it tries to avoid it, represented on the little rattling when getting near to the obstacle. It does not achieve it, and crashes with the obstacle, sliding over it until reaching the top. The problem may be on the obstacle

recognition algorithm implementation, as it may not output the obstacle's position correctly.

5.5. Grassfire algorithm

This section presents the results obtained from the grassfire algorithm, a classical path-finding algorithm widely used in robotics and navigation. Analysing this algorithm only involves checking if the maze is solved, and how much time does it take to calculate.

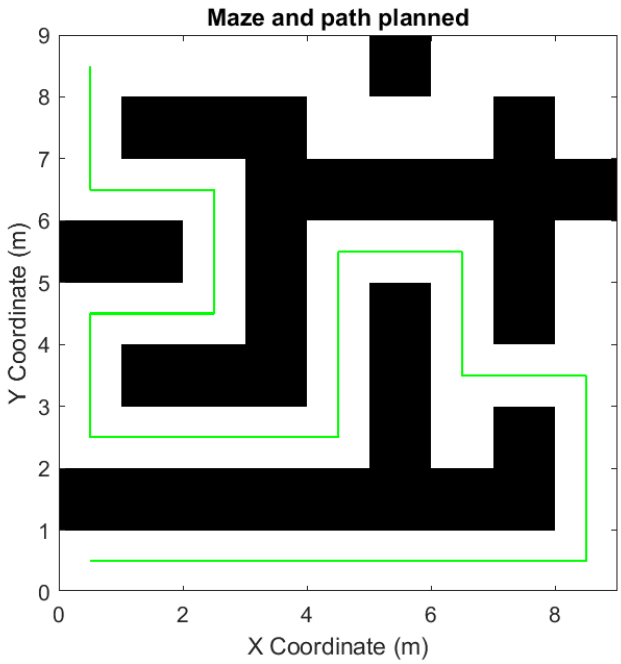

Figure 42 - Grassfire algorithm maze solution output.

Table 12 - Time to calculate Grassfire maze solution.

Time to calculate (s)	0.059236 s

Once calculated, the Pure Pursuit algorithm is applied, obtaining the movement through all the maze without any problem.

5.6. Probabilistic roadmaps generation

It's interesting to analyse this algorithm before getting into search algorithms (Dijkstra and A*). This probability roadmap technique generates a graph that will then be explored by both Dijkstra and A* algorithms, in order to search for optimal paths.

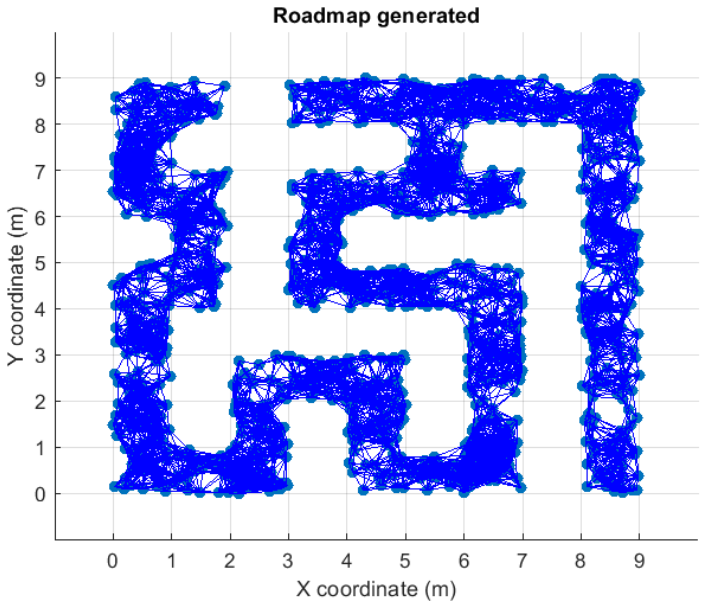

Figure 43 - PRM graph generated for maze arena.

Table 13 - Maze graph calculation statistics.

Generated points	2000
Valid nodes	1202
Generated edges	12474
Generation time (s)	0.1573

For the first maze case, the algorithm seems quite efficient and accurate. All nodes and edges can be clearly distinguished in Figure 43, and how do they fit around the maze's walls. The algorithm is also time efficient, with a generation time of 0.1573 seconds, quite acceptable.

For the second map, we have this output:

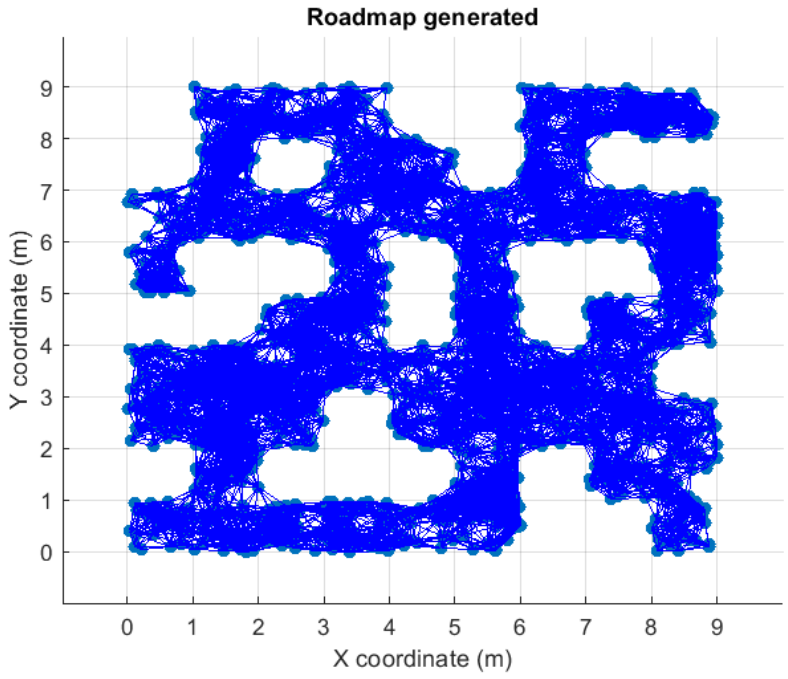

Figure 44 - Graph generated by PRM algorithm for search algorithms second test.

Table 14 - Test arena 2 graph generation statistics.

Generated points	2500
Valid nodes	1698
Generated edges	24104
Generation time (s)	0.32754

Again, the roadmap is successfully generated. In this case, there's some more points being generated, as the map is a bit more complex, and has a widespread variety of obstacles. Again, all roadmaps and nodes are shown in Figure 44, and time taken to generate those is acceptable, less than half a second.

5.1. Dijkstra and A* algorithms

In this part, results from the evaluation and comparison of this two widely used pathfinding algorithms will be presented. These algorithms are really efficient and accurate in path

searching over graphs like the ones generated by the probability roadmap algorithm showed above.

The first test, was based on a simple maze:

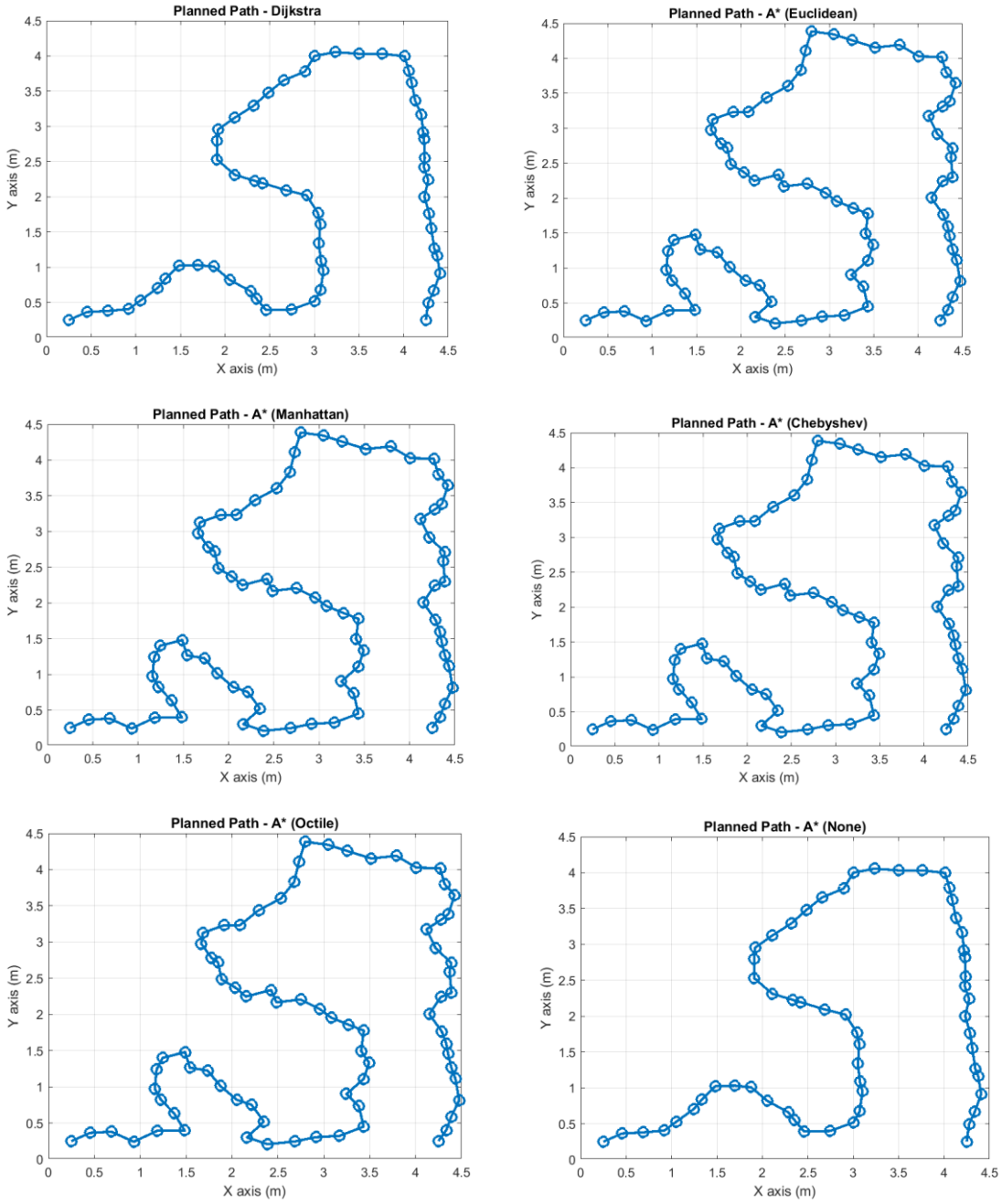

Figure 45 - Maze graph path output from Dijkstra's and A heuristic algorithms.*

Table 15 - Search algorithms output statistics.

Method	Dijkstra	A* (Euclidean)	A* (Manhattan)	A* (Chebyshev)	A* (Octile)	A* (None)
Searching time (s)	0.087503	0.14497	0.078754	0.16201	0.12418	0.088238
Path cost (m)	25.7441	34.8839	34.8839	34.8839	34.8839	25.7441

Knowing how do this algorithms work, it's easy to see that there's something not making sense on these results. A* star algorithm is a complete and effective algorithm, that always finds the shortest path if there's any. In this case, A* is searching and finding a path (every different heuristic is doing so), but Dijkstra's algorithm is always outputting a shorter path, meaning that the ones found by A* are not optimal, when they should be at least equal.

Observing the case in which the A* algorithm is run, but without any heuristic function (this makes the algorithm run exactly as Dijkstra's), gives a clue on where the error may be. This one outputs just the same result (even almost the same search time), as Dijkstra's, what shows that the A* algorithm itself is well set. The problem may be then on the heuristic's implementation.

On Figure 45, we can clearly see the difference between paths. The A* outputs are all the same, but clearly more erratic and larger than the others. Despite all of this, all of these

combinations make the robot work and go from star point to target without any problems, using the Pure Pursuit algorithm over the calculated paths.

For the second test, results don't really vary at all:

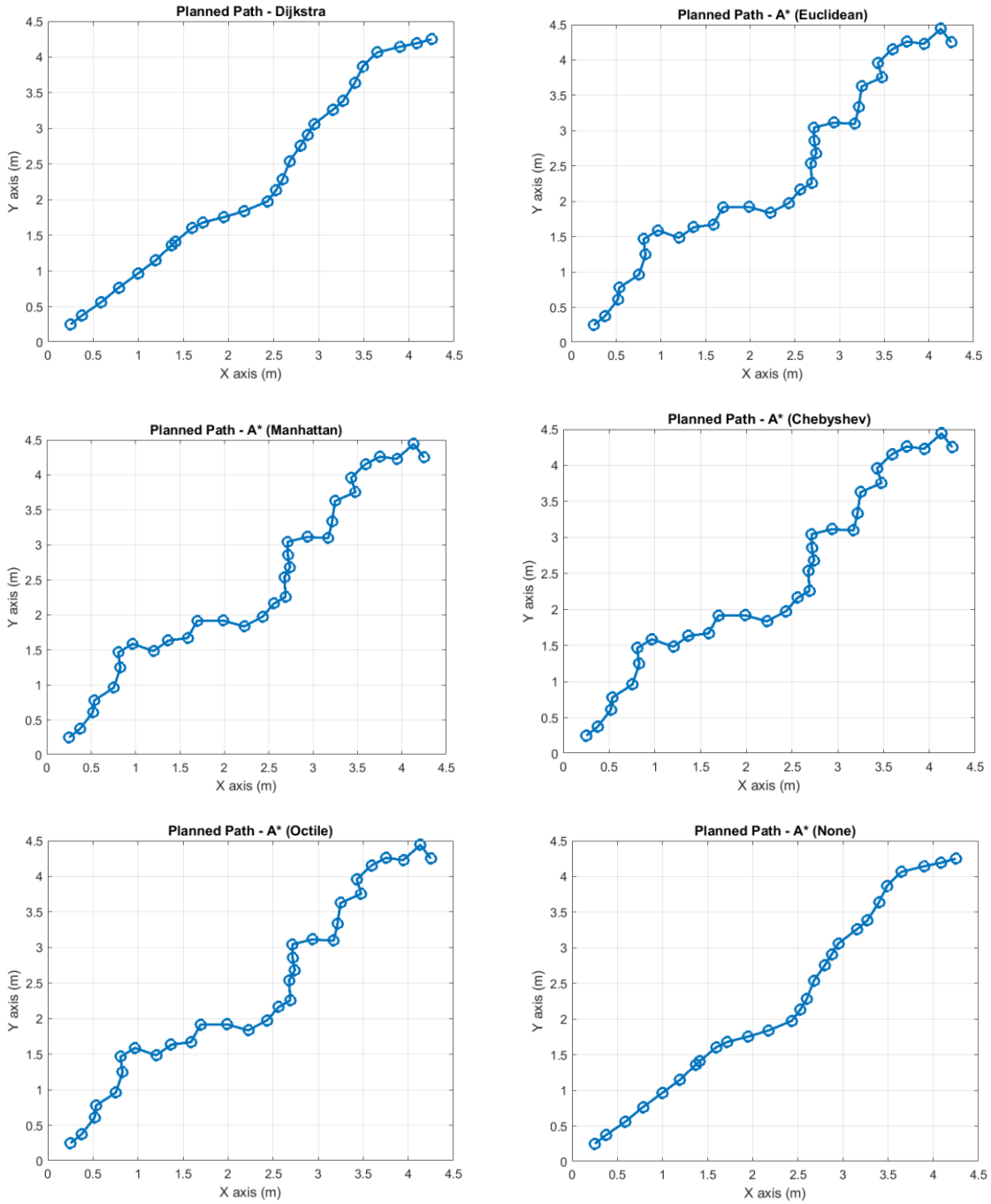

Figure 46 - 2nd test arena path output from Dijkstra's and A heuristic algorithms.*

Table 16 - 2nd test arena output statistics for search algorithms.

Method	Dijkstra	A* (Euclidean)	A* (Manhattan)	A* (Chebyshev)	A* (Octile)	A* (None)
Searching time (s)	0.21534	0.17455	0.12777	0.1514	0.10555	0.12409
Path cost (m)	11.8171	14.4198	14.4198	14.4198	14.4198	14.4198

In this case, all A* outputs seem more efficient timewise, as they have lower calculation times. But again, the path found is not as optimal as Dijkstra's, and using no heuristic function gives again the same output as this one, giving again the same conclusion, there is something wrong with the heuristics (although they may be doing something to guide the algorithm, as times are quite shorter than Dijkstra's this time).

5.1. RRT (Rapidly-Exploring Random Tree)

Last algorithm being tested is RRT, a path generating algorithm that grows a random point tree from a starting point, until reaching (or not), the goal point. The implementation has 2 variables: number of iterations and distance between points generated. Depending on these parameters, trees will have longer or shorter links (distance between nodes) and will find or not a feasible

path (depends on the number of iterations inputted, and if that is enough or not). This algorithm was teste against the same maze used on some sections above.

For a same number of iterations (5000 should be enough), some distances are sampled:

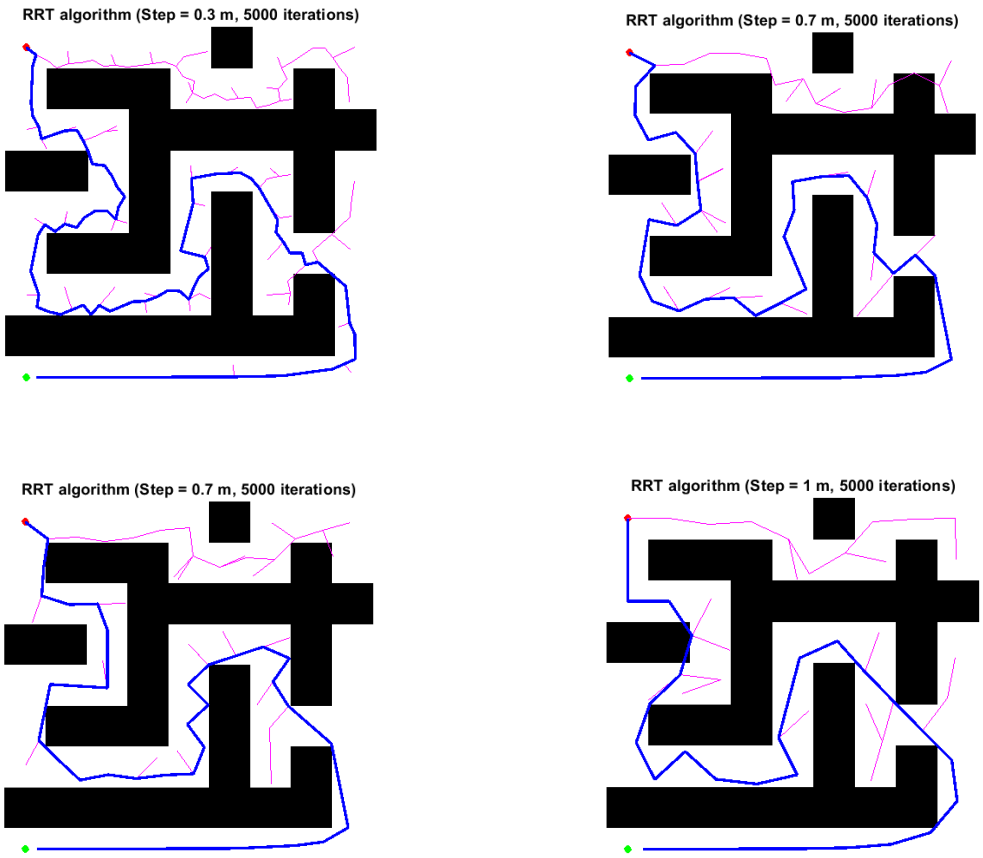

Figure 47 - RRT algorithm paths for different distance random points generation.

Table 17 - RRT algorithm output statistics for different distance generated points.

Max distance between points (m)	0.1	0.3	0.5	0.7	1
Searching time (s)	0.10587	0.08958	0.03802	0.02139	0.01785
Iterations needed	5000	3754	2651	1875	1361
Target reached	no	yes	yes	yes	yes

To make the 0.1 distance value find a viable path, we need to increment the number of iterations, e.g. 20000 iterations:

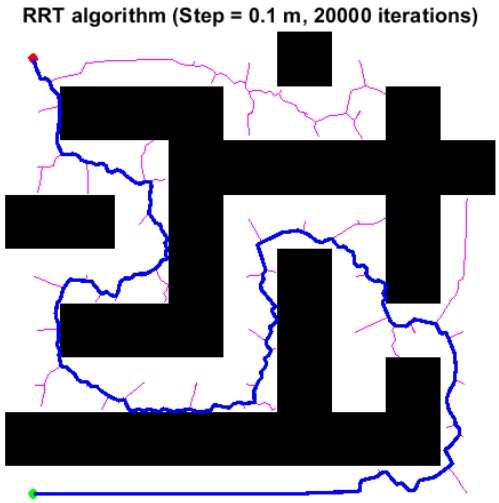

Figure 48 - RRT path for 0.1 m distance generation.

Table 18 - RRT output statistics for high iterations case.

Max distance between points (m)	0.1
Searching time (s)	0.27813
Iterations needed	10410
Target reached	Yes

This algorithm also outputs viable paths to follow. Its nature makes it really random, for example, in this case to output a 0.1m distanced nodes path, 20000 iterations were needed, while next time executed, the code may need much less, or many more.

We can also distinguish how do paths change, depending on the distance used. In cases where long distances are used, the path even goes through some of the wall's vertexes, as the Bresenham algorithm used, is not precise enough.

6. Conclusion and future work

In this project, a diverse range of algorithms has been explored, implemented, and analysed. Algorithms for robotics path planning and navigation including go to a point, Pure pursuit, Bug algorithms, Artificial Potential Fields; path searching algorithms such as Grassfire, Dijkstra, and A*; or graph generating techniques like Probabilistic Roadmaps (PRMs) and Rapidly-Exploring Random Trees (RRT). Each algorithm presented a unique behaviour, with their strengths and limitations, offering some valuable insight into this gigantic world of robotics and its algorithms.

This experimentation revealed some key findings or challenges:

- Algorithm's performance: a valuable analysis of different algorithms, and its internal parameters and variables, which tuning turned out to be essential in order to achieve good performances. Some algorithms such as the move to a point, Pure Pursuit, Bug2 or Grassfire, turned out to be robust and efficient in some way; while others such as APF, which proved challenging to tune

effectively, or A*, which results discovered some errors or malfunctions on the heuristics implementation.

- Simulation tools: the project has also given a deep analysis and use of the Webots platform, discovering a “user-friendly” interface with a really high simulation power and precision. These results should also be tried to reproduce in a real-life application, in order to test the scalability of this kind of tools.
- Computational efficiency: these kinds of realistic and precise simulations, are usually really demanding computationally talking. This project has also given a valuable insight into the need to optimise the coding of each algorithm.
- Sensitivity to environment changes: most of the algorithms have proved to work nicely on each of its test arenas, but are they robust enough to work as well on other more complex or completely different scenarios? Some other implementations like the APF have proven to be really scenario-dependant.

Building upon all these findings and experimentation, a good basis for future research and development can be stated:

- Real-world applications: transitioning to real world deployment and testing, addressing some challenges mentioned like noise, hardware limitations, environment variability or power consumption would be the next logical step.

- APF algorithm refinement or redesign: due to the failure to achieve the functioning of this algorithm, exploring an alternative approach or redesign the parameter tuning and optimization would make sense. Including some more investigation of more advanced control techniques, machine learning or others, to enhance the APF algorithm implemented.

- A* heuristics review: conducting the data analysis of this algorithm, showed some limitations and inaccuracies that would impact the optimal path search.

This review should include experimenting with different heuristic functions and refining the ones already implemented.

- General code efficiency and redesign: a complete code review is always advantageous in order to achieve computationally efficient and fast results. Also improving the code's readability and maintainability will enhance its performance.

- Path refinement: implementing some post-processing techniques to paths generated by search algorithms (Dijkstra, A*, RRT) such as smoothing techniques or obstacle avoidance manoeuvres form example. These would improve the path's quality, reduce unnecessary turns causing energy waste or enhance overall navigation efficiency.

- Replanning: implementing dynamic replanning strategies, particularly for cases like A* algorithm, when encountering unexpected obstacles or changes in the environment.

- Sensor integration and data collection: sensors are always theoretically fabulous, but when getting into a real application, noise or error are always present. Refining the sensor inputs or data collected would mitigate errors and uncertainties of real-world applications. Implementing filtering or sensor-fusion algorithms would improve data's quality and reliability.

- Machine learning: investigating the integration of these techniques, such as reinforcement learning or deep learning into path planning algorithms could enable more adaptive and flexible navigation strategies capable of handling uncertain situations.

- Research on more techniques and algorithms: as mentioned on the literature review there's plenty of other algorithms or improvements for the ones developed on this project.

7. Potential client companies

There's a wide range of companies that would be interested in this kind of research. Robotics and their control algorithms are truly relevant in the actual engineering's landscape, demanding the research and development of these control techniques for lots of products and applications.

There are many companies which this project would suit to, depending on the engineering area they focus on, some examples are given:

- Research institutions: MIT Robotics, many other universities.
- Robotics companies: Boston Dynamics, iRobot.
- Autonomous vehicles companies: Tesla, many other manufacturers working on these technologies.
- Healthcare robotics: Intuitive Surgical, Hansen Medical.
- Drone and UAV manufacturers: DJI, AeroVironment.
- Logistics and warehouse: Amazon, KUKA.
- Government agencies: NASA, numerous defence departments.

Here there are three of the most interesting options:

- iRobot: This company is a pioneer in consumer and home robotics, very well known for some products lime the Roomba vacuum cleaner. This project aims and research, completely aligns with the development of this kind of products, and the challenges faced when implementing these in real-world environments.
- This company could be interested in the project as these kinds of algorithms are the base of home robots like this vacuum cleaners, and how they manage to follow particular paths or cover complete areas with precision. Also,

understanding each solution strengths and weaknesses would help to enhance their technologies and overall performance.

- Tesla: This company's big focus is on autonomous vehicles, so the project involving navigation, route planning or localization algorithms would be relevant in this. Also, the project's emphasis on simulation and analysis of data and results aligns with what autonomous cars or systems need in its early development being made now. Analysis the performance of some algorithms on complex scenarios and performing accurate and realistic simulations on some kind of software would also be a great aspect that would be on this company's interest.

- Amazon: Amazon's logistics and on-time deliveries are well known worldwide. This is also possible because of how they manage and control their products on warehouses. The project's focus on indoor navigation, planning and obstacle avoidance algorithms are a particularly relevant task for Amazon's engineering teams, as their logistics robots need to do these tasks accurately and efficiently.

The value on this project could be found on the development of this algorithms focused on indoor environments, in order to enhance Amazon's robotic systems in warehouse settings.

Project planning

Task	OCT	NOV	DEC	JAN	FEB	MAR	APR	MAY
Background								
A&O								
Introduction section								
Literature review								
WeBots software training								
Design								
Simple movement algorithms design								
Reactive Navigation algorithms design								
Map-Based Algorithms design								
Map generation algorithms design								
Implementation								
Coding algorithms								
Simulations set up								
Testing								
Performing simulations								
Results analysis								
Analysing data from simulations								
Report Writing								
Poster design								
Conclusions writing								

Figure 49 - Gantt chart illustrating the project's tasks and planification.

References

Choset, H., Hager, G. & Dodds, Z., 2010. Robotic Motion Planning: Bug Algorithms slides. s.l.:CMU School of Computer Science.

Choset, H. M., 2005. Principles of robot motion theory, algorithms, and implementation. s.l.:Cambridge Mass, MIT Press.

Corke, P., 2017. Robotics, Vision, and Control. 2nd ed. s.l.:Springer.

Farley, A., Wang, J. & Marshall, J. A., 2022. How to pick a mobile robot simulator: A quantitative comparison of CoppeliaSim, Gazebo, MORSE and Webots with focus on accuracy of motion. Simulation modelling practice and theory, 120(102626).

Gonçalves, P. J., Paulo J.D. Torres, C. M. A. & Mondada, F., 2009. The e-puck, a Robot Designed for Education in Engineering. Proceedings of the 9th Conference on Autonomous Robot Systems and Competitions, Volume 1.

McGuire, K., Croon, G. d. & Tuyls, K., 2018. A comparative study of bug algorithms for robot navigation. CoRR.

Muñoz, N. D. & Valencia, J. A., 210. Quantitative metrics for Mobile Robots Navigation. Politécnico Colombiano Jaime Isaza Cadavid, University of Antioquia.

Universidad de Oviedo, n.d. Intensificación en Sistemas robóticos, Industrial Engineering Master. s.l.:s.n.

Yufka, A. & Parlaktuna, O., 2009. Performance Comparison of the BUG's Algorithms for Mobile Robots.

Annexes

All code and simulated worlds have been uploaded to GitHub:

Webots Based Implementation and Simulation of Robotics Algorithms GitHub