
Aprendizaje Automático, Computación Evolutiva e Inteligencia de Enjambre para Aplicaciones de Robótica

Gabriela Iriarte Colmenares



UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería



Aprendizaje Automático, Computación Evolutiva e
Inteligencia de Enjambre para Aplicaciones de Robótica

Trabajo de graduación presentado por Gabriela Iriarte Colmenares para
optar al grado académico de Licenciada en Ingeniería Electrónica

Guatemala,

2021

Vo.Bo.:

(f) _____
Dr. Luis Alberto Rivera Estrada

Tribunal Examinador:

(f) _____
Dr. Luis Alberto Rivera Estrada

(f) _____
MSc. Miguel Enrique Zea Arenales

(f) _____
MSc. Carlos Alberto Esquit Hernández

Fecha de aprobación: Guatemala, 12 de enero de 2021 .

Prefacio

La elaboración de esta tesis surgió del interés personal en el área de inteligencia computacional. La inteligencia artificial es un tema que considero que está presente en muchos ámbitos de la vida cotidiana, haciéndola más fácil y amena. Hoy en día contamos con aparatos que reconocen la voz o las canciones al instante, cuando antes era necesario comprar el CD de su artista preferido para poder escucharlo. Desde pequeña sabía que quería ser ingeniera, aunque no estaba muy segura en qué, sí estaba segura de que quería estudiar algo relacionado con tecnología. Este trabajo abarca áreas como la robótica, sistemas de control, computación paralela, inteligencia computacional, evolutiva y de enjambre, por lo que, definitivamente, cumplí mi sueño de estudiar algo tecnológico.

Considero pertinente agradecer a Dios, al comité de ayuda financiera, al programa de becas Potencia-T de la SEEA y a mi madre por brindarme la oportunidad de estudiar en la Universidad del Valle de Guatemala. Además quiero agradecer a mi asesor de tesis Dr. Luis Alberto Rivera por compartir conmigo su conocimiento y brindarme su tiempo para resolver mis cincuenta mil dudas semanales, y a MSc. Miguel Zea por proveerme un poco de su tiempo para darme retroalimentación útil y compartirme sus conocimientos de robótica para esta tesis. Asimismo, quisiera agradecer a todas las personas que he conocido durante mi estudio en UVG, pues todas las memorias compartidas con ustedes impactaron mi vida de una u otra forma.

Debido a la actual coyuntura mundial, el presente trabajo quiero dedicarlo a todos los médicos que valientemente salen a trabajar en plena pandemia por el bien común, y a las víctimas de la pandemia COVID-19, especialmente a todos los médicos que no lograron la batalla.

Índice

Prefacio	III
Lista de figuras	X
Lista de cuadros	XII
Resumen	XIII
Abstract	XIV
1. Introducción	1
2. Antecedentes	2
2.1. Megaproyecto Robotat Fase II	2
2.2. Robotarium de Georgia Tech	3
2.3. Wyss Institute Swarm Robots	3
2.4. Aplicaciones de Ant Colony Optimization (ACO)	3
3. Justificación	4
4. Objetivos	5
4.1. Objetivo general	5
4.2. Objetivos específicos	5
5. Alcance	6
6. Marco teórico	7
6.1. Inteligencia Computacional de Enjambre	7
6.2. Particle Swarm Optimization (PSO)	7
6.2.1. Funcionamiento del algoritmo PSO	8
6.2.2. Mejora del algoritmo	8
6.3. Ant Colony Optimization (ACO)	9
6.3.1. Simple Ant Colony Optimization (SACO)	9
6.3.2. Ant System	11

6.4.	Computación evolutiva	11
6.4.1.	Algoritmos Genéticos (GA)	12
6.4.2.	Métodos de codificación	12
6.4.3.	Función de costo	13
6.4.4.	Métodos de selección proporcional	14
6.4.5.	Operadores de crusa en GA	15
6.4.6.	Métodos de mutación	16
6.5.	Grafos	16
6.5.1.	Grafos en Matlab	17
6.6.	Programación orientada a objetos	17
6.7.	Computación paralela	18
6.7.1.	Programación paralela en Matlab	18
6.8.	Planificación de movimiento	19
6.8.1.	Espacio de configuración	19
6.8.2.	Planificación de trayectorias	19
6.9.	Métodos de planificación de movimiento	20
6.9.1.	Métodos completos	20
6.9.2.	Métodos de cuadrícula	20
6.9.3.	Métodos de muestreo	20
6.9.4.	Artificial Potential Fields (APF)	21
6.10.	Modelado de robots móviles	22
6.10.1.	Modelo uniciclo	23
6.10.2.	Modelo diferencial	23
6.10.3.	Difeomorfismo	24
6.11.	Controladores de posición y velocidad de robots diferenciales	24
6.11.1.	Control proporcional de velocidades con saturación limitada	24
6.11.2.	Control PID de velocidad lineal y angular	25
6.11.3.	Control PID de acercamiento exponencial	26
6.11.4.	Control de pose	26
6.11.5.	Control de pose de Lyapunov	26
6.11.6.	Control Closed-Loop Steering	26
6.11.7.	Control LQR	27
6.11.8.	Control LQI	27
7.	Diseño experimental Ant System	28
7.1.	Validación del algoritmo Ant System	28
7.2.	Validación del algoritmo Ant System paralelizado	30
7.3.	Ant System paralelizado con PRM	33
7.4.	Ant System paralelizado con RRT	33
7.5.	Ant System con grafo de visibilidad	33
7.6.	Elección de parámetros	34
7.7.	Efecto de la paralelización	35
8.	Resultados Ant System	36
8.1.	Validación del algoritmo Ant System	36
8.2.	Validación del algoritmo Ant System paralelizado	37
8.3.	Ant System paralelizado con PRM	38
8.4.	Ant System paralelizado con RRT	38

8.5. Ant System con grafo de visibilidad	39
8.6. Elección de parámetros	42
8.6.1. Rho	42
8.6.2. Alpha	43
8.6.3. Beta	47
8.6.4. Q	51
8.6.5. Número de hormigas	55
8.7. Efecto de la paralelización	56
8.8. Análisis de resultados	58
9. Controladores punto a punto	59
9.1. Simulación por Software	59
9.1.1. Webots	60
9.1.2. Robot e-puck	60
9.2. Control TUC	62
9.3. Control PID de velocidad lineal y angular	63
9.4. Control PID de acercamiento exponencial	64
9.5. Control de Pose	65
9.6. Control de pose de Lyapunov	66
9.7. Control Closed-Loop Steering	67
9.8. Control LQR	68
9.9. Control LQI	69
10. Ant System controlado	70
10.1. Controladores simples en mundo cuadriculado	70
10.1.1. Control TUC	71
10.1.2. Control de pose	72
10.1.3. Control de pose de Lyapunov	73
10.1.4. Control LQR	74
10.1.5. Control LQI	75
10.2. Controladores modificados en mundo cuadriculado	76
10.2.1. Control TUC	77
10.2.2. Control de pose	78
10.2.3. Control de pose de Lyapunov	79
10.2.4. Control LQR	80
10.2.5. Control LQI	81
11. Algoritmos Genéticos (GA)	82
11.1. Minimizando funciones de costo	82
11.2. Resultados	83
12. Conclusiones	85
13. Recomendaciones	87
14. Bibliografía	89
15. Anexos	92
15.1. Especificaciones de computadora de alto desempeño	92

Lista de figuras

1.	Ejemplo de un grafo [7].	10
2.	Gráfico y función de Rosenbrock [14].	13
3.	Gráfico y función de Ackley [14].	13
4.	Gráfico y función de Rastrigin [14].	14
5.	Gráfico y función de Booth [14].	14
6.	Selección de la ruleta en forma gráfica para 6 individuos.	14
7.	Computación serial [25]	18
8.	Computación paralela [25]	18
9.	Ejemplo de grafo de visibilidad en Matlab.	20
10.	Ejemplo de un RRT [31]	21
11.	Ejemplo de un PRM en Matlab.	21
12.	Ejemplo de un APF en Matlab.	22
13.	Distancia recorrida de una rueda para el robot móvil.	22
14.	Piscina paralela en Matlab.	32
15.	Aplicación que genera grafos de visibilidad.	34
16.	Camino óptimo encontrado del nodo (1,1) al (6,6).	36
17.	Feromona del camino óptimo encontrado del nodo (1,1) al (6,6).	37
18.	Animaciones generadas con mundo cuadriculado.	38
19.	Animaciones generadas con PRM.	39
20.	Animaciones generadas con RRT.	39
21.	Animaciones generadas con grafo de visibilidad.	40
22.	Animaciones generadas con grafo de visibilidad.	41
23.	Costo para cada valor de ρ	42
24.	Tiempo para cada valor de ρ	43
25.	Costo del barrido de α para $\rho = 0.6$	44
26.	Tiempo para cada valor de α para $\rho = 0.6$	44
27.	Costo del barrido de α para $\rho = 0.4$	45
28.	Tiempo para cada valor de α para $\rho = 0.4$	45
29.	Costo del barrido de α para $\rho = 0.8$	46
30.	Tiempo para cada valor de α para $\rho = 0.8$	46
31.	Costo del barrido de β para $\rho = 0.6$	47

32.	Tiempo para cada valor de β para $\rho = 0.6$	48
33.	Costo del barrido de β para $\rho = 0.4$	49
34.	Tiempo para cada valor de β para $\rho = 0.4$	49
35.	Costo del barrido de β para $\rho = 0.8$	50
36.	Tiempo para cada valor de β para $\rho = 0.8$	50
37.	Costo del barrido de Q para $\rho = 0.6$	51
38.	Tiempo para cada valor de Q para $\rho = 0.6$	52
39.	Costo del barrido de Q para $\rho = 0.4$	53
40.	Tiempo para cada valor de Q para $\rho = 0.4$	53
41.	Costo del barrido de Q para $\rho = 0.8$	54
42.	Tiempo para cada valor de Q para $\rho = 0.8$	55
43.	Costo del barrido de hormigas para $\rho = 0.6$	56
44.	Tiempo para cada valor de hormigas para $\rho = 0.6$	56
45.	Barrido de núcleos para AS.	58
46.	Ángulos en Webots.	60
47.	Ángulos del e-puck.	61
48.	Posición inicial de robot para pruebas con controladores.	61
49.	Gráficas de controlador TUC.	62
50.	Gráficas de controlador PID.	63
51.	Gráficas de controlador PID de acercamiento exponencial.	64
52.	Gráficas de controlador de pose.	65
53.	Gráficas de controlador de pose de Lyapunov.	66
54.	Gráficas de controlador Closed-Loop Steering.	67
55.	Gráficas de controlador LQR.	68
56.	Gráficas de controlador LQI.	69
57.	Gráficas de controlador TUC.	71
58.	Gráficas de controlador de pose.	72
59.	Gráficas de controlador de pose de Lyapunov.	73
60.	Gráficas de controlador LQR.	74
61.	Gráficas de controlador LQI.	75
62.	Camino interpolado y no interpolado para control de pose.	76
63.	Trayectoria generada con el controlador TUC.	77
64.	Gráfica de velocidad en los motores para controlador TUC.	78
65.	Trayectoria generada con el controlador de pose.	78
66.	Gráfica de velocidad en los motores para controlador de pose.	79
67.	Trayectoria generada con el controlador de pose de Lyapunov.	79
68.	Gráfica de velocidad en los motores para controlador de pose de Lyapunov.	79
69.	Trayectoria generada con el controlador LQR.	80
70.	Gráfica de velocidad en los motores para controlador LQR.	80
71.	Trayectoria generada con el controlador LQI.	81
72.	Gráfica de velocidad en los motores para controlador LQI.	81
73.	Minimización de la función de Rastrigin.	83
74.	Minimización de la función de Rosenbrock.	83
75.	Minimización de la función de Ackley.	84
76.	Minimización de la función de Booth.	84

77. Procesador Intel Xeon Gold [34]	92
-------------------------------------	----

Lista de cuadros

1.	Parámetros del experimento 1.	30
2.	Parámetros utilizados en AS paralelizado.	32
3.	Rango en el que se realizó el barrido de parámetros.	35
4.	Parámetros utilizados en el barrido de núcleos.	35
5.	Tiempo y costo de 10 ejecuciones de AS.	37
6.	Tiempo y costo de 10 ejecuciones de AS paralelizado.	38
7.	Resultados del AS con PRM.	39
8.	Resultados del AS con RRT.	40
9.	Resultados del AS con grafo de visibilidad.	41
10.	Resultados del barrido de ρ	42
11.	Resultados del barrido de α para $\rho = 0.6$	43
12.	Resultados del barrido de α para $\rho = 0.4$	45
13.	Resultados del barrido de α para $\rho = 0.8$	46
14.	Resultados del barrido de β para $\rho = 0.6$	47
15.	Resultados del barrido de β para $\rho = 0.4$	48
16.	Resultados del barrido de β para $\rho = 0.8$	50
17.	Resultados del barrido de Q para $\rho = 0.6$	51
18.	Resultados del barrido de Q para $\rho = 0.4$	52
19.	Resultados del barrido de Q para $\rho = 0.8$	54
20.	Resultados del barrido de hormigas para $\rho = 0.6$	55
21.	Resultados del barrido de núcleos.	57
22.	Parámetros utilizados para el controlador TUC.	62
23.	Parámetros utilizados para el controlador PID.	63
24.	Parámetros utilizados para el controlador PID con acercamiento exponencial.	64
25.	Parámetros utilizados para el controlador de pose.	65
26.	Parámetros utilizados para el controlador de pose de Lyapunov.	66
27.	Parámetros utilizados para el controlador Closed Loop Steering.	67
28.	Parámetros utilizados para el controlador LQR.	68
29.	Parámetros utilizados para el controlador LQI.	69
30.	Parámetros utilizados en la modificación.	76

31. Parámetros que varían según la función de costo.	83
--	----

Resumen

Para que los robots móviles puedan navegar de un inicio a un final es necesario contar con un planificador de trayectorias. En esta investigación se buscó implementar un algoritmo de inteligencia de enjambre como planificador de trayectorias. Dicho algoritmo fue el *Ant System* (AS), también llamado *Ant Colony*, que se basa en el comportamiento de las hormigas al buscar y hallar alimento. La principal motivación para implementar este algoritmo fue compararlo contra el algoritmo desarrollado en la fase anterior del proyecto Robotat. Este último algoritmo es el *Modified Particle Swarm Optimization* (MPSO), que es un algoritmo de inteligencia de enjambre basado en aves y peces. De este modo, se tendría una alternativa de planificación de trayectorias para los Bitbots de UVG.

Para lograr esta comparación se implementaron siete controladores distintos. Luego, se eligió a los cinco mejores para unirlos con el AS. Igualmente, se modificó el camino encontrado por el AS, interpolándolo para crear más metas y lograr una trayectoria suave. Dependiendo de si se busca controlar a muchos o a solo un agente, se recomienda utilizar el MPSO o el AS respectivamente. Asimismo, los controladores con la respuesta más suave de velocidad son el controlador de pose y el controlador de pose de Lyapunov.

Adicionalmente, se realizaron pruebas preliminares con computación paralela y algoritmos genéticos. El primero se utilizó para agilizar el proceso de planificación de trayectorias con el AS. De este modo, se puede aprovechar el hecho de que las computadoras modernas cuentan con más que un solo núcleo. El segundo se utilizó para explorar la posibilidad de utilizar algoritmos genéticos como alternativa al MPSO y el AS.

Abstract

Differential robots need a trajectory planner to be able to move from a source point to an end point. For this reason, this research implemented a Swarm Intelligence algorithm as a trajectory planner. The chosen algorithm was Ant System (AS), which is also called Ant Colony. AS is based on the behavior exhibited by ants when they search for food. The main motivation for implementing this algorithm was to compare it against the one developed on the previous stage of the project. The aforementioned Swarm Intelligence algorithm was a Modified Particle Swarm Optimization (MPSO). Therefore, we would have an alternative path planning algorithm for the UVG Bitbots.

To achieve the comparison, seven controllers were implemented. Subsequently, the best five were chosen to then be merged with AS. Additionally, the path found by AS was modified by an interpolation to create more goals. Therefore, this results in a smooth trajectory. If the application requires to control several robots instead of just one, the MPSO is suggested. If not, then AS with Pose Controller or Lyapunov Pose controller are recommended.

In addition, preliminary tests were made with parallel computation and genetic algorithms. The first one was used to speed up the path planning process with AS. Thus, it's possible to take advantage of the fact that modern computers have more than one core. The second one was used to explore the possibility of using genetic algorithms as an alternative to MPSO and AS.

CAPÍTULO 1

Introducción

En este trabajo se presenta una propuesta de inteligencia de enjambre para aplicaciones de robótica. Una de las motivaciones para realizar este trabajo es que en el futuro los robots puedan utilizarse para aplicaciones de búsqueda y rescate. Principalmente se desea comparar el rendimiento entre el algoritmo *Particle Swarm Optimization* (PSO:) y *Ant System* (o simplemente *Ant Colony*), que son algoritmos de inteligencia de enjambre con distintos enfoques (función de costo vs. búsqueda en grafos). Uno de los objetivos es encontrar cuál de estos dos algoritmos es el mejor, para que en un futuro sea implementado en robots físicos similares a un robot e-puck.

Como metodología, primero se implementó el algoritmo *Simple Ant Colony*, y después el algoritmo *Ant System*. Estos algoritmos difieren en la forma en la que toman la decisión de qué camino seguir. Los parámetros encontrados para el correcto funcionamiento del AS: se validaron por medio de simulaciones computarizadas que permiten la visualización del comportamiento de la feromonía depositada por la colonia. Finalmente, se adaptaron los modelos del movimiento y de la cinemática de los Bitbots, desarrollados en la fase anterior del proyecto, al algoritmo ACO. De esta forma, fue posible implementarlo en el software Webots para comparar su desempeño con el del *Modified Particle Swarm Optimization* (MPSO:).

A continuación se presentan antecedentes del tema, marco teórico necesario para la comprensión de la tesis, diseño experimental y resultados. De estos últimos se muestran los obtenidos para el algoritmo *Ant System* y para una propuesta que se realizó de algoritmos genéticos como alternativa adicional.

CAPÍTULO 2

Antecedentes

2.1. Megaproyecto Robotat Fase II

En la segunda fase del proyecto Robotat¹ [1] se elaboró un algoritmo basado en el *Particle Swarm Optimization* (PSO) para que distintos agentes llegaran a una meta definida (el mínimo de una función de costo). El algoritmo modificado elaboraba una trayectoria a partir de la actualización de los puntos de referencia a seguir, generados por el PSO clásico. Asimismo, se realizaron distintas pruebas para encontrar los mejores parámetros del algoritmo PSO, probando distintas funciones de costo. Para realizar dicho algoritmo se tomó en cuenta que la velocidad de los robots diferenciales tiene un límite, además de tener que seguir las restricciones físicas de un robot real. Para simular estas restricciones se utilizó el software de código abierto Webots, luego de haberlo implementado en Matlab como partículas sin masa ni restricciones. Además, en [2] se realizaron experimentos con el mismo algoritmo, pero utilizando *Artificial Potential Fields* (APF) como forma de evasión de obstáculos.

En Webots también se implementaron distintos controladores para que la trayectoria fuera suave y lo más uniforme posible. Entre los controladores que se implementaron están: Control proporcional de cinemática transformada (TUC), PID con acercamiento exponencial, PID de velocidad lineal y angular, Controlador simple de pose (SPC), Controlador de pose Lyapunov (LSPC), Controlador de direccionamiento de lazo cerrado (CLSC), LQR y LQI. El controlador con el mejor resultado en esta fase fue el LQI, pues generaba trayectorias casi rectas y una pequeña variación en la velocidad de las ruedas del robot.

¹Proyecto de la línea de investigación en robótica de enjambre de la Universidad del Valle.

2.2. Robotarium de Georgia Tech

El Robotarium del Tecnológico de Georgia en Estados Unidos desarrolló una mesa de pruebas para robótica de enjambre para que personas de todo el mundo pudieran hacer pruebas con sus robots. De este modo, ellos están apoyando a que más personas, sin importar sus recursos económicos, puedan aportar a la investigación de robótica de enjambre. Además, eso ayudaría a los investigadores a ir más allá de la simulación y realizar pruebas con sus algoritmos en prototipos reales. Para utilizar la plataforma es necesario descargar el simulador del Robotarium de Matlab o Python, registrarse en la página del Robotarium y esperar a ser aprobado para crear el experimento [3].

2.3. Wyss Institute Swarm Robots

El Instituto Wyss de Harvard desarrolló robots de bajo costo miniatura llamados *Kilobots* para probar algoritmos de inteligencia computacional de enjambre. Este tipo de robots son desarrollados para realizar tareas complejas (potencialmente) en el ambiente como lo son polinizar o crear construcciones. Estos Kilobots cuentan con sensores, micro actuadores y controladores robustos para permitir a los robots adaptarse a los ambientes dinámicos y cambiantes [4].

2.4. Aplicaciones de Ant Colony Optimization (ACO)

En [5] se aplica el método de ACO para planificar la trayectoria de un robot autónomo en una cuadrícula con un obstáculo estático. El robot se coloca en la esquina inferior izquierda y se pretende que alcance el objetivo en la esquina superior derecha de la cuadrícula. La simulación de este trabajó se realizó en Matlab y se implementó con una cuadrícula de 100×100 unidades con la unidad como el tamaño de las aristas. Además, en la cuadrícula los agentes podían moverse norte, sur, este, oeste y en forma diagonal.

También en [6] se realizó una comparación de los algoritmos PSO y ACO, en donde se resaltaron algunas ventajas y desventajas de cada uno de los métodos. El PSO es simple, pero sufre al quedarse atascado con mínimos locales por la regulación de velocidad. Por otro lado, el ACO se adapta muy bien en ambientes dinámicos, pero el tiempo de convergencia puede llegar a ser muy largo (aunque la convergencia sí está garantizada).

CAPÍTULO 3

Justificación

En la segunda fase del proyecto Robotat de la Universidad del Valle de Guatemala se desarrolló un algoritmo basado en el algoritmo *Particle Swarm Optimization* para planificar una trayectoria óptima para que los robots llegaran a una meta determinada. También se diseñaron distintos controladores para asegurar que los robots recibieran velocidades coherentes con respecto a sus limitantes físicas. Ya que este algoritmo y controlador fueron exitosos en simulación, se desearía comparar con otros algoritmos para que, de este modo, el mejor algoritmo sea finalmente implementado en los robots físicos.

Como propuesta de algoritmo se tiene el *Ant Colony Optimization* (ACO), pues es otro de los más utilizados de la rama de inteligencia computacional de enjambre. Se debe encontrar los mejores parámetros para que el algoritmo converja en un tiempo similar al logrado con el PSO de la fase II del proyecto Robotat. Asimismo se busca implementar los mismos controladores que en el proyecto mencionado anteriormente para que la comparación sea equitativa. Con este trabajo se logra encontrar alternativas en planificación de movimiento para robots móviles.

CAPÍTULO 4

Objetivos

4.1. Objetivo general

Implementar y verificar algoritmos de inteligencia de enjambre y computación evolutiva como alternativa al método de *Particle Swarm Optimization* para los Bitbots de UVG.

4.2. Objetivos específicos

- Implementar el algoritmo *Ant Colony Optimization* (ACO) y encontrar el valor de los parámetros de las ecuaciones del ACO que permitan a los elementos de la colonia encontrar una meta específica, en un tiempo finito.
- Validar los parámetros encontrados por medio de simulaciones computarizadas que permitan la visualización del comportamiento de la colonia.
- Adaptar los modelos del movimiento y de la cinemática de los Bitbots, desarrollados en la fase anterior del proyecto, al algoritmo ACO.
- Validar el algoritmo ACO adaptado implementándolo en robots físicos simulados en el software Webots, y comparar su desempeño con el del *Modified Particle Swarm Optimization* (MPSO).

CAPÍTULO 5

Alcance

El alcance de este trabajo abarcó la implementación simulada del algoritmo *Ant Colony* (ACO) para planificar trayectorias en robots diferenciales. Para realizar esto, primero se programó una versión simple del algoritmo llamada *Simple Ant Colony* (SACO) y luego se implementó la versión planteada por Marco Dorigo, *Ant System* (AS). Asimismo, se implementaron varios controladores en la velocidad de las ruedas de los robots para considerar sus restricciones físicas. Los robots utilizados en esta fase fueron los robots e-puck, por lo que en futuras implementaciones se debería de considerar robots con dimensiones similares. Esto último se realizó con el *software* Webots con el lenguaje de programación Matlab. Para futuras implementaciones se podrá implementar este algoritmo en los Bitbots físicos de UVG. Además, también podría realizarse pruebas con obstáculos o incluso con otros lenguajes de programación para comparar su desempeño.

CAPÍTULO 6

Marco teórico

6.1. Inteligencia Computacional de Enjambre

El campo de la Inteligencia de enjambre:, o *Swarm*, parte de la idea de que individuos interactúan entre ellos para resolver un objetivo global por medio del intercambio de su información local. De esta manera, la información se propaga más rápido y por tanto, el problema se resuelve de una manera más eficiente. El término *Swarm Intelligence* (SI) se refiere a esta estrategia de resolución de problemas de forma colectiva, mientras que *Computational Swarm Intelligence* (CSI) se refiere a algoritmos que modelan este comportamiento [7].

Al tener diversos estudios sobre distintos tipos de animales es posible crear algoritmos que modelen su comportamiento. Algunos sistemas biológicos que han inspirado este tipo de algoritmos son: hormigas, termitas, abejas, arañas, escuelas de peces y bandadas de pájaros. Dos de los algoritmos que se estudiarán más adelante están basados en pájaros y hormigas, modelando su comportamiento individual y colectivo.

6.2. Particle Swarm Optimization (PSO)

Este algoritmo hace referencia al comportamiento de las bandadas de pájaros y cardúmenes de peces al realizar búsquedas óptimas [8]. Las partículas encuentran la solución a partir de su mejor posición y la mejor posición de todas las partículas [9]. Las ecuaciones que modelan esta situación son las siguientes [8], [9]:

$$v_{id}^{t+1} = v_{id}^{t+1} + c_1 r_1 (p_{id}^t - x_{id}^t) + c_2 r_2 (p_{gd}^t - x_{id}^t) \quad (1)$$

$$x_{id}^{t+1} = x_{id}^t + v_{id}^{t+1} \quad (2)$$

Donde v es la velocidad, x la posición, c_1 y c_2 son constantes de aceleración, p_{id} es la mejor posición personal, p_{gd} es la mejor posición que tuvo todo el enjambre en la iteración t y r_1 y r_2 son números aleatorios entre 0 y 1. La fase de exploración ocurre cuando se tiene una diferencia grande entre la posición de la partícula y el mejor de la partícula (**pbest**) o mejor global (**gbest**). La ventaja de este método contra los que necesitan usar el gradiente es que no se requiere que el problema sea diferenciable. Además, pueden optimizarse problemas con ruido o problemas dinámicos o cambiantes. La función que se desea optimizar en este caso se denomina función de costo [8].

6.2.1. Funcionamiento del algoritmo PSO

Se inicia el programa creando las partículas, dándoles una posición y velocidad inicial. Luego se introducen esos valores a la función de costo y se actualizan los valores de **pbest** y **gbest**. Este proceso se repite hasta que se cumpla un número de iteraciones o se logre la convergencia del algoritmo. La convergencia se alcanza cuando todas las partículas son atraídas a la partícula con la mejor solución (**gbest**). Un factor que se debe tomar en cuenta es el de restringir los valores que pueden tomar las funciones para que el algoritmo no diverja [9].

6.2.2. Mejora del algoritmo

El algoritmo puede mejorarse a través del incremento del número de partículas, introducción del peso de inercia w o la introducción de un factor de restricción K . El factor de inercia controla las fases de explotación y desarrollo del algoritmo. Se sugiere utilizar un valor mayor de inercia y decrementarlo hasta un valor menor para tener una mayor exploración al principio, pero al final mayor explotación [8].

$$v_{id}^{t+1} = wv_{id}^{t+1} + c_1r_1(p_{id}^t - x_{id}^t) + c_2r_2(p_{gd}^t - x_{id}^t) \quad (3)$$

$$v_{id}^{t+1} = K(wv_{id}^{t+1} + c_1r_1(p_{id}^t - x_{id}^t) + c_2r_2(p_{gd}^t - x_{id}^t)) \quad (4)$$

El otro caso es utilizar el factor de restricción K para mejorar la probabilidad de convergencia y disminuir la probabilidad de que las partículas se salgan del espacio de búsqueda [8].

6.3. Ant Colony Optimization (ACO)

A partir de las observaciones de los entomólogos, se determinó que las hormigas tienen la habilidad de encontrar el camino más corto entre una fuente de alimento y su hormiguero. Marco Dorigo desarrolló un algoritmo con base en el comportamiento de estos insectos, a partir del cual se han derivado muchas otras variantes. Sin embargo, la idea principal de utilizar a las hormigas como base del algoritmo puede verse como una instancia de la Metaheurística: de Ant Colony Optimization (ACO-MH). Algunas de las instancias más conocidas son: Ant System (AS), Ant Colony System (ACS), Max-Min Ant Aystem (MMAS), Ant-Q, Fast Ant System, Antabu, AS-rank y ANTS [7].

Cuando una hormiga encuentra una fuente de alimento, al regresar al nido esta deja un rastro de feromonas para indicarle a sus compañeras que si siguen ese camino, encontrarán comida. Mientras más hormigas escojan ese camino, más fermona habrá y asimismo, mayor probabilidad de que las demás hormigas elijan esa ruta. Esta forma de comunicación indirecta que tienen las hormigas es denominada *stigmergy*. “La feromona artificial imita las características de la feromona real, indicando la popularidad de la solución del problema de optimización que se está resolviendo. De hecho, la feromona artificial funciona como una memoria a largo plazo del proceso completo de la búsqueda” [7].

Al principio, las hormigas se comportan de una manera aleatoria. Deneubourg realizó un experimento en el que se colocó comida a cierta distancia del hormiguero, pero solo podían acceder a ella en dos caminos diferentes. En este experimento, los caminos eran del mismo tamaño, pero con el comportamiento aleatorio de las hormigas uno de ellos fue seleccionado. Este experimento es conocido como el "puente binario". Asimismo, en un experimento similar pero con uno de los caminos más corto que el otro, las hormigas al principio eligieron los dos de forma equitativa, pero conforme el tiempo pasó eligieron el más corto. Este efecto es denominado “largo diferencial” [7].

6.3.1. Simple Ant Colony Optimization (SACO)

Este algoritmo es el que se utilizó en la implementación del experimento del puente binario. Este considera el problema general de búsqueda del camino más corto entre un conjunto de nodos en un grafo $G = (V, E)$ donde la matriz V representa a todos los vértices o nodos del grafo y la matriz E a todas las aristas o *edges* del grafo [7].

Asimismo, el largo L^k del camino construido por la hormiga k se calcula como el número de saltos en el camino desde el nodo que representa al nido hasta el que representa el destino con la comida. En la Figura 1 puede verse un camino marcado con flechas continuas con un largo de 2. En cada arista (i,j) del grafo se tiene cierta concentración de feromona asociada τ_{ij} . En este algoritmo, la concentración inicial de feromona se asigna de forma aleatoria¹ (aunque en la vida real sea de 0). Cada hormiga decide inicialmente de forma aleatoria² a qué arista dirigirse. K hormigas se colocan en el nodo fuente (*source*). Para cada iteración, cada hormiga construye una solución al nodo destino. En cada nodo i, cada hormiga k determina a qué nodo j debe de dirigirse basado en la probabilidad p:

¹Sin embargo en [10] recomiendan que todas las aristas comiencen con el mismo valor, como 1 por ejemplo.

²En [10] y [7] recomiendan usar el algoritmo *Roulette Wheel*.

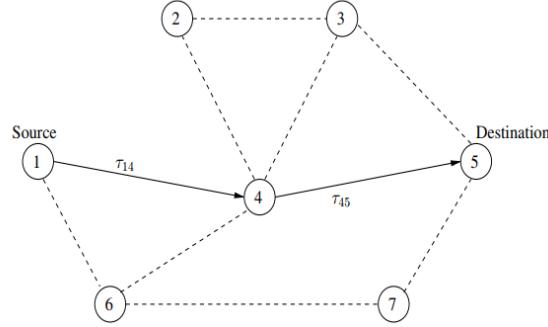


Figura 1: Ejemplo de un grafo [7].

$$p_{(i,j)}^k(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha}{\sum_{k \in J_k} (\tau_{ij}(t))^\alpha} & \text{si } j \in N_i^k \\ 0 & \text{si } j \notin N_i^k \end{cases} \quad (5)$$

Donde N representa al conjunto de nodos viables conectados al nodo i . Si en cualquier nodo el conjunto N es el conjunto vacío, entonces el nodo anterior se incluye. En la ecuación (5), α es una constante positiva que se utiliza para modular la influencia de la concentración de feromonas. Cuando este parámetro es muy grande la convergencia puede ser extremadamente rápida y resultar en un camino no óptimo. Cuando las hormigas llegan al nodo destino regresan a su nodo fuente por el mismo camino por el cual llegaron y depositan feromonas en cada arista [7]. En este proceso también se modela una forma de evaporación de feromonas para aumentar la probabilidad de exploración del terreno.

La ecuación que gobierna la evaporación de la feromona en los trayectos es la siguiente:

$$\tau_{ij}(t+1) = (1 - \rho) \cdot \tau_{ij}(t) + \sum_{k=1}^m (\Delta \tau_{ij}^k(t)) \quad (6)$$

Donde m es el número de hormigas, ρ es el factor de evaporación de feromona (entre 0 y 1). Además, se sabe que:

$$\Delta \tau_{ij}^k(t) = \frac{Q}{L_k}(t) \quad (7)$$

Donde Q es una constante y L es el costo del trayecto, como el largo del mismo. El valor representa el cambio de feromona entre el nodo i y j que la hormiga visitó en la iteración t [8].

6.3.2. Ant System

Este fue el primer algoritmo desarrollado que emula el comportamiento de las hormigas, que consiste en hacer ciertas mejoras al algoritmo presentado anteriormente. El algoritmo anterior tenía un objetivo más instructivo, por lo que era más simple. Algunas de las mejoras son la incorporación de información heurística en la ecuación de probabilidad y agregando capacidad de memoria con una lista tabú. La nueva ecuación de probabilidad es la siguiente:

$$p_{(i,j)}^k(t) = \begin{cases} \frac{(\tau_{ij}(t))^\alpha \cdot (\eta_{ij}(t))^\beta}{\sum_{k \in J_k} (\tau_{ij}(t))^\alpha \cdot (\eta_{ij}(t))^\beta} & \text{si } j \in N_i^k \\ 0 & \text{si } j \notin N_i^k \end{cases} \quad (8)$$

Donde el nuevo parámetro η representa el inverso del costo de la arista. En ambos algoritmos presentados, la condición de paro puede ser un límite de iteraciones o generaciones, o bien, cuando un porcentaje determinado de hormigas haya elegido la misma solución [7].

6.4. Computación evolutiva

La evolución (en biología) puede ser definida como un proceso de optimización donde los individuos cambian de forma dinámica para sobrevivir en ambientes competitivos. Aunque Darwin se considera el padre de la teoría de la evolución, fue Jean-Baptiste Lamark quien en realidad lo teorizó. Esta teoría consistía en que los padres heredan ciertas características a sus hijos y luego estos continúan adaptándose. Además, la teoría de selección natural de Darwin resume el hecho de que los individuos con mejores características tienen mayor probabilidad de supervivencia y de reproducirse. Por tanto, las características menos útiles y los individuos más débiles suelen desaparecer. La computación evolutiva (EC por sus siglas en inglés) se encarga de resolver problemas utilizando modelos de procesos evolutivos como la selección natural y supervivencia del más apto [7]. Dependiendo de la implementación, estos algoritmos pueden dividirse en los siguientes paradigmas:

- Algoritmos genéticos (GA)
- Programación genética (GP)
- Programación evolutiva (EP)
- Estrategias evolutivas (ES)
- Evolución diferencial (DE)
- Evolución cultural (CE)
- Co-evolution (CoE)

6.4.1. Algoritmos Genéticos (GA)

Los algoritmos genéticos son un tipo de algoritmo que utiliza de forma inteligente los conocimientos sobre evolución en biología para resolver problemas en distintos ámbitos de forma computacional. En esta evolución, las características de los individuos se expresan en genotipos y se tiene un operador de selección y otro de cruce o recombinación. El operador de selección se encarga de modelar la supervivencia de los individuos más aptos, mientras que el operador de recombinación modela la reproducción entre los individuos. Asimismo, la etapa de la recombinación o cruce se encarga de explotar las soluciones, mientras que la etapa de mutación se encarga de proveerle cierto grado de diversidad a la población para así evitar convergencia prematura a soluciones sub-óptimas [7].

Inicialmente se tiene una población de soluciones conformada por cromosomas (cada cromosoma representa una solución), los cuales están conformados por genes o características. Los valores que pueden tomar estos genes son los alelos y dependen de la codificación que se le da al problema. La codificación es importante porque necesitamos traducir las variables físicas con las que trabaja el problema para que la computadora pueda interpretarlas y así aplicarle los operadores genéticos [7].

6.4.2. Métodos de codificación

John Holland es considerado el padre de los algoritmos genéticos, luego de realizar trabajos con los mismos a partir de los trabajos previamente realizados por Bremermann y Reed. En este primer algoritmo genético canónico se utilizó una codificación binaria con formato de *string*, selección proporcional para elegir qué padres serán recombinados, cruce en un punto para producir “cromosomas hijos” y mutación uniforme [7].

Codificación binaria

Algunas de sus ventajas son que casi todos los tipos de hipótesis pueden ser planteadas utilizando números binarios y que las computadoras tienen como lenguaje primario al código binario. Por lo tanto, las computadoras realizan todas sus operaciones en binario por ser el único lenguaje que conocen y si le damos directamente números en su idioma el tiempo de procesamiento será más rápido. Para realizar el diseño de la codificación de un espacio de hipótesis con código binario se representan ciertos atributos utilizando subconjuntos del mismo *string* de números binarios. Estos subconjuntos (o genes en nuestro caso) pueden ser alterados sin necesidad de alterar a sus subconjuntos vecinos [11].

Codificación entera

Además de la codificación binaria también existen codificaciones en base 10 como lo son la codificación de enteros y la de los reales. La codificación de enteros es muy conveniente en problemas que requieran como respuesta un orden específico de objetos ordenados. Un ejemplo donde es fácil visualizar la utilidad de este tipo de codificación es en el problema del vendedor viajero o TSP por sus siglas en inglés. Este problema busca encontrar el orden en el que el viajero debe de visitar ciertas ciudades dadas, tomando en cuenta que se debe de optimizar la ruta según la distancia de ciudad en ciudad. La codificación binaria podría utilizarse aquí, pero es posible que se requiera algún algoritmo corrector extra luego de hacer las operaciones de cruce o mutación pues algunos resultados pueden llevar a nodos (o ciudades) no existentes en la precisión dada por la longitud del número binario [12].

Codificación real

Finalmente tenemos la codificación de valores reales, que puede ser conveniente en problemas donde se esté trabajando en tiempo continuo, por lo que lo más intuitivo es utilizar números de punto flotante. Estos números claro que pueden ser expresados en binario, pero puede llegar a ser muy complicado su manejo. Por lo tanto, en estos casos donde la complejidad de la representación de los objetos es alta con números binarios se recomienda utilizar valores reales [13].

Como se mostró anteriormente, la codificación binaria es lo suficientemente robusta para representar todos los casos. Cabe recordar que nuestra computadora utiliza binario como lenguaje nativo y por lo tanto es posible codificar todo en este lenguaje. Sin embargo, en algunos casos es más intuitivo y fácil de calcular y manejar el problema utilizando números en base 10. Por lo tanto, se recomienda al lector apegarse a lo que le sea más fácil de interpretar, claro, sin comprometer la velocidad y complejidad del algoritmo.

6.4.3. Función de costo

Esta es la función que se desea minimizar (o maximizar) utilizando los algoritmos genéticos. Existen algunas funciones famosas utilizadas para probar algoritmos de optimización que tienen ciertas características como: muchos mínimos locales, forma de plato hondo, forma de valle, con gotas, etc. A continuación se presenta algunos ejemplos de este tipo de funciones [14].

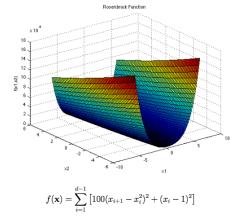


Figura 2: Gráfico y función de Rosenbrock [14].

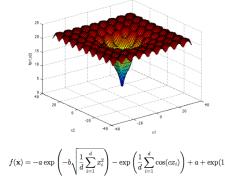


Figura 3: Gráfico y función de Ackley [14].

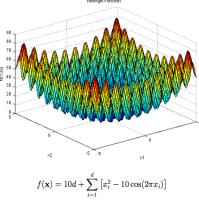


Figura 4: Gráfico y función de Rastrigin [14].

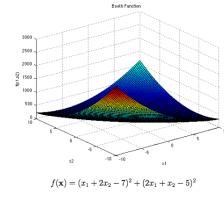


Figura 5: Gráfico y función de Booth [14].

6.4.4. Métodos de selección proporcional

El método de selección proporcional fue propuesto por John Holland y sesga la selección según el costo de los individuos; es decir, los mejores individuos tienen mayor probabilidad de ser seleccionados [15]. Por tanto, se crea una distribución proporcional al costo como se muestra a continuación:

$$\phi_s(x_i(t)) = \frac{f_\gamma(x_i(t))}{\sum_{l=1}^{n_s} f_\gamma(x_l(t))} \quad (9)$$

Donde n_s representa al número de individuos en la población, $\phi_s(x_i)$ es la probabilidad de que un individuo x_i sea seleccionado y $f_\gamma(x_i(t))$ es igual al valor máximo de la función de costo menos el valor de la función de costo para el individuo x_i .

Determinístico

Uno de los métodos más populares y el método utilizado en este trabajo es el de la selección mediante una ruleta. En este caso cada sección de la ruleta es proporcional a la probabilidad normalizada de un individuo.

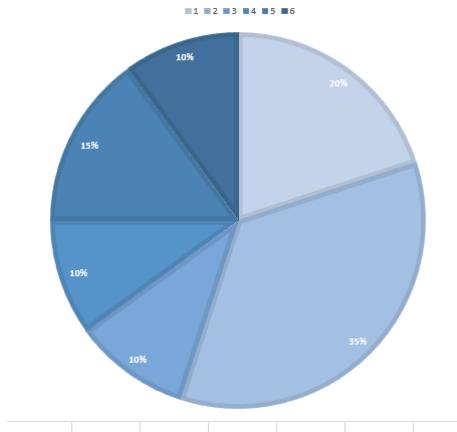


Figura 6: Selección de la ruleta en forma gráfica para 6 individuos.

A parte de este método de selección también existe el método Estocástico: y los métodos de selección tipo torneo. Sin embargo estos no son relevantes para esta investigación por lo que se dejará al lector profundizar al respecto en [7] si así lo desea. Asimismo puede investigarse sobre métodos de selección más elaborados como el presentado en [16].

6.4.5. Operadores de crusa en GA

Existen tres clases principales de operadores de crusa o recombinación: asexual, sexual y multi-recombinación. En el caso se la asexual, los descendientes son generados a partir de solamente un parente. En la segunda clase, los descendientes son generados a partir de dos padres seleccionados. Cabe mencionar que en este caso la cantidad de descendientes puede ser de 1 o 2. Finalmente, tenemos el caso de multi recombinación donde la única diferencia con el anterior es que puede producir más hijos (más de 2) si así se desea. Aparte de esta clasificación de los operadores, también se tiene la clasificación por tipo de dato (binario, entero o decimal) que se utilizará en la codificación de los cromosomas [7].

En la mayor parte de los operadores binarios se utiliza la reproducción sexual con dos padres, donde debe especificarse qué bits serán intercambiados para crear a los dos hijos. Como se mencionó anteriormente, para codificar los cromosomas se utiliza el alfabeto finito binario, constituido por los valores de 0 y 1 únicamente. Este fue el método usado por Holland y el más simple de utilizar con *strings* de bits de un largo fijo. Algunas desventajas son el hecho de que el largo tiene que ser fijo y que por lo tanto, la precisión de la respuesta también está sujeta a dicho parámetro [11].

En la crusa de un punto se elige un punto en los dos individuos seleccionados, cruzando la segunda parte de sus dígitos. Por ejemplo, si elijo intercambiar los últimos dos dígitos, el resultado A' y B' sería como sigue:

$$\begin{array}{ll} A = 111|11 & B = 000|00 \\ A' = 111|00 & B' = 000|11 \end{array}$$

En la crusa de dos puntos se elige dos puntos y se intercambian los segmentos del medio:

$$\begin{array}{ll} A = 101|010|1010 & B = 010|101|0101 \\ A' = 101|101|1010 & B' = 010|010|0101 \end{array}$$

Además de estos métodos presentados en [15], también existen otros métodos más complejos como los presentados en [17], donde explican 11 métodos diferentes de crusa para representaciones con números enteros. En este último artículo se centran en investigar el desempeño de varios métodos de crusa para la solución del problema del agente viajero.

6.4.6. Métodos de mutación

Un método de mutación puede ser el intercambio aleatorio de uno o más bits en GA binarios (*Bit Flip Mutation*). También se puede hacer el llamado *Random Resetting*, donde se cambia un valor en una cadena de números enteros por uno de los valores permitidos. Asimismo, para las cadenas de enteros existe la mutación de cambio de valores o *Swap Mutation*. A continuación se da un ejemplo de *Swap Mutation* [18].

$$\begin{aligned}A &= 1234567890 \\A' &= 1634527890\end{aligned}$$

La mutación debe de aplicarse con una probabilidad menor al 10% y teniendo cuidado de no eliminar soluciones potenciales. La utilidad de la mutación en los algoritmos genéticos está en la exploración de nuevas soluciones, mientras que en la cruxa ocurre explotación de las soluciones encontradas. La clave de un buen algoritmo genético está en encontrar la combinación adecuada entre explotación y exploración de las soluciones [15]. Al igual que con la cruxa, en [17], Larrañaga *et. al* compara también 6 distintos métodos de mutación para el problema del agente viajero.

6.5. Grafos

Como pudo observarse anteriormente, el algoritmo *Ant System* es un algoritmo de búsqueda basado en grafos y no funciones como el PSO. Por lo tanto, es necesario definir qué es un grafo y de qué maneras puede representarse. Un grafo G es un par de conjuntos (V, E) donde V representa al conjunto de vértices del grafo y E al conjunto de pares no ordenados de elementos de V [19].

$$\begin{aligned}V &= \{V_1, V_2, \dots, V_n\} \\E &= \{(V_i, V_j), (V_i, V_j), \dots\}\end{aligned}$$

Existen los grafos dirigidos (o digrafos), donde el enlace del nodo i al j no es el mismo que del nodo j al i. Estos grafos sí cuentan con dirección. Asimismo, existen los grafos (y digrafos) ponderados. A estos se les asocia un número de peso o costo a cada arista. Este número podría ser distancia, capacidad o valor temporal dependiendo de la aplicación [20].

Los grafos pueden representarse a partir de dibujos como el de la Figura 1, donde los grafos se muestran como en la Figura y los digrafos con flechas para indicar la dirección. Además, también pueden representarse a partir de matrices como la matriz de adyacencia. En esta matriz cuadrada de tamaño número de vértices se muestra qué vértices están y no están conectados con un 1 y un 0 respectivamente [21]. A continuación se muestra un ejemplo de matriz de adyacencia de la Figura 1.

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 \end{bmatrix}$$

Como puede observarse, cada fila y columna representa a un vértice, empezando desde el vértice 1 en la primera fila y primera columna (esquina superior izquierda).

6.5.1. Grafos en Matlab

Para representar grafos en Matlab se utiliza la clase `graph`. Los objetos de la clase `graph` tienen métodos para acceder y modificar nodos y aristas, para mostrar el grafo en representación de matrices, mostrar información del grafo, visualización e incluso para encontrar el camino más corto entre nodos. Algunos de los atributos (o propiedades como les llama Matlab) son las aristas y los nodos, aunque es posible agregarle más atributos a la clase. Para mas información y documentación acerca de la clase `graph` de Matlab, visitar [22].

6.6. Programación orientada a objetos

Como se explicó anteriormente, el software Matlab representa los grafos como objetos, por lo que es necesario saber lo básico de cómo funcionan estos para implementarlos. Un objeto es una colección de datos con ciertos comportamientos asociados. Un objeto puede ser de una clase en específico, que se define como una plantilla para crear objetos. El objeto en sí de una clase se llama instancia, y es el que se utiliza activamente en la programación. Para diferenciar objetos se utilizan atributos o características del mismo. Por ejemplo, un objeto de la clase fruta puede ser manzana o pera y se diferencian por sus atributos color y forma (por ejemplo) [23].

Como se explicó en el párrafo anterior, un objeto tiene comportamientos asociados. Por tanto, existen ciertas funciones o acciones relacionadas con el objeto. Estas funciones se denominan métodos y en esencia son funciones que solo pueden ser utilizadas por instancias de la clase donde están definidas [23]. Generalmente para referirse a un atributo se utiliza la notación `instancia.atributo` y para usar un método se utiliza `instancia.función()`. Para conocer y aprender sobre la notación específica de programación orientada a objetos en Matlab puede verse [24].

6.7. Computación paralela

Para realizar pruebas debe de correrse el programa varias veces, por lo que se consumiría demasiado tiempo dependiendo de la cantidad y forma de realizar las pruebas. Por lo tanto, se propone utilizar programación paralela para aprovechar los distintos núcleos de las computadoras para acelerar el proceso [25].

Los *softwares* tradicionales se escriben para computación serial y no paralela, por lo que el problema se divide en pedazos pequeños y se van pasando uno por uno al procesador como puede verse en la Figura 7 [25].

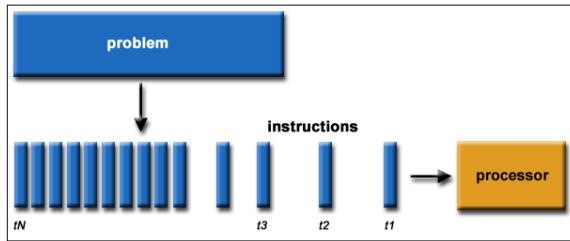


Figura 7: Computación serial [25]

Sin embargo, en la computación paralela la tarea se divide en el número de núcleos que tenga la máquina para realizar tareas en simultáneo y así ahorrar tiempo de ejecución (ver Figura 8). Además, con paralelización es posible resolver problemas más complejos y al minimizar el tiempo (dependiendo de la aplicación) incluso podría ahorrarse algún costo [25].

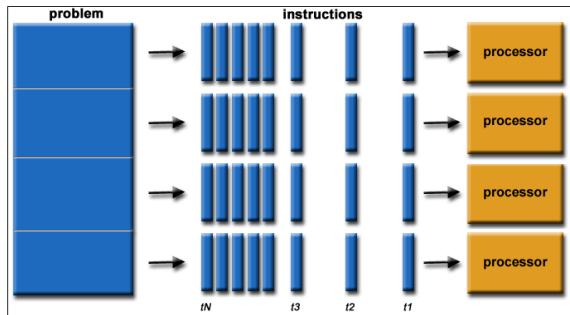


Figura 8: Computación paralela [25]

6.7.1. Programación paralela en Matlab

Matlab tiene un *Toolbox*: de computación paralela con funciones como el `parfor`, que reemplaza el `ciclo for` normal por un `ciclo for` paralelizado. Para correr un programa paralelizado de Matlab primero es necesario habilitar el `Parallel Pool` de Matlab y establecer la cantidad de procesadores (Matlab les llama *workers*) a utilizar [25].

Para reescribir un **ciclo for** normal a un **parfor** se requiere que cada una de las subtareas sea independiente, por lo que no pueden utilizarse entre sí. Además, también es necesario que el orden de ejecución de las tareas no importe (también el orden debe ser independiente). Algunas restricciones para el **parfor** son que no se puede introducir variables utilizando las funciones **load**, **eval** y **global**, no pueden contener **break** o **return** y tampoco puede contener otro **parfor** dentro [25]. Para más información sobre este tema se recomienda ver [26].

Además de la programación paralela en Matlab, se recomienda utilizarla en conjunto con buenas prácticas de programación [27] y técnicas específicas para mejorar el desempeño [28], que incluyen estrategias como vectorización de código [29]. Si llegara a suceder que el código está trabajando de forma muy lenta es posible utilizar la técnica de búsqueda de cuellos de botella dada en [30] específicamente para Matlab utilizando la función *profiler*. Si se desea implementar paralelización en Matlab se recomienda leer todas las referencias bibliográficas de esta sección.

6.8. Planificación de movimiento

Se define como el problema de encontrar el movimiento desde un destino hasta un final esquivando obstáculos (si hubiese) y cumpliendo restricciones de juntas y/o torque para un robot [31].

6.8.1. Espacio de configuración

El espacio de configuración o espacio C corresponde a una configuración **q** del robot. Es decir, cada punto del espacio de configuración corresponde a una configuración diferente del robot. La configuración de un robot serial con n juntas está representada por el vector **q** con la posición de las n juntas, por lo que será un vector de dimensión n. El espacio libre en el espacio de configuración está dado por todas las configuraciones del robot donde este no colisiona con ningún obstáculo [31]. Este concepto de espacio de configuración se utilizará posteriormente para planificar la trayectoria.

6.8.2. Planificación de trayectorias

El problema de planificar una trayectoria es un subproblema de la planificación de movimiento. En este caso se requiere encontrar un camino sin colisiones desde un inicio hasta un destino. Estos problemas pueden ser resueltos *online* u *offline*, dependiendo de si se necesita el resultado inmediatamente o si el ambiente es estático respectivamente [31].

6.9. Métodos de planificación de movimiento

6.9.1. Métodos completos

Los métodos completos se enfocan en representar de manera exacta la geometría del espacio libre de configuración [31].

Grafos de visibilidad

El Grafo de visibilidad: es un mapa que utiliza los vértices de los obstáculos en el espacio de configuración como nodos. Estos se conectan solamente si pueden *verse*. El peso de las aristas del grafo es la distancia euclíadiana entre los nodos. Este método puede utilizarse y resultar adecuado para obstáculos poligonales [31].

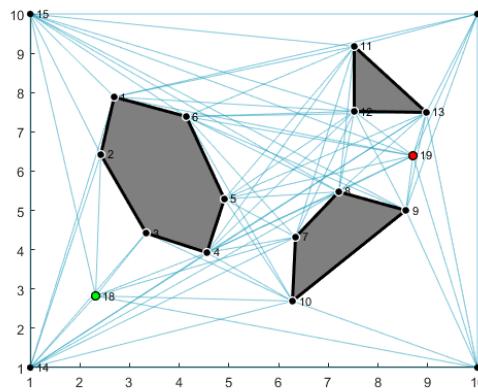


Figura 9: Ejemplo de grafo de visibilidad en Matlab.

6.9.2. Métodos de cuadrícula

Este método discretiza el espacio libre de configuración, creando una especie de cuadrícula para que el algoritmo busque una solución. Estos métodos pueden ser efectivos, pero la memoria requerida y el tiempo de búsqueda crecen de manera exponencial con el número de dimensiones del espacio. Es decir, una resolución de 100 en un espacio de dimensión 2 lleva a una cantidad de 100^2 nodos [31].

6.9.3. Métodos de muestreo

Estos métodos utilizan una función aleatoria o determinística para elegir muestras del espacio de configuración. Luego, evalúan si la muestra tomada está en el espacio libre y determina la muestra más cercana en el espacio libre para conectarlas. El resultado final de este algoritmo es un grafo o árbol que representa los movimientos legales del robot. Estos métodos sacrifican la resolución del mapa para minimizar la complejidad computacional [31].

Rapidly Exploring Random Trees (RRT:)

Este método representa el espacio de configuración en forma de un árbol (grafo con nodos hijos), utilizando el método de muestreo aleatorio presentado anteriormente para encontrar el siguiente nodo disponible. Existe una variante de este método, donde se arma dos árboles: uno desde el nodo inicio y otro desde el nodo destino y cada cierto tiempo el algoritmo intenta unir ambos árboles.

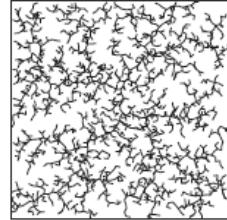


Figura 10: Ejemplo de un RRT [31]

Probability Road Maps (PRM:)

Un PRM también representa el el espacio libre de configuración con puntos aleatorios conectados entre sí. Una ventaja de este algoritmo es que se puede crear de forma rápida, por lo que puede ser eficiente. La clave para el PRM es elegir cómo hacer el muestreo, pues generalmente se hace de forma aleatoria a partir de una distribución uniforme del espacio de configuración, pero se ha mostrado que los mejores resultados se obtienen al muestrear de forma más densa cerca de los obstáculos.

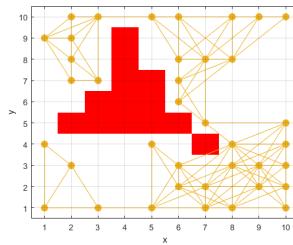


Figura 11: Ejemplo de un PRM en Matlab.

6.9.4. Artificial Potential Fields (APF)

Los campos de potencial artificiales crean fuerzas atractivas en el objetivo, atrayendo al robot en su dirección. Asimismo, crean fuerzas repulsivas en los obstáculos para alejar al robot de los mismos y así evitar colisiones. Este método es relativamente fácil de implementar y evaluar, pero puede quedarse atascado en mínimos locales de la función de potencial [31]. Este fue el método utilizado en la etapa II del proyecto Robotat UVG [2] y puede observarse un ejemplo con un simple obstáculo en la Figura 12.

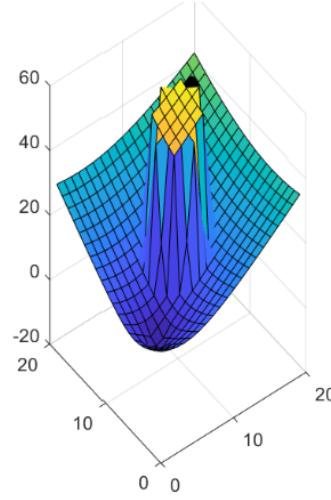


Figura 12: Ejemplo de un APF en Matlab.

6.10. Modelado de robots móviles

Los robots seriales pueden representarse utilizando cinemática directa. Con esta se *mapea* el vector de configuración de robot (dado por los ángulos de las juntas) a la posición (x, y, θ) . Sin embargo, en el caso de los robots móviles³, en lugar de juntas se tienen ruedas. Estas ruedas tendrán una velocidad angular ϕ , por lo que si se desea encontrar la distancia recorrida s (por rueda), sabiendo que r es el radio de las llantas, se tendría que:

$$s = r\phi \quad (10)$$

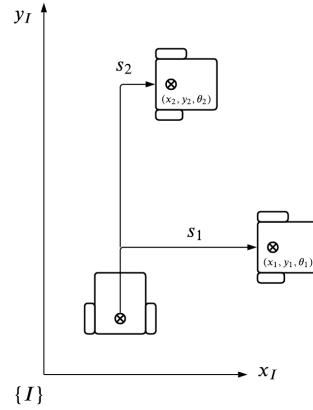


Figura 13: Distancia recorrida de una rueda para el robot móvil.

³Se recomienda al lector ver las lecciones 11 y 12 de robótica 1 en [32] antes de leer esta y la sección siguiente.

En teoría, es posible encontrar la distancia recorrida, pero esto no funciona porque si se realiza una trayectoria similar, pero en otra dirección como en la Figura 13, la distancia resultante sería la misma ($s_1 = s_2$). Sin embargo, el punto (x, y) no es el mismo, entonces este *mapeo* similar al de cinemática directa no es posible. Una configuración lleva a varias poses, indicando que se tienen restricciones no integrables (si se integra la velocidad no se obtiene posición). Estas restricciones también se conocen como restricciones no holonómicas [32]. Debido a que estas restricciones aplican a las velocidades, se aplicará la cinemática diferencial.

6.10.1. Modelo uniciclo

Este es el modelo más simple de un robot móvil. Consiste en una sola rueda y vector de configuración $q = (\phi, x, y, \theta)$. De este se obtienen las velocidades lineales de las ruedas como sigue:

$$v_R = \phi_R r \quad v_L = \phi_L r \quad w = w$$

Donde w representa a la velocidad angular del robot, v representa la velocidad lineal del robot y v_R y v_L son las velocidades lineales de las ruedas derecha e izquierda respectivamente. La velocidad angular de los motores (ϕ) es conocida, por lo que de este modo es posible calcular las velocidades lineales. Estas variables mantendrán el mismo significado en el resto del trabajo.

6.10.2. Modelo diferencial

El modelo diferencial del robot móvil ya toma en cuenta 2 ruedas, por lo que es el que se utilizará en esta tesis. Sin embargo, se necesita el modelo uniciclo para realizar el desarrollo de este último modelo mencionado [31]. Las ecuaciones siguientes describen el mapeo inverso, es decir, velocidad de las llantas en función de la velocidad lineal y la angular. Esto se debe a que en *software* se controla el uniciclo, pero con el *mapeo* del modelo diferencial [32].

$$v_R = \frac{v + wl}{r} \quad v_L = \frac{v - wl}{r}$$

Donde l es el radio del robot (asumiendo el modelo de un e-puck) y r es el radio de las ruedas del mismo.

6.10.3. Difeomorfismo

Cuando un sistema no es controlable se realiza un ajuste llamado difeomorfismo en el que se logra controlar un robot diferencial solo conociendo la velocidad y orientación en un punto. Las ecuaciones modificadas son las siguientes, donde u_1 y u_2 conforman el vector de control u .

$$v = u_1 \cdot \cos\phi + u_2 \cdot \sin\phi$$

$$w = \frac{-u_1 \cdot \sin\phi}{l} + \frac{u_2 \cdot \cos\phi}{l}$$

6.11. Controladores de posición y velocidad de robots diferenciales

Es necesario controlar la posición y velocidad de los robots para garantizar que llegarán a la pose deseada. Asimismo, se busca que estas trayectorias sean suaves y controladas. Para profundizar más en este tema se recomienda leer el capítulo 6 de [1], así como sus referencias. También se recomienda ver el video 12 de [32].

6.11.1. Control proporcional de velocidades con saturación limitada

En este controlador se utilizan constantes de saturación I y la función $\tanh(x)$ para que las velocidades estén acotadas. Las ecuaciones de control son las siguientes:

$$u_1 = I_x \cdot \tanh\left(\frac{k_x \cdot (x_g - x)}{I_x}\right)$$

$$u_2 = I_y \cdot \tanh\left(\frac{k_y \cdot (y_g - y)}{I_y}\right)$$

Donde x_g se refiere a la coordenada en x de la meta y x la posición actual, I son constantes mayores o iguales a 0. Estas ecuaciones luego se combinan con el difeomorfismo encontrado anteriormente para obtener las ecuaciones que se implementan en código. La variable l representa la distancia entre el centro y el punto de difeomorfismo, pero en esta tesis y en [1] se tomó igual al radio del robot.

$$v = I_x \cdot \tanh\left(\frac{k_x \cdot (x_g - x)}{I_x}\right) \cos\phi + I_y \cdot \tanh\left(\frac{k_y \cdot (y_g - y)}{I_y}\right) \sin\phi \quad (11)$$

$$w = \frac{-I_x \cdot \tanh\left(\frac{k_x \cdot (x_g - x)}{I_x}\right) \sin\phi}{l} + \frac{I_y \cdot \tanh\left(\frac{k_y \cdot (y_g - y)}{I_y}\right) \cos\phi}{l} \quad (12)$$

6.11.2. Control PID de velocidad lineal y angular

Uno de los controladores más populares en el área de sistemas de control es el controlador PID. En este se determinan las constantes K_p, K_I, K_d de forma empírica para compensar el error en estado estable y demás parámetros de rendimiento (t_p, t_s, M_p). La ecuación en el dominio del tiempo del control PID en función del error $e(t)$ es el siguiente:

$$PID(e(t)) = K_p \cdot e(t) + K_I \int_0^t e(\tau) \cdot d\tau + K_d \cdot \frac{de(t)}{dt}$$

Por lo tanto, nuestra implementación puede ser resumida como:

$$\begin{aligned} w &= PID(e_o) \\ v &= PID(e_p) \end{aligned}$$

Donde e_o es el error de orientación y e_p es el error de posición. Estos pueden calcularse como sigue:

$$e_o = atan2 \left(\frac{\sin(\theta_g - \theta)}{\cos(\theta_g - \theta)} \right) \quad (13)$$

$$e_p = \sqrt{(x_g - x)^2 + (y_g - y)^2} \quad (14)$$

Donde θ_g es el ángulo calculado desde el punto actual hasta la meta y θ es el ángulo actual.

Algoritmo 6.1: Pseudocódigo de PID [1]

```

1 e_k_1 = 0;
2 E_k = 0;
3 ...
4 e_k = r - y;
5 eD = e_k - e_k_1;
6 E_k = E_k + e_k;
7 u_k = kP*e_k + K_I*E_k + kD*eD;
8 e_k_1 = e_k;

```

6.11.3. Control PID de acercamiento exponencial

En este se corrige el comportamiento en espiral que presenta el controlador PID ordinario.

$$w = PID(e_o) \quad v = \frac{v_o(1 - e^{-\alpha e_p^2})}{e_p} \quad (15)$$

Donde v_o es la velocidad máxima del robot y α es un parámetro de ajuste arbitrario.

6.11.4. Control de pose

A diferencia de los controladores presentados anteriormente, estos sí toman en cuenta la pose final del robot. Este controlador se explica en la sección 6.4.4 de [1]. Las ecuaciones que definen al controlador son las siguientes:

$$\alpha = -\theta + \theta_g \quad (16)$$

$$\beta = -\theta - \alpha \quad (17)$$

$$v = k_p \rho \quad (18)$$

$$w = k_\alpha \alpha + k_\beta \beta \quad (19)$$

Sabiendo que ρ es el error de posición (como el calculado en las secciones anteriores) y que las constantes k siguen las reglas dadas en [1].

6.11.5. Control de pose de Lyapunov

Este control de pose se basa en el criterio de estabilidad de Lyapunov para que el sistema sea asintóticamente estable. En este caso se tiene el mismo cálculo para α y ρ que en la sección anterior, pero se elimina el uso de un parámetro β .

$$v = k_\rho \cdot \rho \cdot \cos(\alpha) \quad (20)$$

$$w = k_\rho \cdot \sin(\alpha) \cdot \cos(\alpha) + k_\alpha \cdot \alpha \quad (21)$$

6.11.6. Control Closed-Loop Steering

Este controlador presenta características similares a los controladores de pose, pero difiere en el cálculo de las velocidades. El cálculo de α , β y ρ es igual al de las secciones anteriores.

$$v = k_\rho \cdot \rho \cdot \cos(\alpha) \quad (22)$$

$$w = \frac{2v}{5\rho} \cdot \left(k_2 \cdot (\alpha + \text{atan}(-k_1 \cdot \beta)) + \left(1 + \frac{k_1}{1 + (k_1 \cdot \beta)^2} \right) \cdot \sin(\alpha) \right) \quad (23)$$

6.11.7. Control LQR

El regulador cuadrático lineal o LQR por sus siglas en inglés, es un controlador óptimo, que utiliza la menor cantidad de control para alcanzar su objetivo. Debido a que el sistema a controlar resulta no ser controlable, al final de aplicar el control LQR se debe de utilizar el difeomorfismo.

$$u = -K \cdot (x - x_g) + u_g$$

6.11.8. Control LQI

Debido a que el LQR es sensible al ruido (no es robusto a perturbaciones) que existe en la medición de los sensores se le agregó una parte integral para compensar el error de estado estable. Al igual que en el controlador anterior es necesario utilizar el difeomorfismo.

$$u = -K_1 \cdot x + K_2 \cdot \sigma$$

CAPÍTULO 7

Diseño experimental Ant System

En este capítulo se muestra la metodología utilizada para los experimentos en Matlab con AS. En estas pruebas se toma al robot como una masa puntual sin restricciones físicas ni actuadores. Para realizar las simulaciones de esta tesis se utilizó el lenguaje MATLAB versión R2018b debido a la experiencia que se contaba con el mismo.

7.1. Validación del algoritmo Ant System

Este experimento consiste principalmente en realizar una simulación simple del algoritmo *Ant System* para comprobar su funcionamiento. Primero se codificó el algoritmo *Simple Ant Colony* y luego se pasó al *Ant System*, que es un modelo un poco más complejo pues toma en consideración el costo por distancia y no solo la cantidad de feromona depositada. Además, este algoritmo también toma en cuenta otra variable Q para escalar la cantidad de feromona que se deposita. A continuación se presenta el pseudocódigo que se siguió (ver el algoritmo 17.3 de [7]).

Algoritmo 7.1: Pseudocódigo de AS.

```

1 inicializar parametros
2 hasta que un porcentaje de las hormigas siga el mismo camino:
3   por cada hormiga:
4     hasta encontrar el nodo destino:
5       caminar al siguiente nodo segun ec. de probabilidad
6     end
7   end
8   por cada arista:
9     evaporar feromonas
10    actualizar feromonas segun largo de cada arista
11  end
12 retornar el camino encontrado

```

A continuación, las variables en negrilla representan a variables tipo celda, de lo contrario la variable es un *array* o matriz. Se asumen k hormigas¹, n nodos en el grafo y un número máximo de iteraciones tf.

Primero se presentarán las variables con forma de matrices de adyacencia con pesos; mientras más grandes sean los valores, mayor probabilidad tendrán de ser escogidos por la función `ant_decision`.

$$\tau = \begin{bmatrix} \tau_{11} & \dots & \tau_{1n} \\ \vdots & \ddots & \vdots \\ \tau_{n1} & \dots & \tau_{nn} \end{bmatrix} \quad \eta = \begin{bmatrix} \eta_{11} & \dots & \eta_{1n} \\ \vdots & \ddots & \vdots \\ \eta_{n1} & \dots & \eta_{nn} \end{bmatrix}$$

La siguiente variable es una celda que contiene vectores columna:

$$vytau = \begin{bmatrix} \text{vecinos nodo 1 } \tau \text{ de los vecinos nodo 1} \\ \vdots \\ \text{vecinos nodo n } \tau \text{ de los vecinos nodo n} \end{bmatrix}$$

A partir de ahora utilizaremos la variable x para referirnos a vectores fila (x,y) apilados en una columna. En `feas_nodes` se guardan los nodos no visitados o viables a los que cada nodo sí puede dirigirse. En la celda `last_node` se guardan los nodos visitados anteriormente.

$$feas_nodes = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kn} \end{bmatrix} \quad last_node = \begin{bmatrix} x_{11} & \dots & x_{1n} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kn} \end{bmatrix}$$

`Blocked_nodes` guarda la lista de nodos ya visitados por cada nodo y `path_k` guarda la lista de los nodos escogidos por la hormiga k (la actual).

¹en el código el número se expresa como `hormigas`, pero por términos de simplificación se utilizará k en este documento

$$\text{blocked_nodes} = \begin{bmatrix} x_{11} \\ \vdots \\ x_{k1} \end{bmatrix} \quad \text{path_k} = \begin{bmatrix} x_{11} \\ \vdots \\ x_{k1} \end{bmatrix}$$

`L` guarda el costo total de cada camino por hormiga (filas) y por iteración (columnas), mientras que `all_path` guarda cada camino (vectores fila apilados) por hormiga (filas) y por iteración (columnas).

$$L = \begin{bmatrix} L_{11} & \dots & L_{1t} \\ \vdots & \ddots & \vdots \\ L_{k1} & \dots & L_{kt} \end{bmatrix} \quad \text{last_node} = \begin{bmatrix} x_{11} & \dots & x_{1t} \\ \vdots & \ddots & \vdots \\ x_{k1} & \dots & x_{kt} \end{bmatrix}$$

Parámetro	Valor
ρ	0.5
α	1.3
β	1
Q	1
tf	70
ϵ	0.9
τ_0	0.1

Cuadro 1: Parámetros del experimento 1.

Los parámetros son sensibles a la lejanía de los nodos inicial y final, por lo que se recomienda hacer el barrido de parámetros con respecto a la lejanía con la que se desea evaluar al algoritmo. Para este experimento se utilizó el nodo (6,6) como nodo de prueba (y el 1,1 como inicial), utilizando los parámetros mostrados en el Cuadro 1. En este experimento solamente se realizaron 10 ejecuciones debido a que la computadora disponible no puede ejecutar muchos experimentos complejos con varios `ciclos for` anidados. Por lo mismo se propone utilizar una computadora de alto rendimiento como se explica a continuación.

7.2. Validación del algoritmo Ant System paralelizado

Se notó que el algoritmo tardaba demasiado tiempo para hacer corridas de pruebas de barrido de parámetros (más de 8 horas para pocos parámetros en `ciclos for` anidados). Por lo tanto, se propuso realizar la implementación del algoritmo paralelizado, para ejecutar el programa en la computadora de alto rendimiento del departamento de Ingeniería Electrónica UVG aprovechando así todos sus núcleos. Para hacer el algoritmo paralelizado fue necesario modificar el `ciclo for` donde las hormigas recorren el mapa y encuentran un camino. Esta modificación transformó el `ciclo for` mencionado en uno completamente independiente. Las variables que causaban problema eran `feas_nodes` y `blocked_nodes`, pero por la forma de implementarlo, no se podía modificar sin cambiar por completo el código.

Como se explicó en el marco teórico, Matlab cuenta con una clase grafo, por lo que se implementó el algoritmo haciendo uso de programación orientada a objetos. Los atributos de la clase grafo son **Nodes** y **Edges**, por lo que si creamos el grafo G, podemos acceder a esos atributos como sigue:

```

1 % Acceso a los atributos
2 grid_size = 10;
3 G = graph_grid(grid_size);
4 G.Nodes
5 G.Edges

```

Donde la función **graph_grid** es una función auxiliar que se encarga de crear un grafo con los atributos **Nodes** y **Edges**, que son tablas con los siguientes parámetros:

```

1 % grid_graph.m
2 G = graph(table(EndNodes, Weight, Eta), table(Name, X, Y));

```

Como puede observarse, el primer argumento de la función **graph** corresponde al atributo **Edge** y el segundo al atributo **Node**. **EndNodes** representa dos nodos que conforman una arista, a la cual le corresponde un peso τ y un η , similar al código anterior. Con esto podemos eliminar las variables **tau**, **eta** y **vytau**. En el atributo **Nodes** tenemos el nombre del nodo y sus respectivas coordenadas.

Un objeto, además de tener datos, tiene funciones o métodos, por lo que también contamos con los métodos **neighbors** y **plot** para encontrar a sus vecinos y visualizar el grafo respectivamente. En el código anterior, sin programación orientada a objetos, se tenían varias funciones como **neighbors** que ya no son necesarias por tener la clase grafo.

Para sustituir el resto de variables, primero debemos darnos cuenta de que estas están relacionadas con cada hormiga. Por lo tanto, podría pensarse que es posible crear un objeto hormiga y que cada una tenga sus atributos. Sin embargo, debemos recordar que cada objeto debe tener datos y funciones relacionadas. Entonces, en este caso no tenemos funciones por lo que se optó por utilizar una estructura de datos de Matlab llamada **ants** de tamaño igual a la cantidad de hormigas y con los siguientes parámetros:

```

1 hormigas = 50;
2 ants(1:hormigas) = struct('blocked_nodes', [], 'last_node',
    'nodo_init', 'current_node', 'nodo_init', 'path', 'nodo_init', 'L',
    zeros(1, t_max));

```

Esto se debe a que cada hormiga tendrá: sus nodos bloqueados o lista tabú, su último nodo, su nodo actual, el camino encontrado y la longitud del mismo. Aparte de estas variables también se tiene un registro histórico de los caminos y longitudes para comparar si los datos se están calculando de forma correcta.

Ya con estos cambios realizados es posible realizar la paralelización, donde cada iteración es completamente independiente de las demás. La paralelización en Matlab se activa cuando utilizamos la función **parfor** (de *parallel for* en inglés) como se muestra a continuación:

```

1 parfor k = 1:hormigas
2     while( no se haya llegado al nodo destino )
3         % construir el camino
4     end
5 end

```

Para ejecutar el código es necesario prender lo que Matlab llama *Parallel Pool* (piscina paralela) en la esquina inferior izquierda (Figura 14a). De no hacerlo, Matlab lo hace al ejecutar el código, pero se recomienda prenderlo antes para que sea más rápida la primera ejecución del programa. En la Figura 14b se observa una piscina paralela de Matlab lista para ser utilizada.

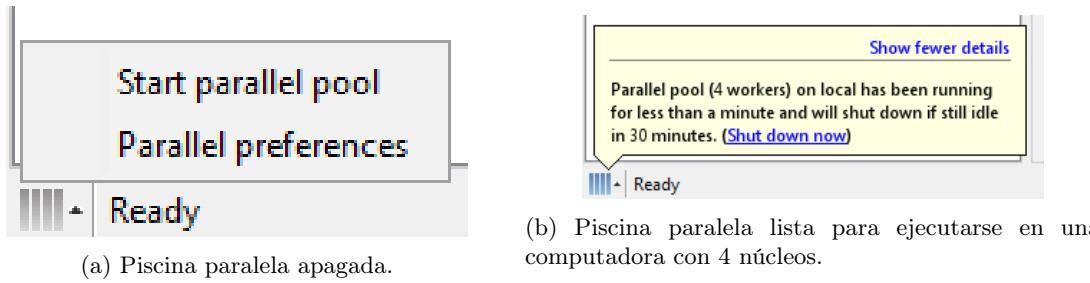


Figura 14: Piscina paralela en Matlab.

El criterio de paro del algoritmo está dividido en dos partes. La primera detiene el algoritmo cuando la frecuencia de la moda corresponde a un ϵ en porcentaje (0-1) y el segundo criterio es un número máximo de iteraciones. Los parámetros utilizados en este experimento se muestran en el Cuadro 2.

Parámetro	Valor
ρ	0.5
α	1
β	1
Q	2
tf	70
ϵ	0.9
τ_0	1

Cuadro 2: Parámetros utilizados en AS paralelizado.

7.3. Ant System paralelizado con PRM

Además de la representación del espacio como cuadrícula, también se incluyó la representación por medio de *probabilistic road maps* (PRM). Esta implementación se realizó utilizando la función del *Toolbox* de Robótica de Peter Corke [33], por lo que necesita instalarlo antes de ejecutar el algoritmo. Se generó un grafo con la función *prm_generator* y se guardó en un archivo *.mat*, para luego cargarlo en Matlab y utilizar el mismo mapa en cada ejecución. La función antes mencionada fue codificada a partir de la función *prm* del *Toolbox* de Peter Corke, realizando modificaciones en el grafo para adaptarlo al *Ant Colony*.

Puede existir un problema de compatibilidad entre el *Toolbox* de Peter Corke y algunas funciones de Matlab. Este problema yace en que algunas funciones de Peter Corke tienen el mismo nombre que otras funciones intrínsecas de Matlab. Este problema se resuelve al modificar el orden del *Matlab search path*, colocando las funciones de Peter Corke hasta arriba.

El experimento de esta sección consistió en ejecutar el programa 10 veces, al igual que en los experimentos anteriores. En este caso se utilizaron los parámetros del Cuadro 2, pero con una cantidad de hormigas igual a 100 (en los otros casos fue de 50) y un tf de 200 iteraciones.

7.4. Ant System paralelizado con RRT

Al igual que en el experimento anterior, se utilizó el *Toolbox* de Peter Corke, por lo que los problemas mencionados se comparten. Para hacer uniformes las ejecuciones de esta representación del mapa se generó un grafo con la función *rrt_generator* y se guardó en un archivo *.mat*, para luego cargarlo en Matlab y así hacer similares las corridas. La función antes mencionada fue codificada a partir de la función de RRT del *Toolbox* de Peter Corke, realizando modificaciones en el grafo para adaptarlo al *Ant Colony*.

El experimento de esta sección consistió en ejecutar el programa 10 veces, al igual que en los experimentos anteriores. En este caso se utilizó los parámetros del Cuadro 2 con 50 hormigas y un tf de 200 iteraciones.

7.5. Ant System con grafo de visibilidad

Adicionalmente, se probó el *Ant System* con un espacio con obstáculos con un grafo de visibilidad. Sin embargo, esta implementación no considera la dimensión del robot, por lo que se asume que es una masa puntual y el espacio de trabajo es el mismo que el de configuración. Para una implementación más realista es necesario tomar en cuenta la dimensión del robot en los obstáculos y hacerlos más anchos.

El grafo de visibilidad se crea con una aplicación llamada `poly_graph.mplapp` que debe abrirse con el *App Designer* de Matlab. Para editar el programa se ejecuta en la línea de comandos de matlab el comando: `appdesigner` y luego se abre el archivo. Para solamente ejecutar el programa y generar el grafo se puede hacer (doble) clic desde la carpeta actual de Matlab. Primero se debe de presionar el botón de `Add Obstacle` y dibujar los polígonos que se desee, luego se presiona `add start point` para agregar un punto de inicio, `add end point` para agregar un punto final y finalmente `visibility graph` para generar el grafo y guardarlo en un archivo .mat en la carpeta donde se está actualmente. En el archivo principal de ACO se importa el último archivo generado para utilizarlo como mapa.

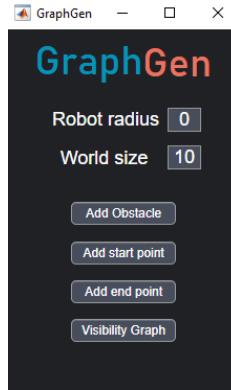


Figura 15: Aplicación que genera grafos de visibilidad.

Se realizó la interfaz de la aplicación para que fuera posible hacer obstáculos con dibujos personalizados por el usuario. De esta forma, cuando se quiera probar con una cámara, se puede hacer el trazo a mano sobre la imagen o incluso con el mismo Matlab se procese la imagen con visión por computadora. Las funciones de Matlab que hacen esto son similares a la utilizada ahora, por lo que en un futuro no debería de ser tan difícil la implementación. La aplicación también cuenta con un espacio para colocar el radio del robot para que en el futuro se pueda implementar con restricciones físicas.

El experimento consistió en ejecutar el algoritmo 10 veces, al igual que en los experimentos anteriores, con 50 hormigas y los parámetros mostrados en el Cuadro 2, exceptuando t_f que fue de 200.

7.6. Elección de parámetros

En este experimento se planificó ejecutar el algoritmo en la computadora de alto rendimiento², realizando un barrido de parámetros para ρ , α y β . Posteriormente se realizó un barrido con Q y el número de hormigas. Todo esto, con el fin de encontrar los mejores parámetros para el AS. Los parámetros que se consideraron como *mejores* fueron los que mostraban un balance entre un tiempo de ejecución rápido y exactitud en encontrar el camino óptimo. Sin embargo, también se realizó el barrido de parámetros eligiendo los valores límite en rapidez para ρ (por ser el primer parámetro).

²En el anexo se presentan las especificaciones de dicha computadora.

Cada simulación se ejecutó 150 veces para cada valor en el rango de valores presentados en el Cuadro 3. Dichos rangos se eligieron para que el tiempo de los barridos no fuera excesivo, variando entre 3 y 24 horas. Se hicieron pruebas preliminares para determinar qué parámetros sí valía la pena considerar. Estas consistieron en hacer un barrido de varios valores, pero con pocas repeticiones (de 2-10 dependiendo del tiempo) en una computadora convencional de 4 núcleos. Todas las pruebas se realizaron con paralelización (**parfor**) exceptuando el barrido de hormigas. Este no se logró paralelizar debido a que no cumplía con la regla de independencia que requiere la paralelización en Matlab.

Parámetro	Rango	Paso
ρ	(0.3, 0.9)	0.1
α	(0.9, 1.5)	0.5
β	(0.9, 1.5)	0.5
Q	(1.2, 2.4)	0.1
Hormigas	(50, 100)	10

Cuadro 3: Rango en el que se realizó el barrido de parámetros.

7.7. Efecto de la paralelización

En este experimento se planificó ejecutar el algoritmo AS con los mejores parámetros (mostrados en el Cuadro 4) en la computadora de alto rendimiento, pero variando el número de núcleos utilizados. Esta variación se realizó en el rango de 1 núcleo (sin paralelización) hasta el máximo (44 núcleos). De esta manera, se busca comprobar si en realidad se decremente el tiempo de ejecución al incrementar el número de núcleos utilizados. El algoritmo AS se ejecutó 50 veces con cada número de núcleos (por cuestión de tiempo no se ejecutó más veces). Luego se calculó la media y desviación estándar para cada núcleo y finalmente se presenta una gráfica con estos parámetros en la sección 8.7.

Parámetro	Valor
ρ	0.6
α	1
β	1
Q	2.1
Hormigas	60

Cuadro 4: Parámetros utilizados en el barrido de núcleos.

CAPÍTULO 8

Resultados Ant System

En este capítulo se muestran los resultados del capítulo anterior (7.2). Estos se presentan en el mismo orden con el que se expusieron anteriormente: validación del algoritmo Ant System, validación del algoritmo Ant System paralelizado, Ant System paralelizado con PRM, RRT y grafos de visibilidad.

8.1. Validación del algoritmo Ant System

A continuación se muestra un camino óptimo y una imagen de la simulación de feromonas hallada por el programa. Los resultados de las 10 ejecuciones del programa se muestran en el Cuadro 5.

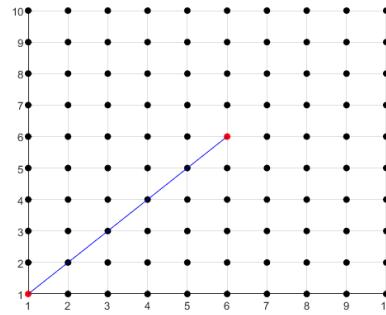


Figura 16: Camino óptimo encontrado del nodo (1,1) al (6,6).

En este ejemplo el costo mínimo que se podía obtener era 2.5, ya que el costo de las diagonales era de 0.5. El tiempo fue medido con las funciones `tic` y `toc` de Matlab. Al final de la Cuadro 5 se muestra la media de cada parámetro. Para cada ejecución es posible

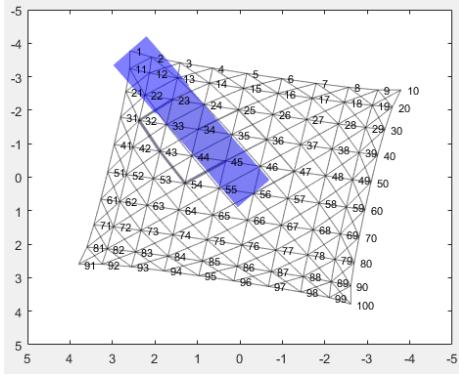


Figura 17: Feromona del camino óptimo encontrado del nodo (1,1) al (6,6).

iteración	tiempo (s)	costo
30	47.72	2.5
33	56.22	2.5
27	46.65	2.5
54	97.47	3.5
24	43.62	2.5
25	44.37	2.5
53	96.31	2.5
70	125.35	2.5
42	69.83	2.5
42	70.58	2.5
40	69.81	2.6

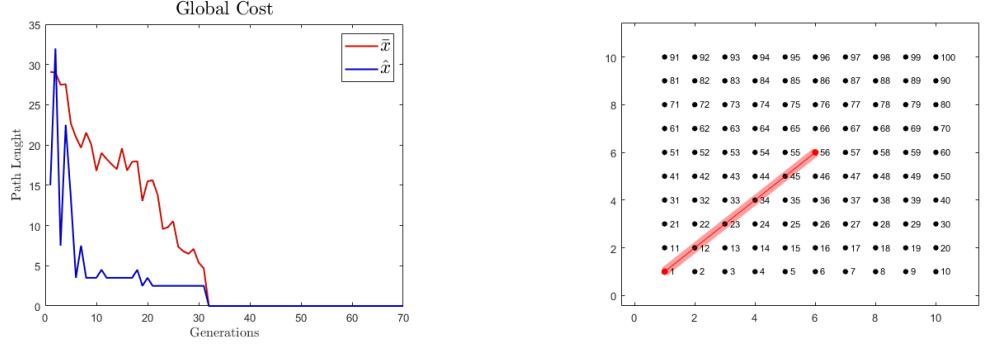
Cuadro 5: Tiempo y costo de 10 ejecuciones de AS.

observar que el tiempo aumenta cuando las hormigas toman malas decisiones al principio. Puede que esto se arregle modificando la tasa de evaporación para aumentar el tiempo de exploración de las hormigas. Como puede observarse en la Figura 17, la cantidad de feromona se ve reflejada en el ancho del camino. Además, los nodos no tienen un orden con respecto al plano X, Y, por lo que en el siguiente experimento también se hicieron esas mejoras gráficas.

8.2. Validación del algoritmo Ant System paralelizado

Al ejecutar el código se ven dos gráficas animadas. La Figura 1 muestra la media \bar{x} del largo del camino y la moda \hat{x} del mismo. La segunda Figura tiene una animación de los caminos y su feromona. Esta última animación cambia de color y de grosor de línea en cada arista que tiene mayor τ^1 . Dentro del código se agrega un nuevo parámetro al atributo **Edges** que guarda la cantidad de feromona normalizada para utilizarla en esta animación. El color rojo indica mayor presencia de feromona, mientras que el color blanco indica que esa arista tiene menor cantidad. Los resultados de las 10 ejecuciones descritas en el capítulo anterior se muestran en el Cuadro 6.

¹Ejemplos de estas gráficas se muestran en la Figura 18



(a) Media y moda del largo de los caminos. (b) Animación de feromona y camino encontrado.

Figura 18: Animaciones generadas con mundo cuadrículado.

iteración	tiempo	costo
14	14.2161	2.5
10	9.6620	2.5
09	9.2031	2.5
18	49.5735	3.5
43	44.7586	3.5
17	16.0491	2.5
26	25.3741	3.5
11	26.8102	2.5
22	60.7632	3.5
13	12.3290	3.5
18.3	26.8739	3

Cuadro 6: Tiempo y costo de 10 ejecuciones de AS paralelizado.

En contraste con el algoritmo sin paralelización, este algoritmo es 2.6 veces más rápido (comparando sus medias).

8.3. Ant System paralelizado con PRM

Al ejecutar el código se ven dos gráficas animadas iguales a las del experimento anterior, con la diferencia de que estas se realizaron con un mapa con PRM. Los resultados de las 10 corridas descritas en el capítulo anterior se muestran en el Cuadro 7.

8.4. Ant System paralelizado con RRT

Al ejecutar el código veremos dos gráficas animadas iguales a las del experimento anterior, con la diferencia de que estas se realizaron con RRT. Los resultados de las 10 ejecuciones descritas en la sección anterior (sec. 8.3) se muestran en el Cuadro 8.

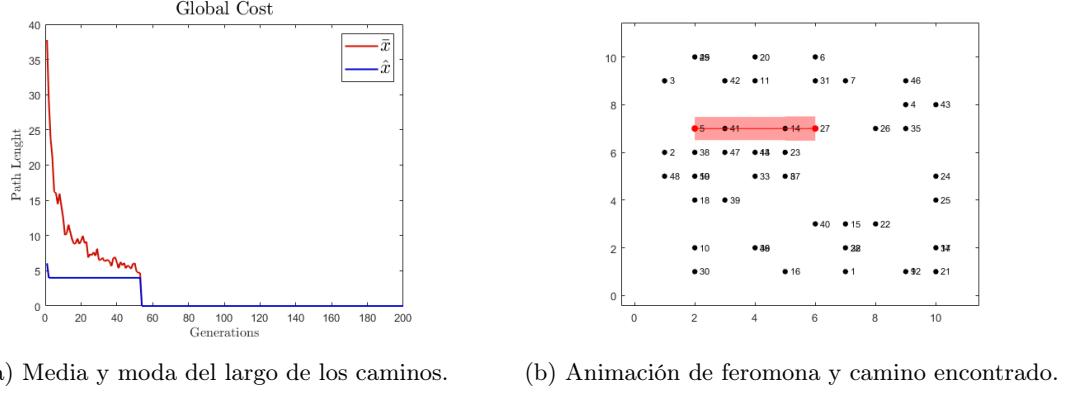


Figura 19: Animaciones generadas con PRM.

iteración	tiempo	costo
69	79.30	4.00
146	152.03	6.00
105	117.28	4.00
82	88.59	4.00
83	87.08	6.00
192	211.31	6.00
178	206.50	6.00
95	106.60	6.00
117	130.32	6.00
53	53.20	4.00
112	123.22	5.2

Cuadro 7: Resultados del AS con PRM.

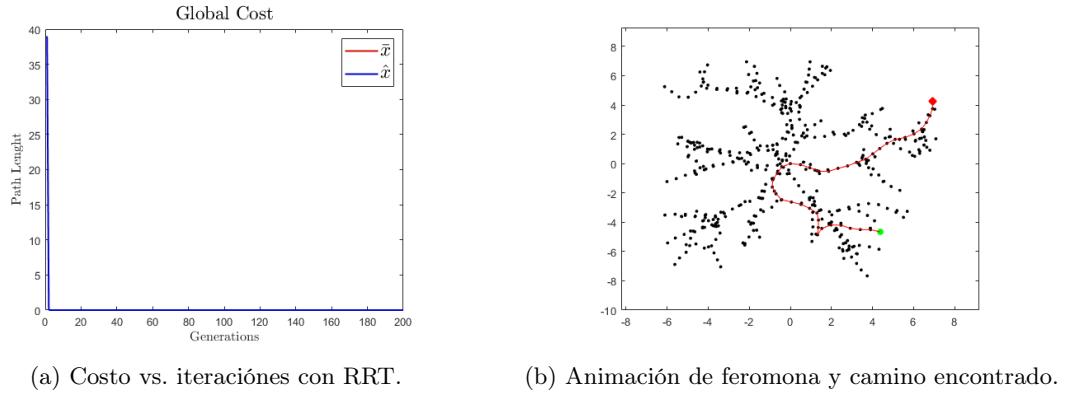


Figura 20: Animaciones generadas con RRT.

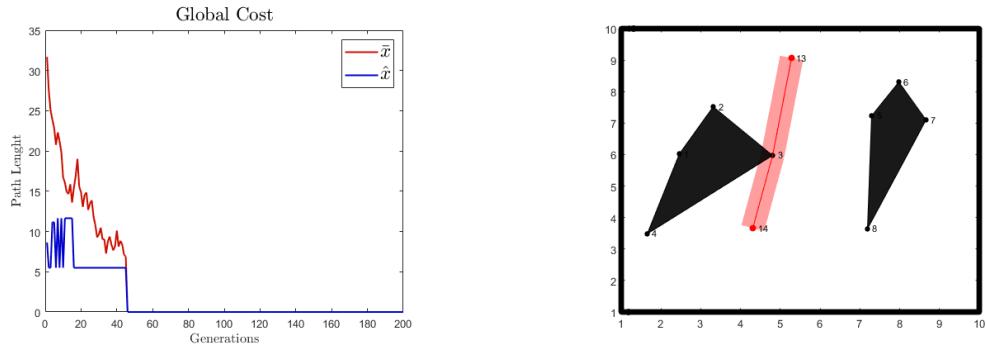
8.5. Ant System con grafo de visibilidad

Al ejecutar el código veremos dos gráficas animadas iguales a las del experimento anterior, con la diferencia de que estas se realizaron con grafos de visibilidad. En la Figura 21b puede observar los obstáculos y el mejor camino generado. Los resultados de las 10 ejecuciones

iteración	tiempo	costo
1	21.26	39
1	21.26	39
1	20.66	39
1	18.71	39
1	21.05	39
1	17.51	39
1	15.67	39
1	23.45	39
1	19.27	39
1	15.50	39
1	23.45	39
1	15.16	39
1	17.34	39
1	24.73	39
1	20.06	39
1	17.81	39
1	18.74	39
1	19.72	39
1	18.39	39
1	18.54	39
1	19.4140	39

Cuadro 8: Resultados del AS con RRT.

descritas en el capítulo anterior se muestran en el Cuadro 9.



(a) Media y moda del largo de los caminos.

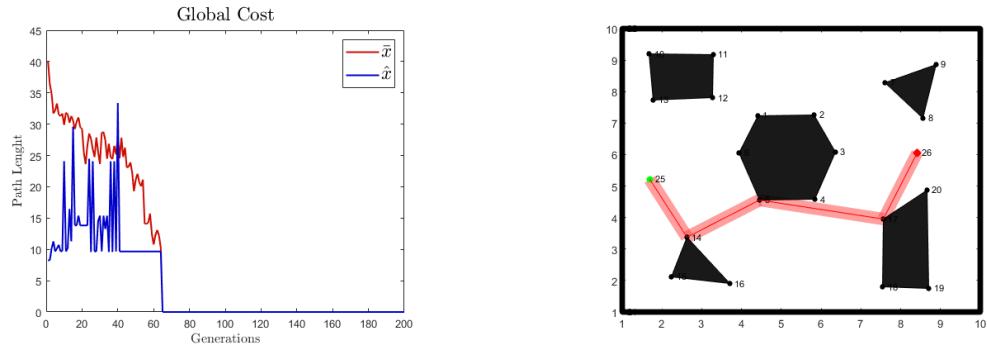
(b) Animación de feromonas y camino encontrado.

Figura 21: Animaciones generadas con grafo de visibilidad.

iteración	tiempo	costo
56	23.46	7.88
114	42.96	7.88
41	17.48	5.49
129	47.03	7.88
100	38.04	7.88
82	30.19	5.49
36	21.72	7.88
172	65.71	7.88
45	31.17	5.49
87	32.18	5.49
86.2	34.99	6.92

Cuadro 9: Resultados del AS con grafo de visibilidad.

Adicionalmente se realizó una prueba donde el nodo inicial y final están separados por un obstáculo para validar el funcionamiento del algoritmo. El costo por iteración y el camino encontrado para esta prueba se puede observar en la Figura 22.



(a) Media y moda del largo de los caminos. (b) Animación de feromona y camino encontrado.

Figura 22: Animaciones generadas con grafo de visibilidad.

8.6. Elección de parámetros

En esta sección se presentan los resultados de los experimentos explicados en la sección 7.6. En los cuadros la columna t representa la media del tiempo (en segundos) de las 150 ejecuciones con su respectivo ρ . Asimismo, la columna de iteraciones representa la media de las iteraciones, la columna costo representa cuántas ejecuciones encontraron el costo óptimo. Este en el mejor de los casos debería de ser 150. La columna de fallos cuenta la cantidad de veces que el algoritmo necesitó más de 150 iteraciones, por lo que no se tuvo convergencia. A continuación también se mostrarán gráficas de barras con el costo encontrado. En el caso óptimo debería de verse que en las 150 ejecuciones se encontró 2.5 como costo óptimo.

8.6.1. Rho

Como se explicó en la sección 7.6, se realizaron 150 ejecuciones del algoritmo con distintos valores para ρ . El resultado de estas ejecuciones puede resumirse en el Cuadro 10. Puede observarse cómo el tiempo es mayor cuando el valor de ρ es menor (Figura 24). Sin embargo, también se observa que se encuentra el óptimo en más ocasiones cuando el tiempo es mayor. En la Figura 23 se puede apreciar cómo al incrementar el valor de ρ se encuentran distintos valores de costo, en lugar de encontrar únicamente el camino óptimo con 2.5 de costo.

ρ	$t(s)$	iteraciones	costo	fallo
0.30	17.33	55.15	148	2
0.40	13.22	43.59	147	0
0.50	12.10	39.81	135	0
0.60	10.40	34.57	130	0
0.70	09.74	32.83	127	0
0.80	08.28	27.85	121	0
0.90	07.99	27.25	113	0

Cuadro 10: Resultados del barrido de ρ .

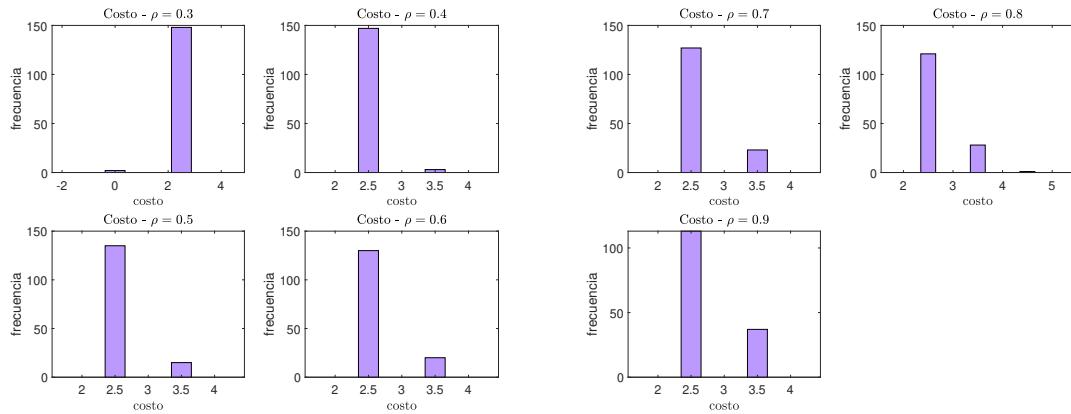


Figura 23: Costo para cada valor de ρ .

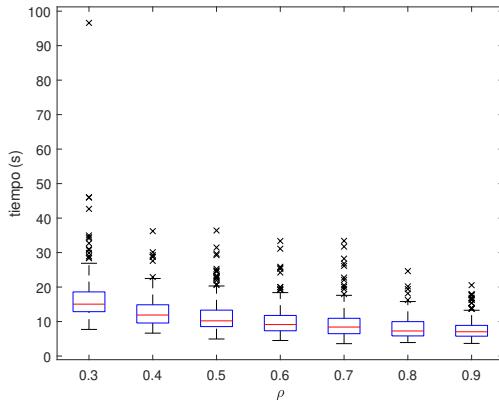


Figura 24: Tiempo para cada valor de ρ .

Para las siguientes ejecuciones se elegirá $\rho = 0.6$ como valor intermedio, pues tiene un balance entre costo y tiempo de ejecución. Asimismo, se utilizará otros dos valores para verificar el comportamiento de los parámetros. El primer valor fue $\rho = 0.4$, debido a que entre 0.3 y 0.4 solo hay una unidad de diferencia de costo, pero 4 segundos de diferencia. El siguiente valor fue $\rho = 0.8$, porque la diferencia entre 0.8 y 0.9 en costo es de 8, mientras que en tiempo es casi de 0.2 s.

8.6.2. Alpha

La primera ejecución que se realizó en este experimento se realizó con $\rho = 0.6$. Los resultados pueden observarse en el Cuadro 11. Al igual que para ρ , conforme se aumenta el valor del parámetro se decremente el tiempo (26). A pesar de esto, la elección del valor óptimo de costo se ve damnificado al aumentar α . En la Figura 25 se puede observar el fenómeno comentado anteriormente, pues las gráficas pasan de tener casi todas las ejecuciones con costo óptimo a no óptimo.

α	t(s)	iteraciones	costo	fallo
0.90	14.87	49.37	150	0
1.00	9.99	33.64	133	0
1.10	7.47	25.78	101	0
1.20	5.98	20.69	84	0
1.30	4.50	15.61	85	0
1.40	4.21	14.65	63	0
1.50	3.61	12.62	75	0

Cuadro 11: Resultados del barrido de α para $\rho = 0.6$.

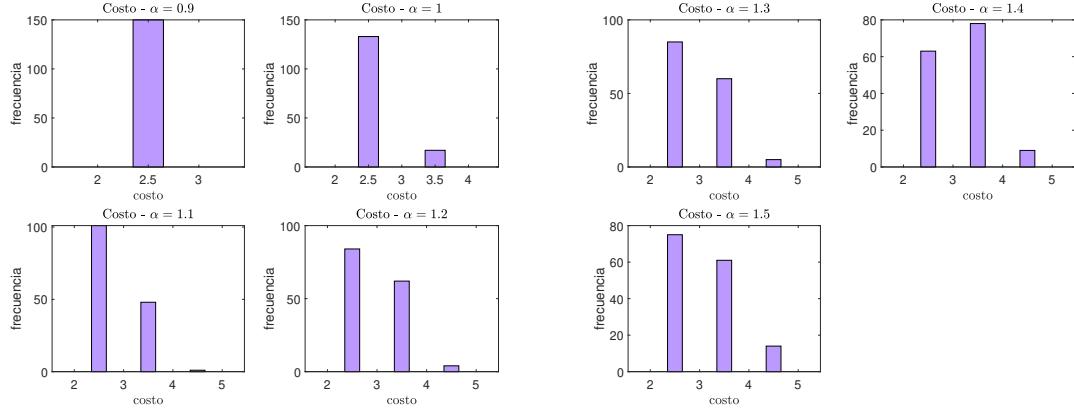


Figura 25: Costo del barrido de α para $\rho = 0.6$.

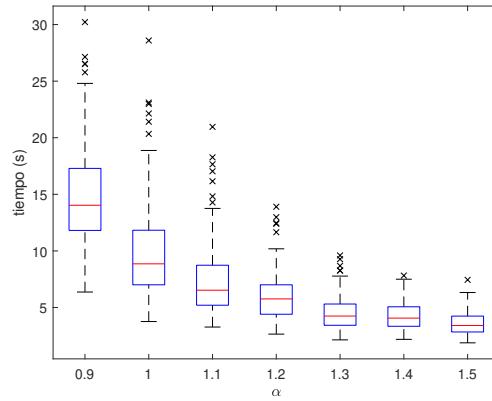


Figura 26: Tiempo para cada valor de α para $\rho = 0.6$.

En comparación con los resultados de ρ puede observarse que el tiempo promedio máximo bajó en 3 segundos aproximadamente. Este tiempo máximo fue resultado de utilizar $\alpha = 0.9$. Este valor no tuvo ningún fallo y tuvo un costo perfecto de $\frac{150}{150}$ correctos. Sin embargo, el tiempo entre los valores 0.9 y 1 para α tiene una diferencia de casi 5 segundos. Además, el costo generado por un α de 1 fue 88.6 % satisfactorio. Por tanto, para ahorrar un poco de tiempo sacrificando costo se eligió proseguir las siguientes ejecuciones con $\alpha = 1$.

La segunda ejecución del barrido de α se realizó con $\rho = 0.4$. Los resultados pueden observarse en el Cuadro 12. Al igual que para ρ , conforme se aumenta el valor del parámetro se decremente el tiempo (28). A pesar de esto, la elección del valor óptimo de costo se ve damnificado al aumentar α . En la Figura 27 se puede observar el fenómeno comentado anteriormente, pues las gráficas pasan de tener casi todas las ejecuciones con costo óptimo a no óptimo.

α	$t(s)$	iteraciones	costo	fallos
0.90	21.16	68.89	150	0
1.00	12.74	42.35	144	1
1.10	09.81	33.13	116	0
1.20	07.65	25.91	102	0
1.30	06.21	21.27	95	0
1.40	04.99	16.93	84	0
1.50	04.48	15.50	76	0

Cuadro 12: Resultados del barrido de α para $\rho = 0.4$.

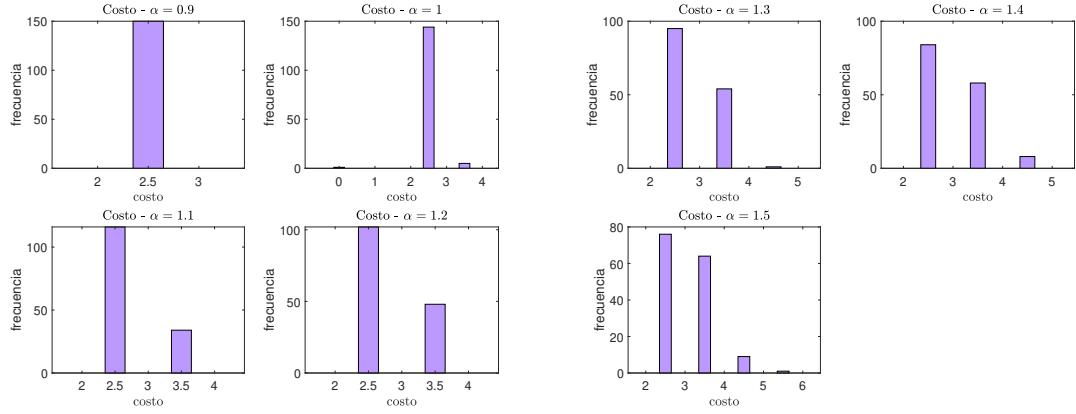


Figura 27: Costo del barrido de α para $\rho = 0.4$.

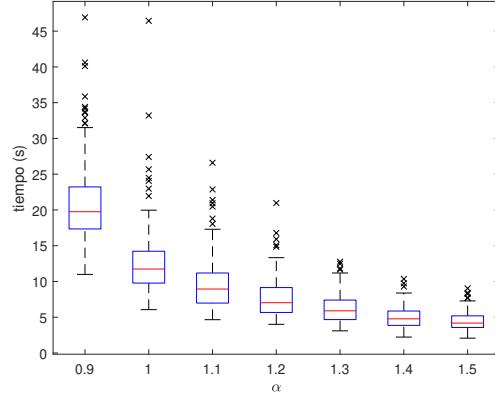


Figura 28: Tiempo para cada valor de α para $\rho = 0.4$.

A diferencia de las ejecuciones con $\rho = 0.6$ la media del tiempo del α más pequeño es más grande (alrededor de 20s). Esto concuerda con el hecho de que este fuera el que producía que el AS fuera más lento. Para las siguientes ejecuciones se utilizó un $\alpha = 1$ debido a que tiene el mejor costo contra tiempo.

La tercera ejecución del barrido de α se realizó con $\rho = 0.8$. Los resultados pueden observarse en el Cuadro 13. Al igual que para ρ , conforme se aumenta el valor del parámetro se decremente el tiempo (30). A pesar de esto, la elección del valor óptimo de costo se ve damnificada al aumentar α . En la Figura 29 se puede observar el fenómeno comentado anteriormente, pues las gráficas pasan de tener casi todas las ejecuciones con costo óptimo a no óptimo.

α	$t(s)$	iteraciones	costo	fallos
0.90	12.59	41.31	145	1
1.00	08.28	28.16	122	0
1.10	05.74	19.59	113	0
1.20	04.64	15.97	81	0
1.30	03.82	13.35	78	0
1.40	03.40	11.88	66	0
1.50	03.04	10.59	55	0

Cuadro 13: Resultados del barrido de α para $\rho = 0.8$.

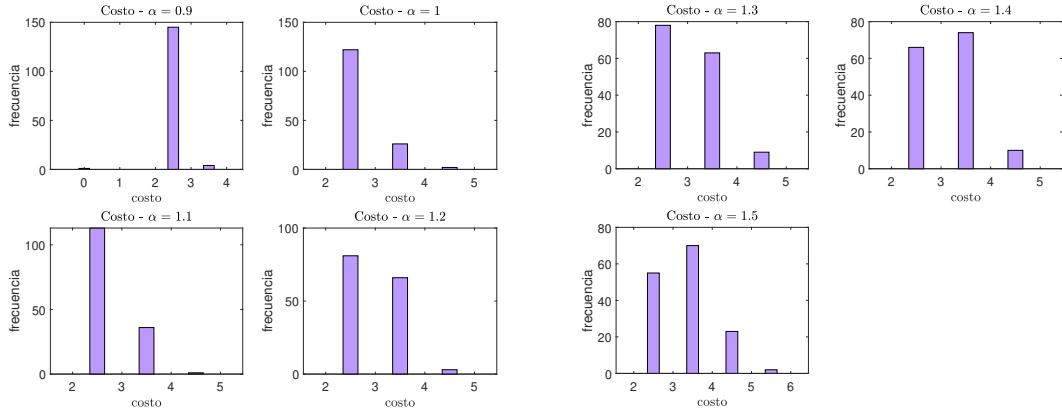


Figura 29: Costo del barrido de α para $\rho = 0.8$.

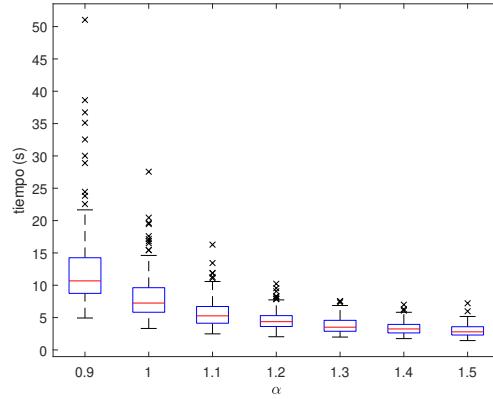


Figura 30: Tiempo para cada valor de α para $\rho = 0.8$.

En comparación con los resultados anteriores el tiempo medio para el caso más pequeño de α es menor (alrededor de 12s). Esto es alrededor de 10 segundos menos que el caso $\rho = 0.4$ y alrededor de 3 segundos menos que $\rho = 0.6$. Este tiempo máximo fue resultado de utilizar $\alpha = 0.9$. Este valor tuvo un fallo y tuvo un costo de $\frac{145}{150}$ correctos. Sin embargo, el tiempo entre los valores 0.9 y 1 para α tiene una diferencia de 4.3 segundos. Además, el costo generado por un α de 1 fue 81.3 % satisfactorio. Se eligió proseguir las siguientes ejecuciones con $\alpha = 0.9$ porque este contaba con el costo 96.6 % satisfactorio con solo 4 segundos de diferencia.

8.6.3. Beta

La primera ejecución que se realizó en este experimento se realizó con $\rho = 0.6$. Los resultados pueden observarse en el Cuadro 14. Al diferencia que para α , conforme se aumenta el valor del parámetro no se decremente el tiempo (32). A pesar de esto, la elección del valor óptimo de costo se ve damnificada al aumentar β (a excepción de los casos $\beta = 1.2 - 1.4$). En la Figura 31 se puede observar el fenómeno comentado anteriormente.

β	t(s)	iteraciones	costo	fallo
0.90	10.78	33.02	133	0
1.00	10.47	33.77	133	1
1.10	10.10	32.79	131	0
1.20	10.48	33.95	127	0
1.30	10.22	33.11	127	0
1.40	10.49	33.85	130	0
1.50	10.71	35.16	126	1

Cuadro 14: Resultados del barrido de β para $\rho = 0.6$.

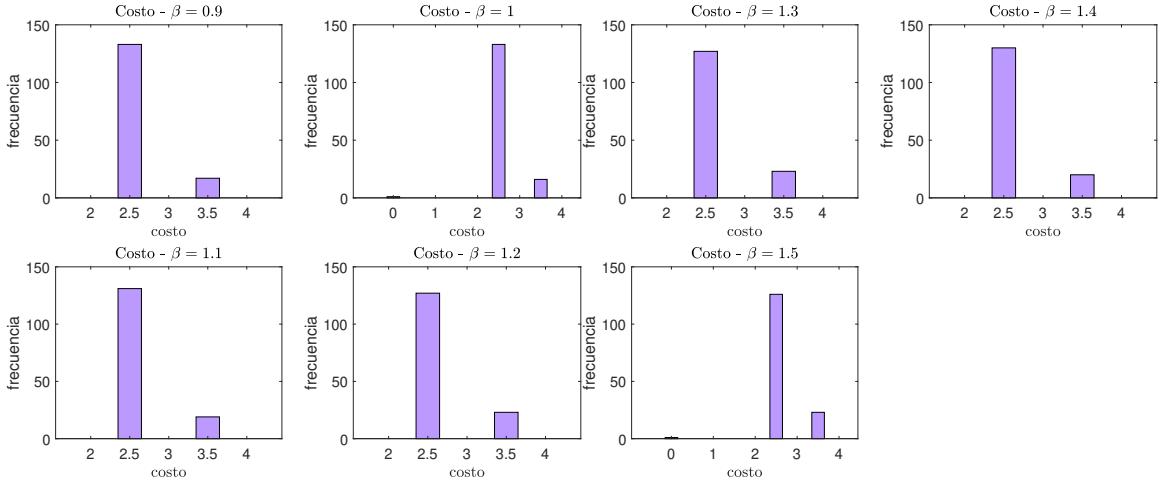


Figura 31: Costo del barrido de β para $\rho = 0.6$.

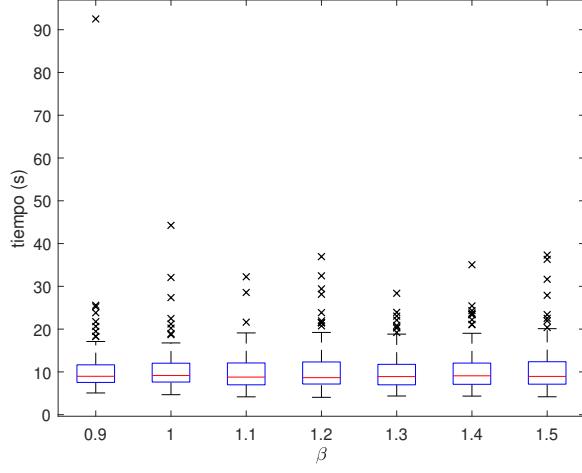


Figura 32: Tiempo para cada valor de β para $\rho = 0.6$.

El tiempo no cambia drásticamente conforme se aumenta el valor de β , por lo que para las siguientes ejecuciones se utilizará $\beta = 1$. Esto debido a que es el que tiene mejor costo con datos atípicos de tiempo menores que el $\beta = 0.9$.

La segunda ejecución que se realizó en este experimento se realizó con $\rho = 0.4$. Los resultados pueden observarse en el Cuadro 15. A diferencia que para α , conforme se aumenta el valor del parámetro no se decremente el tiempo (34). A pesar de esto, la elección del valor óptimo de costo se ve damnificada al aumentar β (a excepción de los casos $\beta = 1.2, 1.3$). En la Figura 33 se puede observar el fenómeno comentado anteriormente.

β	t(s)	iteraciones	costo	fallo
0.90	13.28	43.21	144	1
1.00	14.14	46.87	143	1
1.10	14.11	46.97	144	0
1.20	13.92	46.24	140	0
1.30	13.88	46.29	141	2
1.40	13.72	45.54	139	0
1.50	14.23	47.34	139	3

Cuadro 15: Resultados del barrido de β para $\rho = 0.4$.

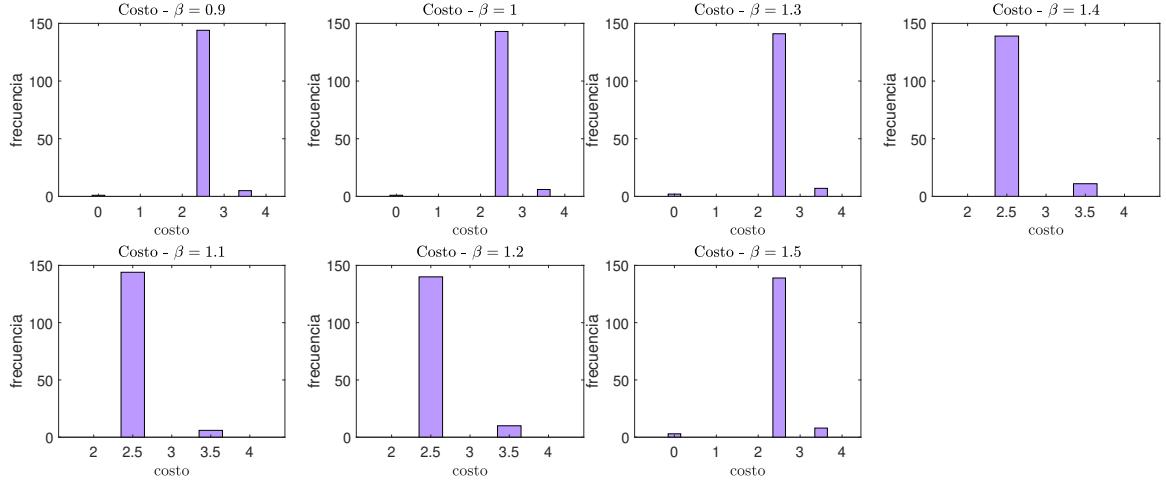


Figura 33: Costo del barrido de β para $\rho = 0.4$.

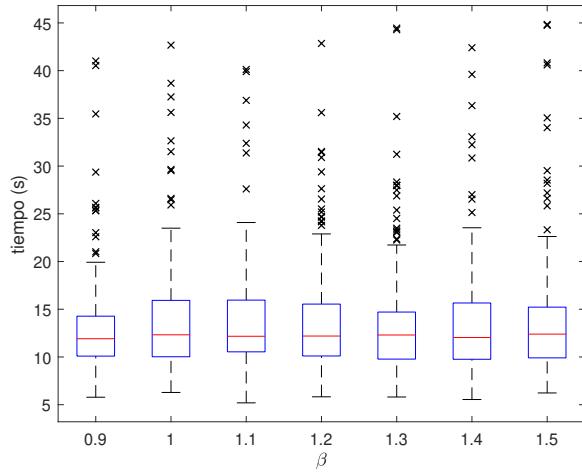


Figura 34: Tiempo para cada valor de β para $\rho = 0.4$.

Casi todos los tiempos fueron similares, por lo tanto la diferencia mayor entre cada ejecución fue el costo resultante. En este caso el $\beta = 0.9$ es el que mejor costo y menor tiempo tuvo. Por tanto, para las siguientes ejecuciones se tomará ese valor de β .

La última ejecución que se realizó en este experimento se realizó con $\rho = 0.8$. Los resultados pueden observarse en el Cuadro 16. A diferencia de los casos anteriores de β , conforme se aumenta el valor del parámetro se incrementa el tiempo (36). Además, el costo (Figura 35) para todos los casos es casi el mismo, por lo que se elegirá con base en el tiempo.

Debido a que este es el caso *rápido* (por la elección de ρ) se decidió continuar las demás ejecuciones con $\beta = 0.9$. Esto se decidió porque se quiso tratar de continuar con un valor pequeño de tiempo y comprobar si podía tenerse un tiempo bajo.

β	t(s)	iteraciones	costo	fallos
0.90	11.93	39.84	146	1
1.00	12.93	42.83	147	2
1.10	12.61	42.17	147	0
1.20	13.74	45.65	146	2
1.30	12.82	42.55	146	1
1.40	13.93	46.51	147	1
1.50	13.81	45.83	147	1

Cuadro 16: Resultados del barrido de β para $\rho = 0.8$.

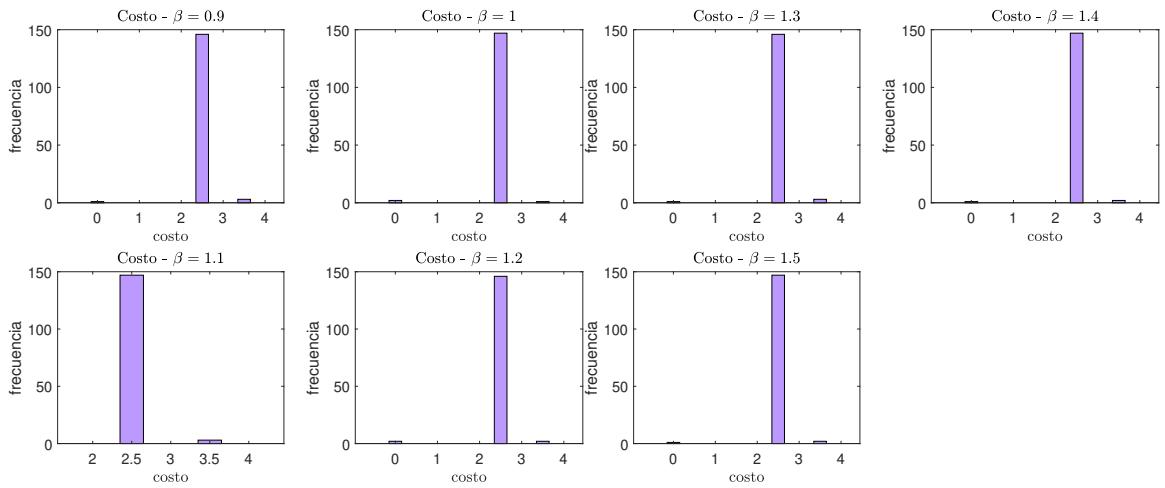


Figura 35: Costo del barrido de β para $\rho = 0.8$.

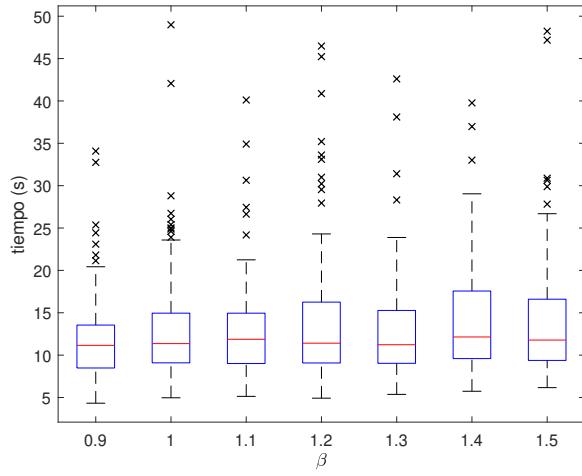


Figura 36: Tiempo para cada valor de β para $\rho = 0.8$.

8.6.4. Q

La primera ejecución de este experimento se realizó con $\rho = 0.6$. Los resultados pueden observarse en el Cuadro 17. Conforme se aumenta el valor del parámetro no se decremente el tiempo. En la Figura 38 puede verse que los tiempos de ejecución conforme se aumenta el valor de Q permanecen casi constantes. En el Cuadro 17 también se puede apreciar que también el costo fue muy similar en todos los casos. En la Figura 37 se puede observar 8 de estos 13 casos ya que al ser similares no es necesario mostrarlos todos.

Q	t(s)	iteraciones	costo	fallo
1.20	09.79	32.97	136	0
1.30	10.20	34.42	135	0
1.40	10.67	36.07	129	0
1.50	10.24	34.69	132	0
1.60	09.50	32.18	132	0
1.70	09.86	33.29	131	0
1.80	10.00	33.69	132	0
1.90	09.97	33.75	137	1
2.00	10.05	34.37	134	0
2.10	09.68	32.79	135	0
2.20	10.23	34.70	128	0
2.30	10.60	35.73	133	1
2.40	09.36	31.62	134	0

Cuadro 17: Resultados del barrido de Q para $\rho = 0.6$.

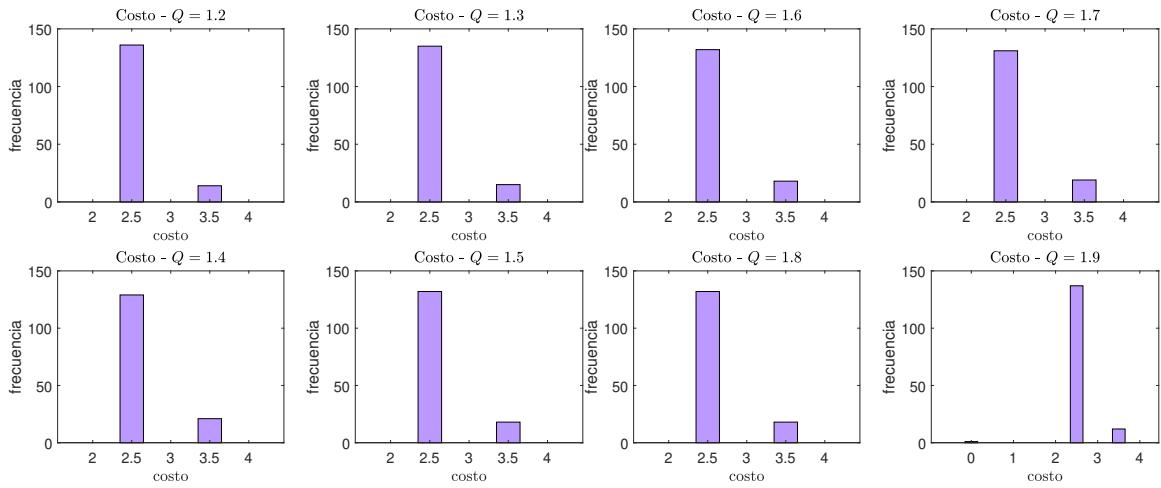


Figura 37: Costo del barrido de Q para $\rho = 0.6$.

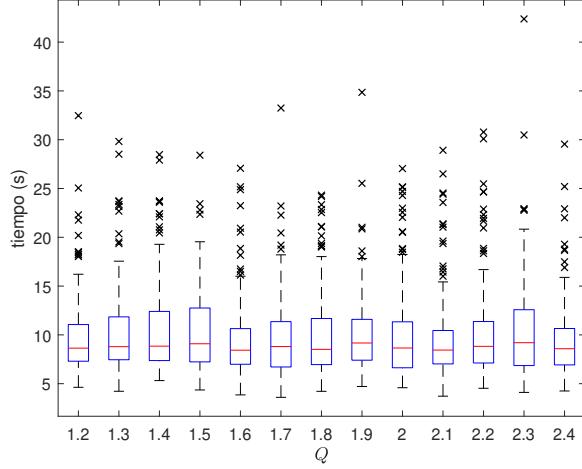


Figura 38: Tiempo para cada valor de Q para $\rho = 0.6$.

Como se mencionó anteriormente, debido a la similitud se eligió al parámetro con el menor tiempo y con costo mayor al 90 % de casos correctos. Por tanto, los candidatos fueron $Q = 1.2, 1.3, 1.9, 2.1$. De estos el que tiene el menor tiempo fue 2.1, por lo que ese valor fue elegido para las siguientes ejecuciones.

La segunda ejecución de este experimento se realizó con $\rho = 0.4$. Los resultados pueden observarse en el Cuadro 18. Al igual que el caso presentado anteriormente, el tiempo no parece cambiar de manera significativa (Figura 40). En el caso del costo tampoco se encontraron cambios bruscos. En la Figura 39 se puede observar 8 de estos 13 casos ya que al ser similares no es necesario mostrarlos todos. Por lo anteriormente descrito, se eligió $Q = 1.60$ porque que tenía el costo correcto igual a 96.7 % y con el menor tiempo.

Q	$t(s)$	iteraciones	costo	fallo
1.20	13.65	44.49	142	0
1.30	13.87	45.72	141	0
1.40	14.08	46.57	143	0
1.50	12.85	42.46	142	1
1.60	13.24	43.66	145	1
1.70	13.54	44.73	145	1
1.80	14.11	46.77	138	1
1.90	14.18	47.18	142	1
2.00	13.39	44.25	145	1
2.10	14.38	47.45	142	1
2.20	14.27	47.33	139	2
2.30	13.24	43.78	141	1
2.40	13.39	44.56	143	2

Cuadro 18: Resultados del barrido de Q para $\rho = 0.4$.

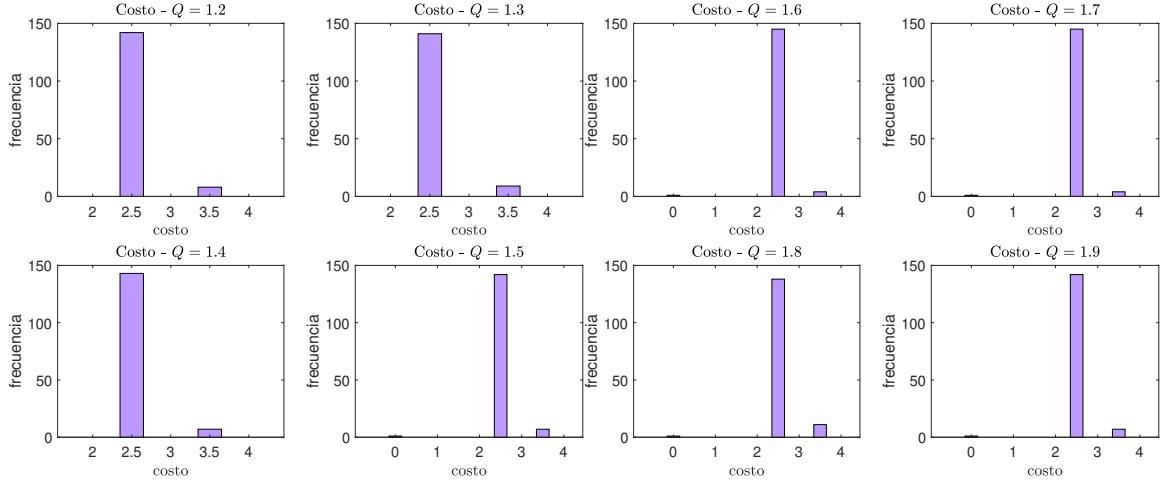


Figura 39: Costo del barrido de Q para $\rho = 0.4$.

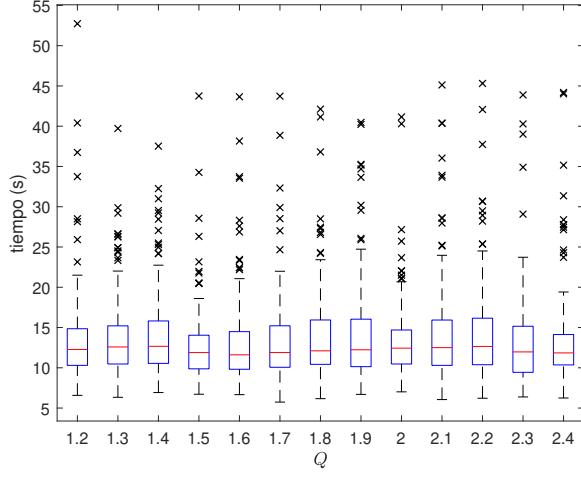


Figura 40: Tiempo para cada valor de Q para $\rho = 0.4$.

La tercera ejecución de este experimento se realizó con $\rho = 0.8$. Los resultados pueden observarse en el Cuadro 19. Al igual que el caso presentado anteriormente, el tiempo no parece cambiar de manera significativa (Figura 42). En el caso del costo tampoco se encontraron cambios bruscos. En la Figura 41 se puede observar 8 de estos 13 casos ya que al ser similares no es necesario mostrarlos todos. El costo de las ejecuciones no llegó al 90 % de casos correctos, por lo que se eligió el valor con mayor costo: $Q = 2.1$.

Q	t(s)	iteraciones	costo	fallo
1.20	11.29	36.91	120	0
1.30	12.14	40.64	119	0
1.40	13.52	45.17	116	2
1.50	11.60	38.86	117	0
1.60	12.05	40.40	118	0
1.70	13.22	44.81	114	3
1.80	12.75	42.74	118	1
1.90	12.58	42.01	120	1
2.00	11.83	39.45	120	0
2.10	12.05	40.48	121	0
2.20	12.51	41.60	118	1
2.30	13.01	43.12	118	0
2.40	13.15	44.40	116	1

Cuadro 19: Resultados del barrido de Q para $\rho = 0.8$.

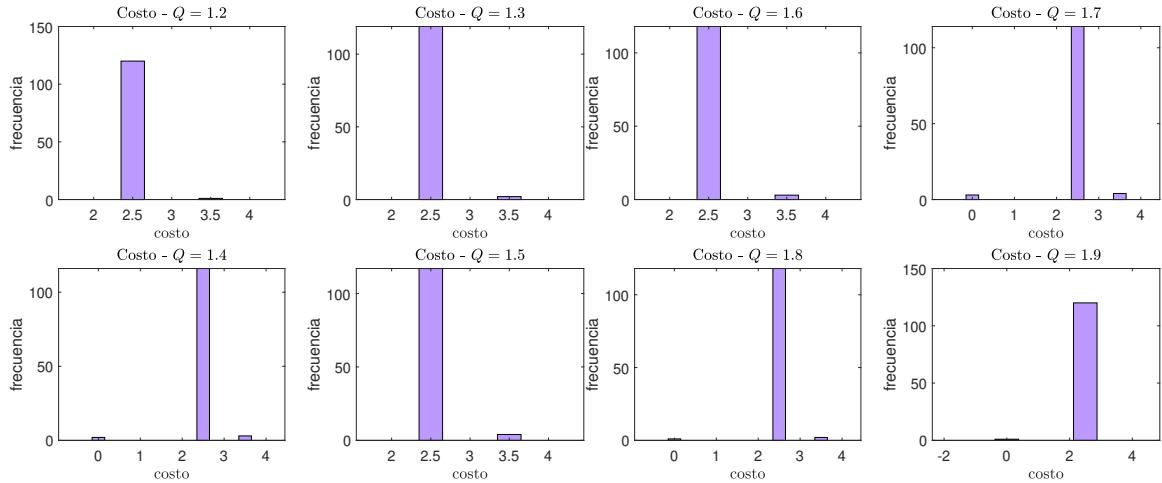


Figura 41: Costo del barrido de Q para $\rho = 0.8$.

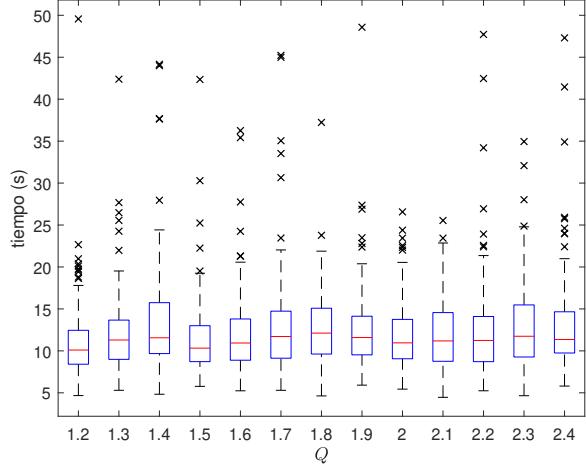


Figura 42: Tiempo para cada valor de Q para $\rho = 0.8$.

8.6.5. Número de hormigas

Para este parámetro solo se realizaron las 150 ejecuciones con el caso medio ($\rho = 0.6$), debido a que se tardó casi 24 horas. El resultado de estas puede apreciarse en el Cuadro 20. Además, puede observarse que el costo (Figura 43) y el tiempo (Figura 44) incrementan junto con el número de hormigas. El tiempo es mucho mayor que en las ejecuciones pasadas porque este barrido no podía parallelizarse. Por lo tanto, esto se realizó con solamente un núcleo. El número de hormigas elegido fue 60 porque tiene el menor tiempo, pero con un costo de 95.33 % correctos.

hormigas	t(s)	iteraciones	costo	fallos
50	62.54	33.20	131	0
60	70.23	31.07	143	0
70	88.59	33.39	138	0
80	95.13	31.23	143	0
90	106.89	31.09	144	0
100	116.20	30.89	145	0

Cuadro 20: Resultados del barrido de hormigas para $\rho = 0.6$.

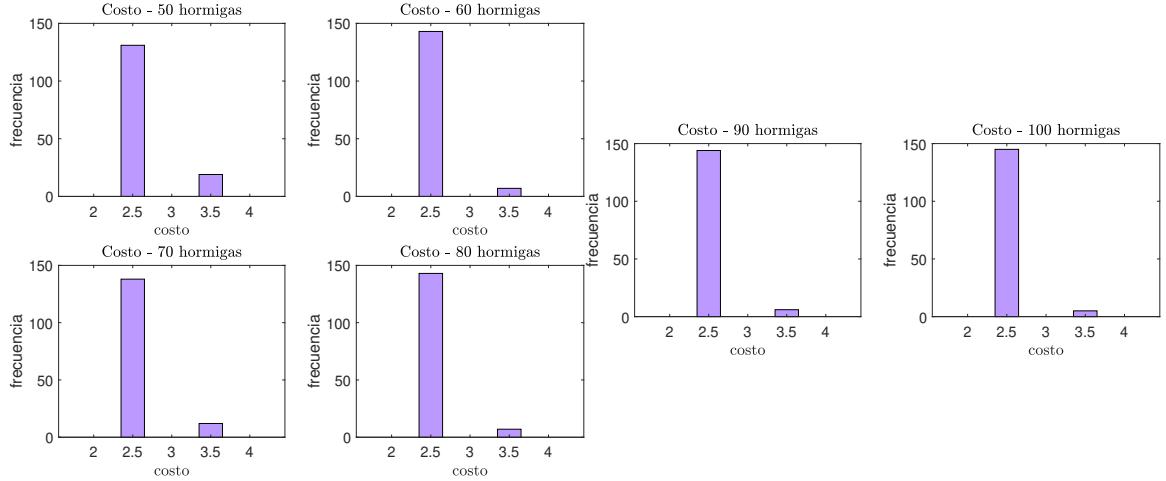


Figura 43: Costo del barrido de hormigas para $\rho = 0.6$.

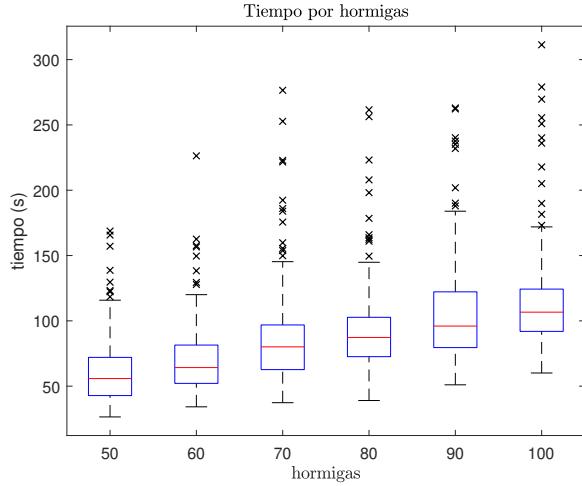


Figura 44: Tiempo para cada valor de hormigas para $\rho = 0.6$.

8.7. Efecto de la paralelización

En la Figura 45 puede observarse cómo conforme se aumenta el número de núcleos se decremente el tiempo medio de ejecución. También se puede observar que existe una gran variabilidad en el tiempo. Sin embargo, esto se justifica en el componente aleatorio que tiene el algoritmo AS. Por tanto, sería interesante probar este barrido de núcleos utilizando otros algoritmos. Además, también es posible observar que eventualmente se llega a un mínimo. Esto puede ser porque no todo en un algoritmo puede paralelizarse, por lo que existe un límite. Por lo tanto, no es siquiera necesario utilizar 44 núcleos para paralelizar este algoritmo. En el Cuadro 21 puede verse que aproximadamente con 31 núcleos se alcanzan casi los 10 segundos. Incluso con 34 núcleos se alcanzó una media menor a la calculada con 44 núcleos.

núcleos	\bar{t} (s)	σ	iteraciones
1	68.67	13.47	30.22
2	36.94	11.40	28.92
3	32.77	16.78	34.42
4	29.78	20.74	35.00
5	24.81	15.81	33.54
6	23.49	15.94	35.86
7	19.06	10.45	30.56
8	18.67	13.83	32.58
9	17.32	16.63	32.64
10	17.56	14.45	35.50
11	16.84	17.81	34.90
12	14.69	19.26	33.50
13	14.60	11.39	31.24
14	14.18	14.63	31.86
15	15.06	13.12	34.94
16	14.75	18.86	34.18
17	13.51	11.23	31.60
18	14.21	22.96	35.18
19	13.39	13.47	33.86
20	12.30	13.11	33.16
21	14.09	16.53	36.12
22	13.28	10.87	33.84
23	13.12	13.35	33.66
24	12.41	16.10	32.82
25	11.46	09.56	29.62
26	13.50	17.35	36.40
27	12.63	16.70	34.66
28	13.54	20.20	38.08
29	11.49	15.08	32.94
30	11.16	15.95	32.92
31	10.98	12.77	32.02
32	12.12	18.21	35.58
33	11.87	21.42	34.72
34	09.83	07.60	28.68
35	11.89	14.55	34.68
36	10.77	16.52	31.92
37	11.44	16.25	33.36
38	11.28	14.75	33.16
39	11.13	20.97	32.66
40	11.07	17.24	32.76
41	12.10	19.11	36.12
42	11.15	17.29	33.30
43	10.65	12.56	31.62
44	10.87	17.74	32.98

Cuadro 21: Resultados del barrido de núcleos.

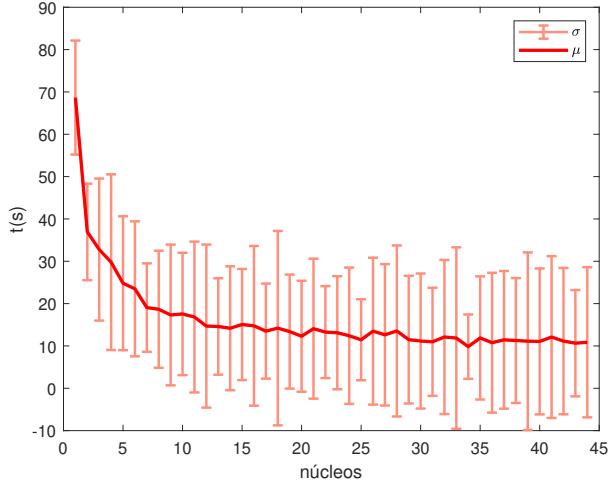


Figura 45: Barrido de núcleos para AS.

8.8. Análisis de resultados

Como pudo observarse anteriormente, las trayectorias más uniformes son generadas por el grafo en forma de cuadrícula (Figura 18b). El PRM y RRT generan trayectorias poco uniformes, especialmente el RRT. El PRM puede generar una trayectoria recta como en la Figura 19b, pero depende de los nodos inicial y final. El RRT generará trayectorias poco uniformes debido a que se genera con forma de árbol como se observa en la Figura 20b. Sin embargo, al comparar las figuras 18a, 19a y 20a puede observarse que el grafo con el que se converge más rápido es el generado con RRT. Esto se debe a que por la forma del grafo, los caminos de un nodo a otro están casi definidos. Luego, el PRM es el que presenta un camino en menos iteraciones que el grafo cuadriculado. Por lo tanto, es posible concluir que se sacrifica rapidez de convergencia por uniformidad del camino generado.

CAPÍTULO 9

Controladores punto a punto

Como se mencionó anteriormente, las pruebas realizadas con Matlab no consideraban las restricciones de los actuadores del robot. Además, tampoco se consideraba un modelo en específico de robot diferencial. Para considerar que los motores que controlan al robot tienen un límite de velocidad y garantizar que este haga lo que deseamos, es necesario implementar controladores. Por tanto, en este capítulo se introducen los controladores y sus parámetros utilizados. Además, también se comenta sobre el software utilizado para la simulación y el modelo de robot seleccionado.

9.1. Simulación por Software

Para hacer simulaciones más realistas se utilizó el software Webots R2020a revisión 1 porque fue el programa utilizado en la fase anterior. En este software se coloca una mesa de pruebas del tamaño que el investigador desee. En este caso se colocó una de 2×2 metros al igual que en la fase II del proyecto Robotat [1]. Esta mesa de pruebas tiene su propio marco de referencia, que se tomó como el inercial. Luego, es posible agregar luces, sombras y por supuesto, el robot a controlar. Ya que uno de los objetivos es comparar el algoritmo de [1] con el de esta tesis, se utilizó el mismo robot e-puck. Este último también tiene su propio marco de referencia, que no coincide con el inercial.

9.1.1. Webots

Webots es un programa muy particular, donde los ejes y ángulos se miden de forma diferente a la común. Cabe mencionar que la parte trasera del e-puck está marcada con un círculo negro. En la Figura 46 se desea medir el ángulo entre la posición actual del robot y la meta¹ marcada con una estrella amarilla en el centro del marco inercial. Este ángulo se denomina θ_g , y como puede verse en la figura, se mide en dirección de las agujas del reloj. Además, cabe mencionar que $\theta_g = \text{atan}2(\frac{\Delta z}{\Delta x})$. En la Figura 46 se tiene dos escenarios diferentes para explicar los ángulos con distintas posiciones. En la Figura 46b notamos que si se hace $-\theta_g$ quedaría $\frac{\pi}{4}$ radianes, que es un ángulo un poco más intuitivo². Asimismo, en la Figura 46a, al cambiarle el signo a θ_g se obtiene el ángulo más intuitivo de $\frac{3\pi}{2}$ medido desde el eje x en dirección contraria a las agujas del reloj.

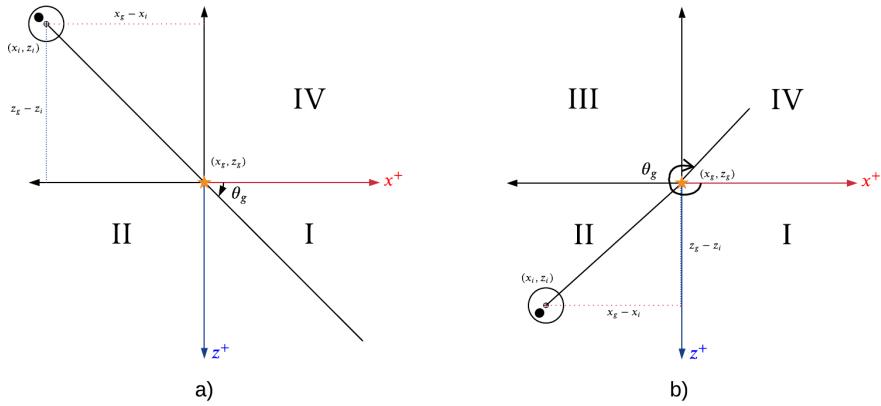


Figura 46: Ángulos en Webots.

9.1.2. Robot e-puck

Para saber la posición del robot en tiempo real se le coloca al e-puck un módulo *compass*. Esta brújula devuelve las componentes en x y z³ del vector norte en el eje del robot. El vector norte por defecto es el eje x positivo del marco inercial. Al operar el arcotangente de las componentes $\text{atan}2(\frac{x}{z})$ se obtiene el ángulo ϕ . Sin embargo, posteriormente para calcular el error de orientación se necesitará encontrar un ángulo similar a θ_g de la sección anterior. Por lo tanto, puede observarse en la Figura 47 que podemos obtener al ángulo como $\theta = \pi - \phi$.

¹La meta no necesariamente tiene que estar en el centro del marco, podría estar en otro lugar, pero se eligió este por razones didácticas.

²Como si se estuviera midiendo desde el eje x inercial en dirección contraria a las agujas del reloj.

³La componente en y también la devuelve, pero no se utiliza debido a que el eje Y apunta hacia fuera de la página.

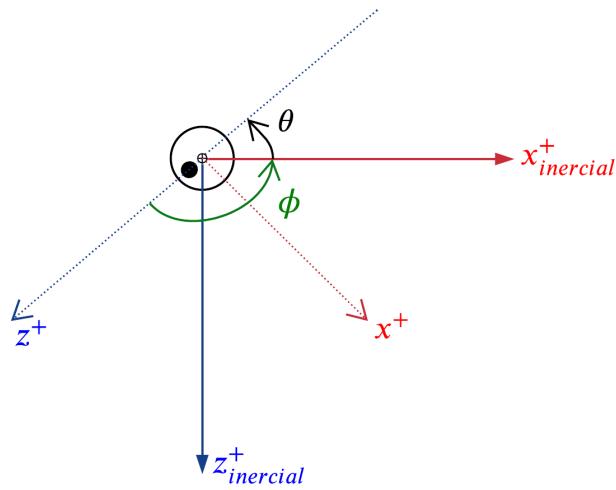


Figura 47: Ángulos del e-puck.

En lo que resta de este capítulo se presenta la implementación de los controladores mostrados en [1] de C a Matlab, para luego unirlos con el AS. En la mayoría de controladores se utilizó los mismos valores que los empleados en la fase anterior del proyecto.



Figura 48: Posición inicial de robot para pruebas con controladores.

En la Figura 48 puede verse la configuración inicial del mundo con la que se realizaron las pruebas. La parte delantera del robot se encuentra hacia abajo, con el propósito de colocarlo en un escenario difícil, pues tiene que darse la vuelta para comenzar su camino hacia la meta marcada con una estrella. La finalidad de realizar las pruebas de esta sección es descubrir qué controladores proporcionan los mejores resultados para luego ser implementados en conjunto con el AS. Las métricas que se utilizaron para medir este desempeño fueron: la variabilidad de la velocidad en las ruedas del motor, forma de la trayectoria y velocidad de convergencia del controlador.

9.2. Control TUC

Parámetro	Valor
I_x	2
I_y	2
l	35
v_o	3.12
k	$\frac{v_o(1-e^{-2eP})}{e_P}$

Cuadro 22: Parámetros utilizados para el controlador TUC.

Los valores utilizados en la implementación de la ecuación (11) son los mostrados en la Cuadro 22. En este caso e_P es el error de posición, calculado como en la ecuación (14). El control proporcional de velocidades con saturación limitada presenta poca variación y rápida convergencia en las ruedas del motor. Esto puede evidenciarse en las figuras 49a, 49b y 49c. También es posible notar que el controlador para casi de repente, aunque no de una manera tan brusca como en otros controladores que se presentarán a continuación. La mayor desventaja que posee este controlador es que la trayectoria generada no es la óptima pues presenta una ligera curvatura visible en 49d.

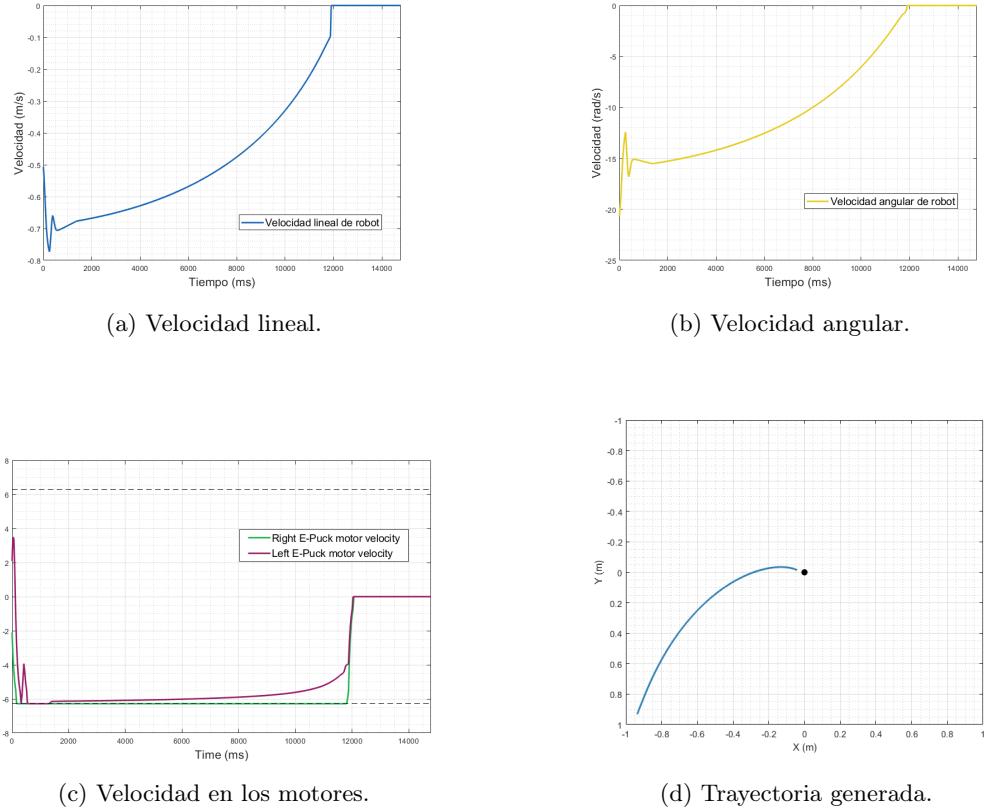


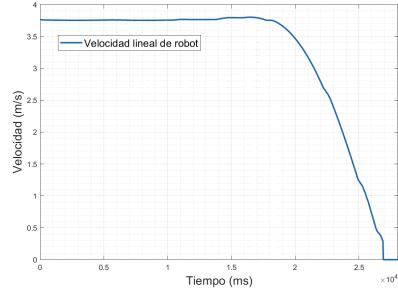
Figura 49: Gráficas de controlador TUC.

9.3. Control PID de velocidad lineal y angular

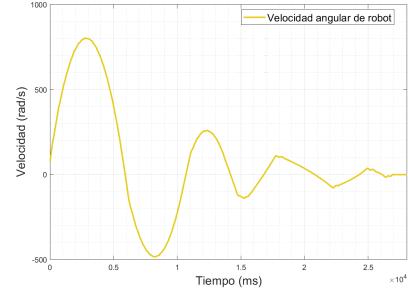
Parámetro	Valor
K_{PO}	0.01
K_{DO}	0
K_{IO}	0
K_{PP}	2
K_{DP}	0
K_{IP}	0.0001

Cuadro 23: Parámetros utilizados para el controlador PID.

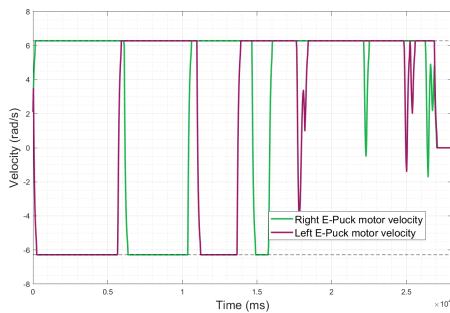
Para el PID se utilizaron los parámetros del Cuadro 23. Además, se utilizó el algoritmo presentado en la sección 6.11.2 para implementarlo en Matlab. Este controlador no converge de manera rápida. Aunque en la Figura 50a no se ven cambios bruscos, en las figuras 50b y 50c sí pueden evidenciarse los cambios bruscos de velocidad en los motores del robot. Incluso puede evidenciarse que comienza con oscilaciones y un porcentaje de sobre elevación alto. Asimismo es posible observar en la Figura 50d que la trayectoria se aleja de ser suave y óptima porque realiza cruces extraños debido a que el controlador no ha terminado de converger.



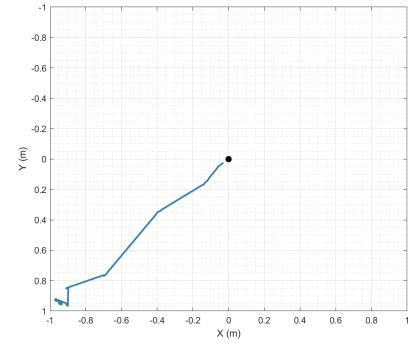
(a) Velocidad lineal.



(b) Velocidad angular.



(c) Velocidad en los motores.



(d) Trayectoria generada.

Figura 50: Gráficas de controlador PID.

9.4. Control PID de acercamiento exponencial

Los parámetros utilizados con este controlador PID son los mostrados en el Cuadro 24. Estos se sustituyeron en la ecuación (15). La velocidad lineal se ve suave (Figura 51a), como en el PID anterior. El controlador no converge de manera rápida, pero puede evidenciarse en la Figura 51b que lo hace más rápido que en el control PID de velocidad y posición. Asimismo, en la Figura 51c puede verse que la variación entre la velocidad de los motores es grande, similar al controlador anterior. En contraste con el PID est醤dar, este generó una trayectoria m醩 uniforme como puede verse en la Figura 51d.

Parámetro	Valor
K_{PO}	20
K_{DO}	5
K_{IO}	9

Cuadro 24: Parámetros utilizados para el controlador PID con acercamiento exponencial.

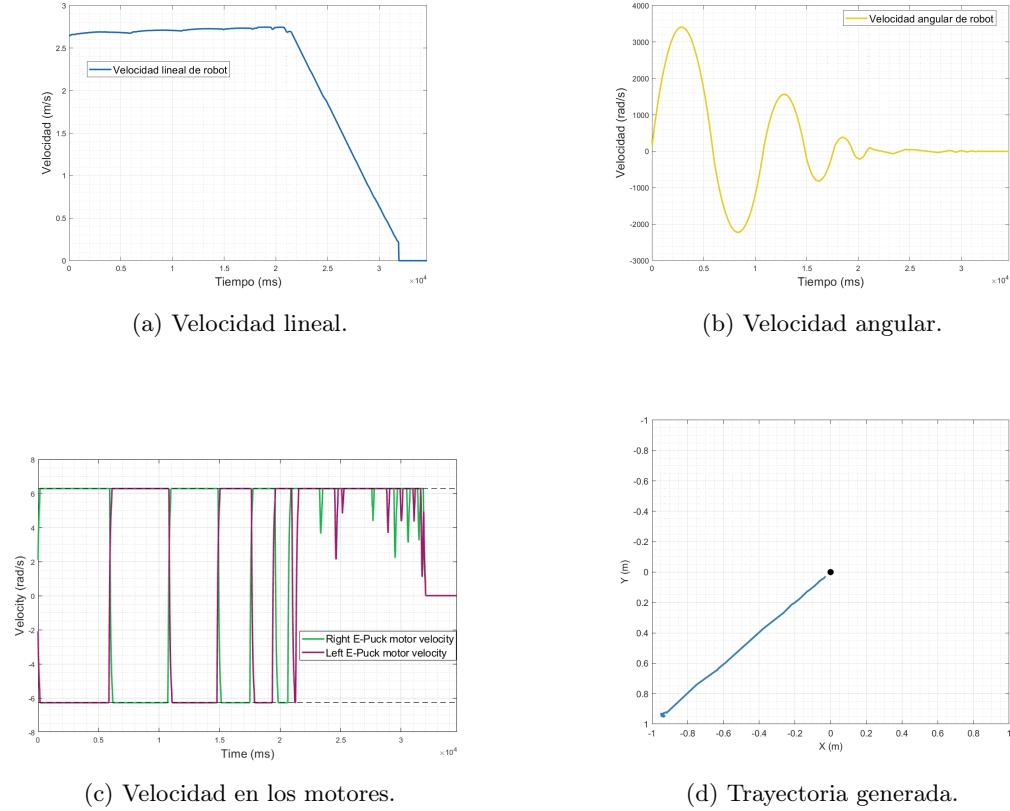


Figura 51: Gráficas de controlador PID de acercamiento exponencial.

9.5. Control de Pose

Parámetro	Valor
K_ρ	0.09
K_α	25
K_β	-0.05

Cuadro 25: Parámetros utilizados para el controlador de pose.

Los parámetros de este controlador se presentan en el Cuadro 25. Estos se utilizaron en conjunto con las ecuaciones (17-19). El control de pose presenta una aceleración alta al principio, pero conforme llega a la meta la velocidad baja hasta parar por completo. Este comportamiento suave puede apreciarse en las figuras 52a, 52b y 52c. Además, este controlador presenta una trayectoria recta y suave, como puede observarse en la Figura 52d.

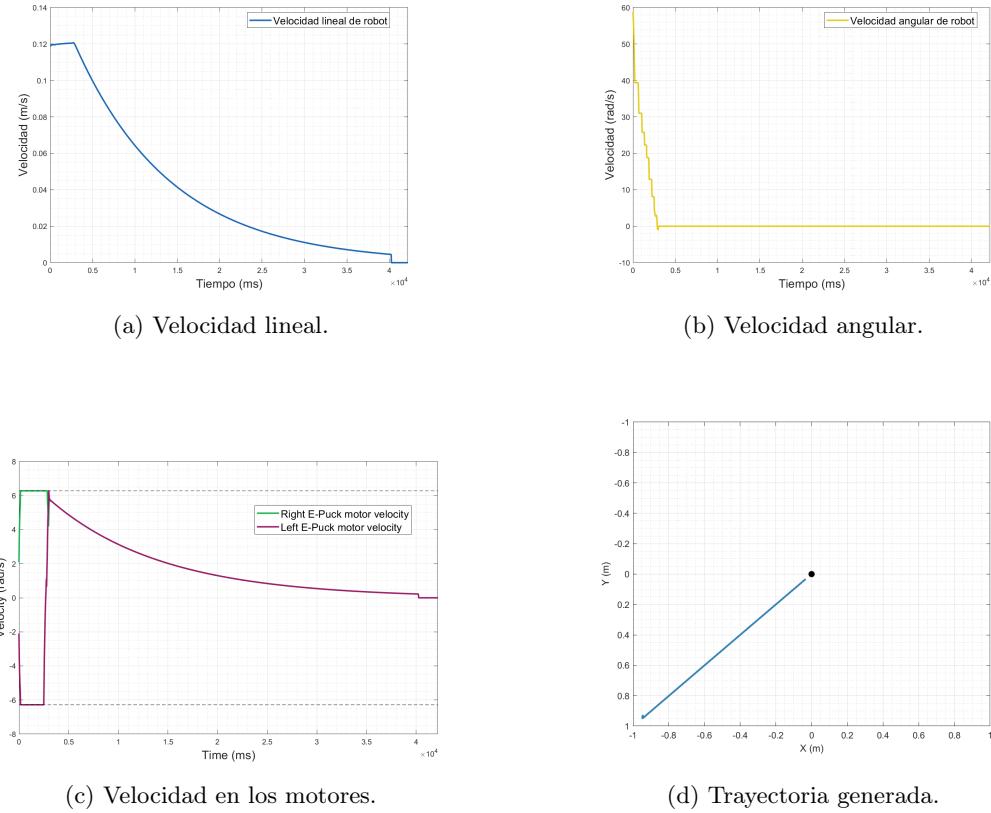


Figura 52: Gráficas de controlador de pose.

9.6. Control de pose de Lyapunov

Parámetro	Valor
K_ρ	0.09
K_α	25

Cuadro 26: Parámetros utilizados para el controlador de pose de Lyapunov.

Para la implementación de este controlador se utilizaron los valores mostrados en el Cuadro 26 en conjunto con las ecuaciones (20) y (21). La mayor diferencia entre este y el controlador anterior puede verse en la gráfica de velocidad lineal (Figura 53a). Este controlador de pose presenta una respuesta casi igual al controlador anterior, por lo que su trayectoria y sus velocidades son suaves (figuras 53b, 53c y 53d).

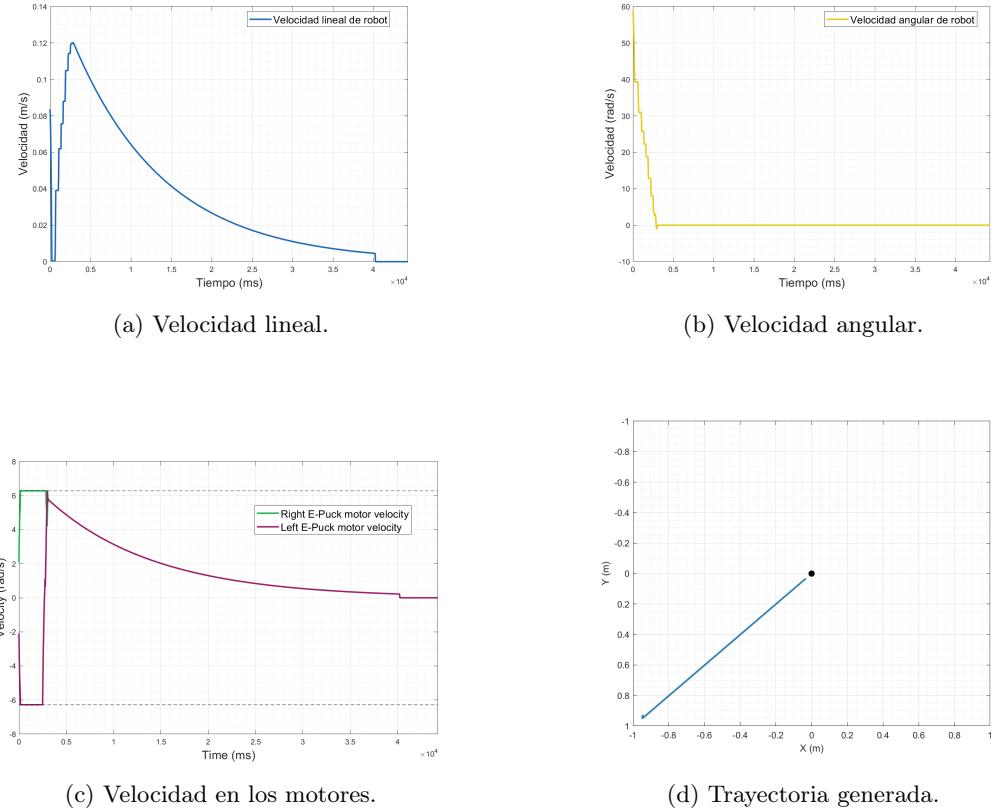


Figura 53: Gráficas de controlador de pose de Lyapunov.

9.7. Control Closed-Loop Steering

Parámetro	Valor
K_1	1
K_2	10

Cuadro 27: Parámetros utilizados para el controlador Closed Loop Steering.

Para este controlador se utilizaron las constantes indicadas en el Cuadro 27 con las ecuaciones (22-23). Aunque la velocidad lineal es suave (como se observa en la Figura 54a), la agresiva variación de la velocidad en las ruedas puede afectar el desempeño de los actuadores del robot. Las velocidades oscilan de una manera muy agresiva como puede observarse en las figuras 54b y 54c. El controlador *Closed-Loop Steering* genera una trayectoria suave (Figura 54d) a cambio de tener velocidades oscilantes.

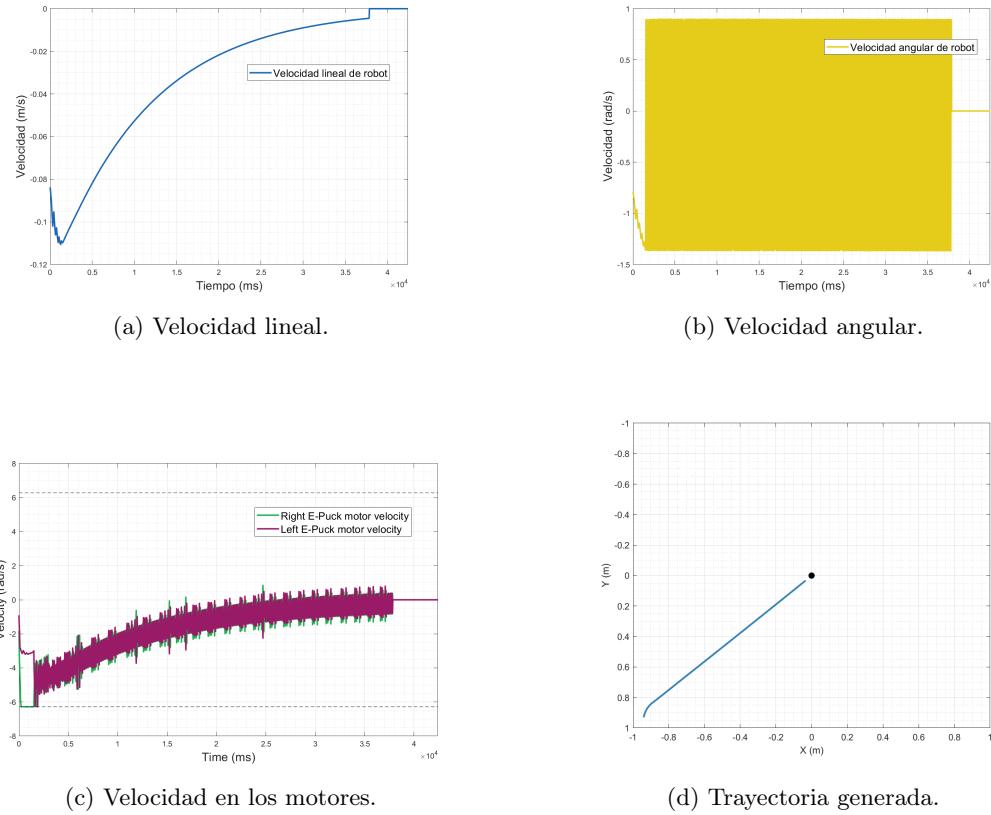


Figura 54: Gráficas de controlador Closed-Loop Steering.

9.8. Control LQR

Parámetro	Tamaño de matriz	Valor de matriz
A	2×2	$\mathbf{0}_{2 \times 2}$
B	2×2	$\mathbf{I}_{2 \times 2}$
Q	2×2	$0.1 \cdot \mathbf{I}_{2 \times 2}$
R	2×2	$\mathbf{I}_{2 \times 2}$
K_{LQR}	2×2	$0.6325 \cdot \mathbf{I}_{2 \times 2}$

Cuadro 28: Parámetros utilizados para el controlador LQR.

Las matrices utilizadas para calcular el LQR se presentan en el Cuadro 28. Puede observarse que sus velocidades no oscilan ni varían de forma muy drástica (figuras 55a y 55b). Este controlador presenta una respuesta similar al del controlador TUC, pero con la ventaja de que este se detiene en la meta de forma suave, como puede observarse en la Figura 55c. Su trayectoria no es tan recta, como puede observarse en 55d, pero es compensado con la suavidad de sus velocidades.

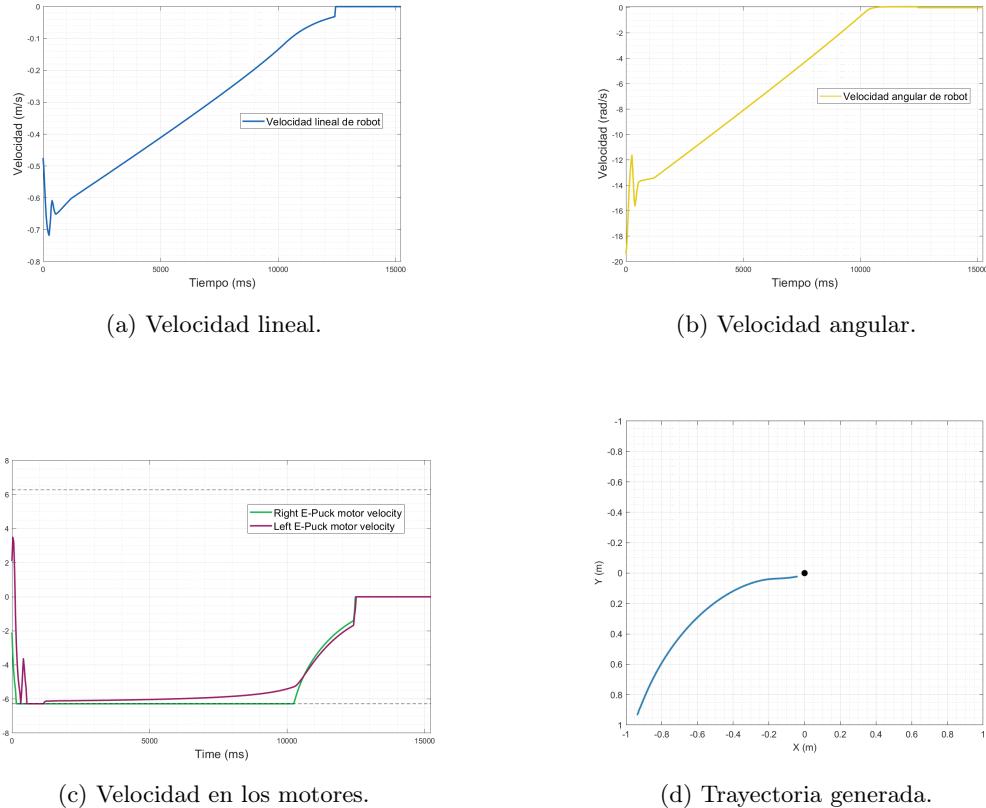


Figura 55: Gráficas de controlador LQR.

9.9. Control LQI

Las matrices utilizadas para calcular el LQI en Matlab son las presentadas en el Cuadro 29. Este controlador, al igual que el anterior, no presenta oscilaciones ni variaciones bruscas en las velocidades (figuras 56a, 56b y 56c). De hecho, este controlador también se detiene de manera suave al llegar a la meta. A diferencia del controlador de la sección anterior, este genera una trayectoria bastante recta, tal como puede observarse en la Figura 56d. Por tanto, no es sorpresa que este controlador haya sido uno de los mejores presentados en [1].

Parámetro	Tamaño de Matriz	Valor de Matriz
Q	4×4	$I_{4 \times 4}$
R	2×2	$2000 \cdot I_{2 \times 2}$
K_{LQR}	2×2	$0.2127 \cdot I_{2 \times 2}$
K_I	2×2	$0.0224 \cdot I_{2 \times 2}$

Cuadro 29: Parámetros utilizados para el controlador LQI.

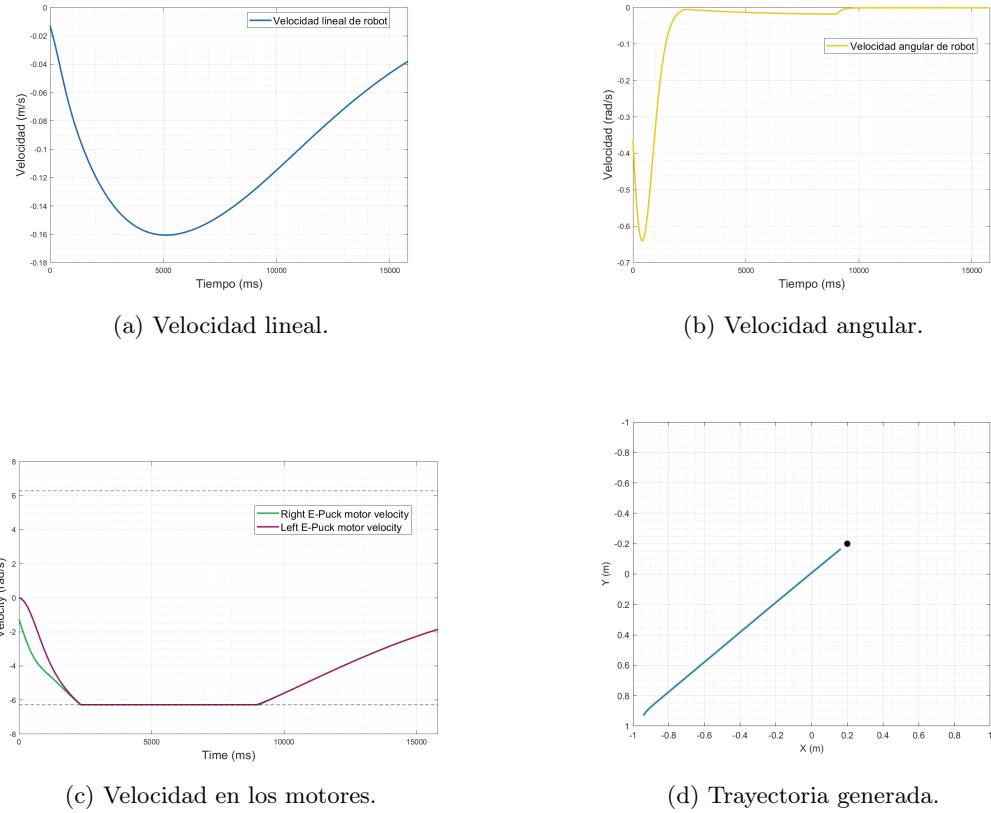


Figura 56: Gráficas de controlador LQI.

CAPÍTULO 10

Ant System controlado

En esta sección se modificó el archivo del algoritmo AS para que exportara el camino encontrado por las hormigas a la carpeta donde se encuentra el controlador de Webots. A este camino se le tuvo que modificar las coordenadas (Y en Matlab o Z en Webots) debido a que los ejes de la mesa de pruebas o marco inercial de Webots se encuentra reflejado. Además, es necesario ajustar la cuadrícula de 10×10 unidades a la mesa de pruebas de 2×2 metros. Por lo tanto el ajuste que se realizó fue la siguiente operación entre vectores en Matlab:

```
1 bpath = [G.Nodes.X(best_path), G.Nodes.Y(best_path)];  
2 webots_path = (bpath - grid_size/2).*[1/5 -1/5];
```

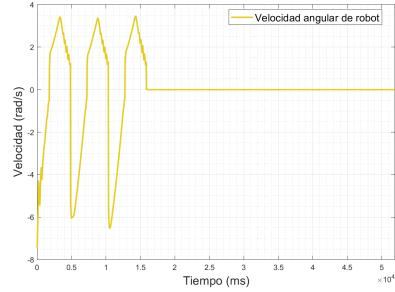
Donde se guardan las coordenadas del mejor camino encontrado para trasladarlo y hacerle el ajuste para que cada separación de la cuadrícula sea de 0.2 m.

10.1. Controladores simples en mundo cuadriculado

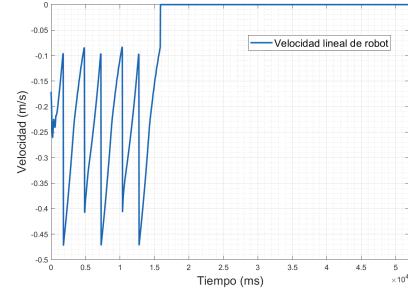
Para este experimento se utilizó el camino generado de la Figura 18b. Este camino pasa por 6 nodos, por lo que se tiene 6 metas distintas. El cambio de meta ocurre al llegar a un error de posición menor o igual a $\epsilon = 0.05$. En esta sección solo se utilizaron los controladores que presentaron mejores resultados en el capítulo 9: TUC, Pose, Pose de Lyapunov, LQR y LQI. En todos los casos se utilizaron los mismos parámetros que los presentados en el capítulo 9.

10.1.1. Control TUC

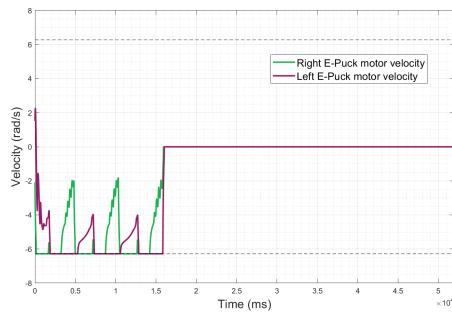
El control proporcional de velocidades con saturación limitada presentó una alta variación en la velocidad (figuras 57a y 57b). En la Figura 57c puede observarse que la velocidad se satura y cuenta con altos picos y cambios bruscos. Además, en la Figura 57d se observa cómo la trayectoria presenta varias oscilaciones en lugar de ir de forma recta hacia la meta. Puede notarse que aproximadamente en cada cambio de meta hay un pico en las velocidades.



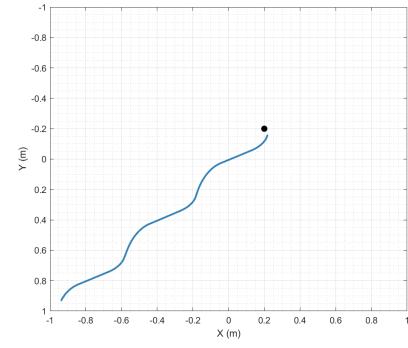
(a) Velocidad lineal.



(b) Velocidad angular.



(c) Velocidad en los motores.



(d) Trayectoria generada.

Figura 57: Gráficas de controlador TUC.

10.1.2. Control de pose

En el controlador de pose es más marcado el cambio de metas mencionado anteriormente. Puede observarse en la figuras 58a y 58c cómo el robot desacelera cuando se acerca a la meta y vuelve a acelerar cuando ocurre un cambio. Además solo al principio existe un pico de velocidad angular, cuando el robot corrige su orientación (Figura 58b). En contraste con el controlador anterior, este presenta una trayectoria recta como es posible apreciar en 58d.

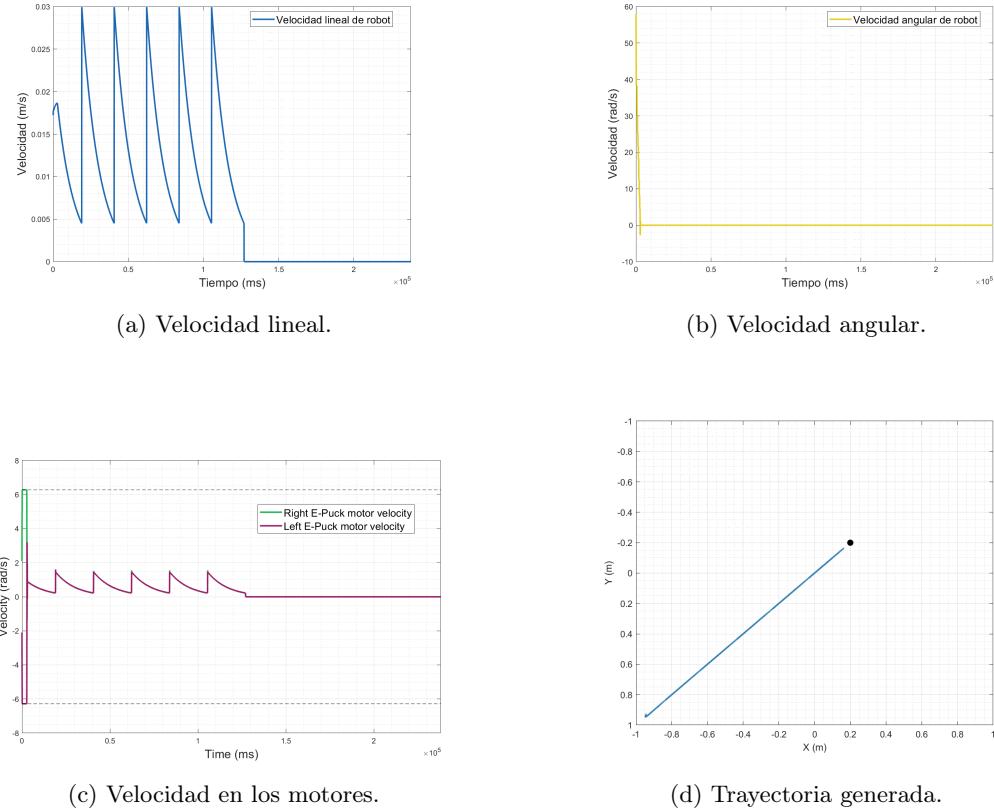


Figura 58: Gráficas de controlador de pose.

10.1.3. Control de pose de Lyapunov

Este controlador posee características similares a las del controlador de pose simple. También se aprecia la aceleración y desaceleración, y por supuesto, la trayectoria recta hacia la meta (Figura 59d). En las figuras 59a y 59c puede verse una respuesta similar en forma y tiempo que en las figuras 58a y 58c. La velocidad angular se activa solo al corregir la orientación al principio como se ve en la Figura 59b. Además la trayectoria es recta (Figura 59d), similar al controlador de pose simple.

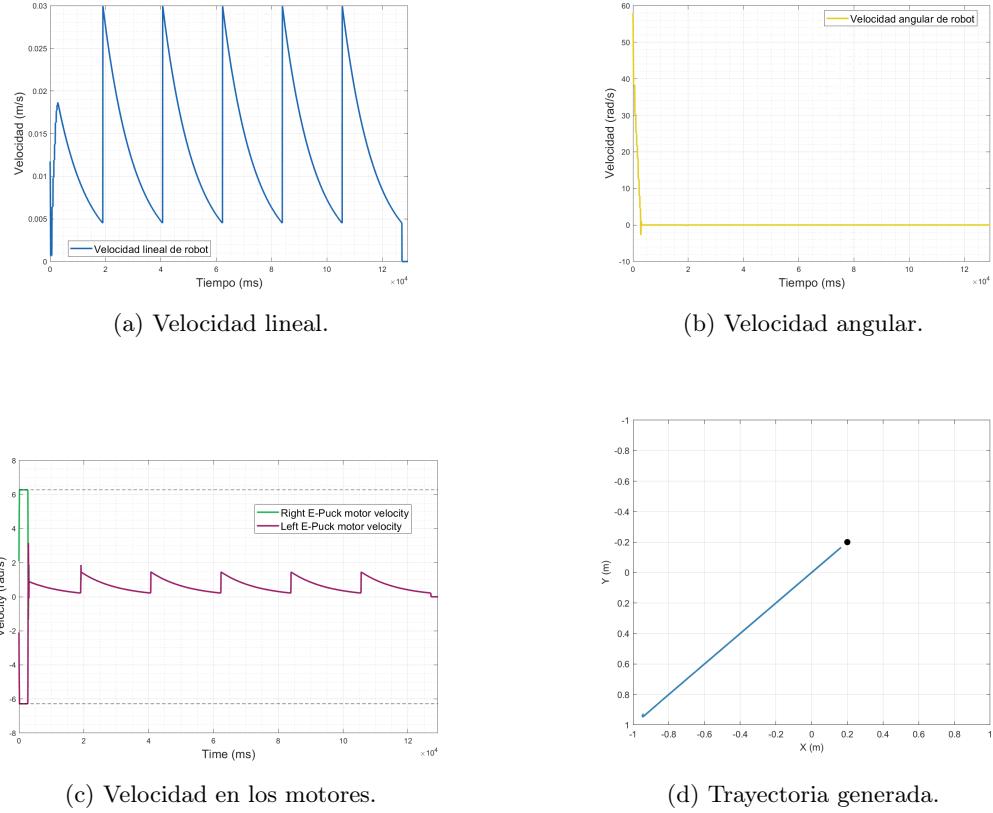


Figura 59: Gráficas de controlador de pose de Lyapunov.

10.1.4. Control LQR

Este controlador presenta una gráfica de velocidad de las llantas similar a las dos anteriores (figuras 60a y 60b), pero con saturación de la velocidad (Figura 60c), por lo que llega más rápido a la meta. Sin embargo, la trayectoria de este controlador no es tan recta como la de los controladores de pose como puede verse en la Figura 60d. A pesar de esto, la trayectoria es mejor que la generada con el controlador TUC.

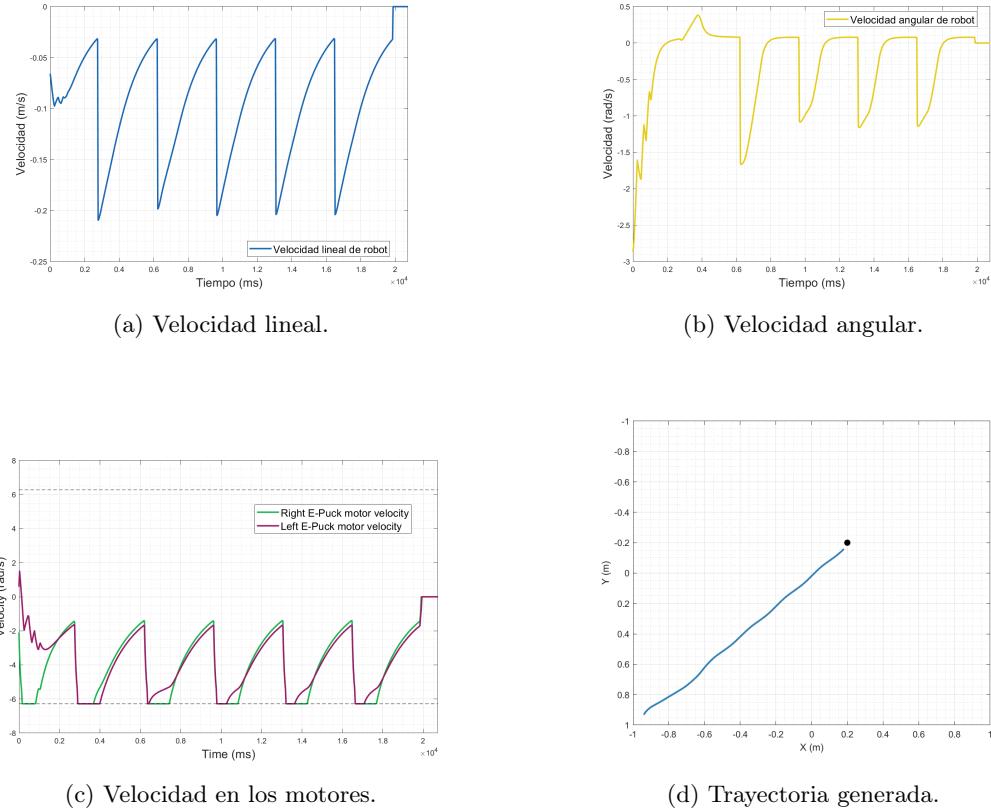
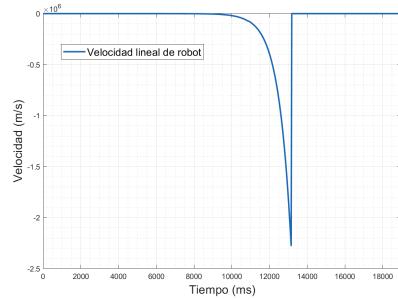


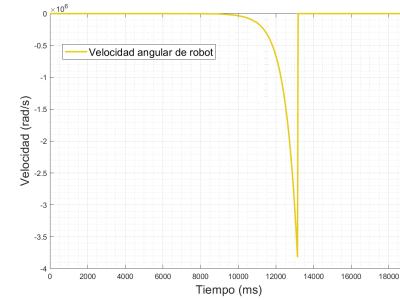
Figura 60: Gráficas de controlador LQR.

10.1.5. Control LQI

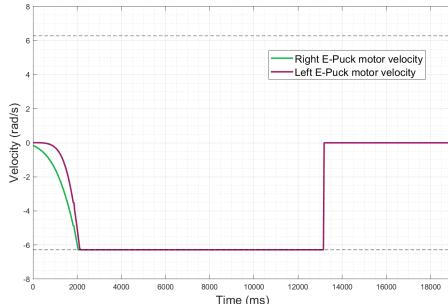
Este controlador es el más uniforme y rápido de todos. En las figuras 61a y 61b se observa cómo la velocidad queda estable y disminuye de forma suave al llegar a la meta. Además, como puede verse en la Figura 61c, el robot acelera suavemente hasta llegar a su velocidad máxima. Luego de esto, se mantiene con la misma velocidad hasta llegar a la meta. Aunque la trayectoria de este controlador es completamente recta, el problema yace en el error de posición. En la Figura 61d puede apreciarse que el robot se detiene aproximadamente a 0.1 metros en x de la meta. Si en la aplicación no se requiriera mayor precisión que esta, este controlador sería ideal por ser rápido y amigable con los motores del robot.



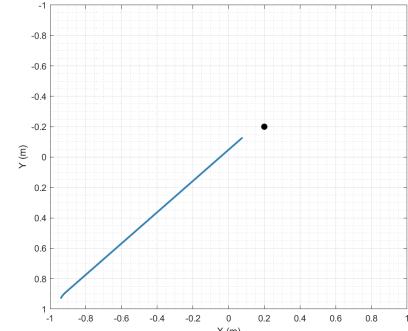
(a) Velocidad lineal.



(b) Velocidad angular.



(c) Velocidad en los motores.



(d) Trayectoria generada.

Figura 61: Gráficas de controlador LQI.

10.2. Controladores modificados en mundo cuadriculado

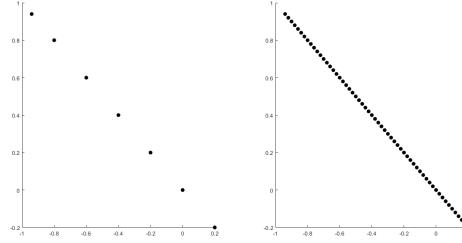


Figura 62: Camino interpolado y no interpolado para control de pose.

Para solucionar el hecho de que la velocidad de los motores tuviera cambios bruscos se planteó hacer una interpolación de los puntos como se muestra en la Figura 62. De esta manera, se tiene una mayor cantidad de puntos entre el nodo de inicio y el final, haciendo que el cambio de la velocidad sea menor. Luego de esto, se aplicó un filtro suavizador también llamado *leaky integrator* con $\lambda = 0.95$. Este filtro tiene función de transferencia (24) y ecuación de diferencias (25). De este modo es posible suavizar los pequeños picos en la velocidad generados por los cambios entre metas.

$$H(z) = \frac{1 - \lambda}{1 - \lambda z^{-1}} \quad (24)$$

$$y[n] = x[n - 1] + y[n - 1] \quad (25)$$

Para cada filtro se utilizó un distinto paso (`interpolate_step`) para interpolar y un distinto ϵ para realizar el cambio de meta. Además, para que la velocidad fuera más rápida se realizó un *offset* de la velocidad al sumarle o restarle cierta cantidad. Los parámetros utilizados por cada controlador se muestran en el Cuadro 30.

Controlador	Paso	ϵ	<i>offset</i>
Pose	0.02	0.01	+3.14
Pose de Lyapunov	0.02	0.01	+3.14
TUC	0.005	2.5×10^{-3}	$\times 2$
LQR	0.01	6×10^{-3}	-2
LQI	0.1	0.06	0

Cuadro 30: Parámetros utilizados en la modificación.

Además, en el siguiente código se muestra el proceso de interpolación utilizado en Matlab. El vector `pos` es de dimensión 1×3 y guarda la posición de la brújula de Webots.

Algoritmo 10.1: Interpolación de AS.

```

1 x = [ pos(1); webots_path(:, 1) ];
2 y = [ pos(3); webots_path(:, 2) ];
3 xi = (x(1):interpolate_step:x(end))';
4 yi = interp1q(x, y, xi);
5 goals = [ xi(2:end), yi(2:end) ];

```

10.2.1. Control TUC

En la Figura 63a puede verse que el MPSO forma una curva hasta llegar a la meta, mientras que el AS llega de forma recta. El control proporcional de velocidades con saturación limitada sin modificaciones presentó una alta variación en la velocidad. En la Figura 57c puede observarse que la velocidad se satura y cuenta con altos picos y cambios bruscos. En cambio, en la Figura 64b se muestra un pequeño cambio al principio y luego la velocidad se queda casi constante hasta el final. Además, en la Figura 57d la trayectoria presenta varias oscilaciones en lugar de ir de forma recta hacia la meta como en la Figura 63b.

También en la Figura 64a el MPSO combina saturación de motores con cambios bruscos de velocidad, pudiendo causar daños en los motores del robot. Sin embargo, el MPSO llegará antes a la meta que el AS, por lo que si se requiriera de una rápida llegada de varios robots y no solo uno se recomendaría el uso de MPSO.

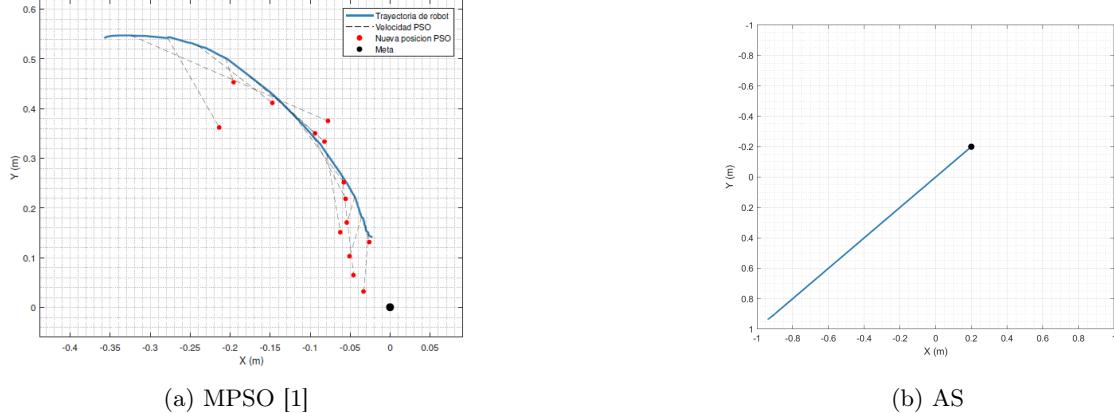
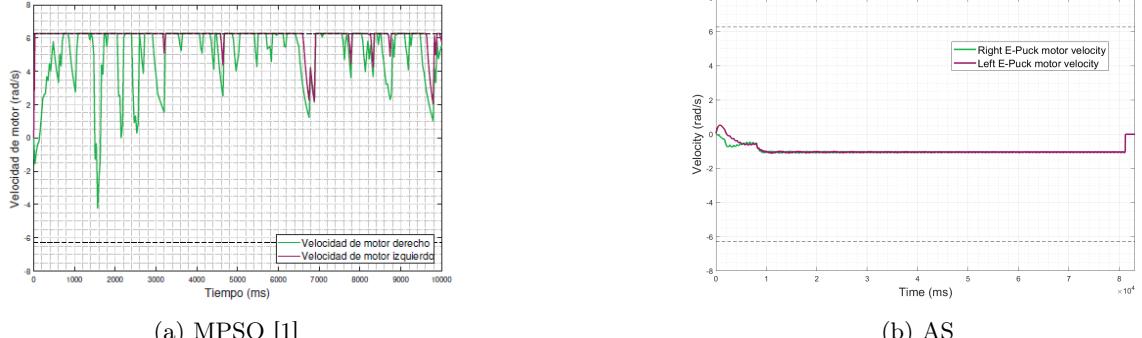


Figura 63: Trayectoria generada con el controlador TUC.



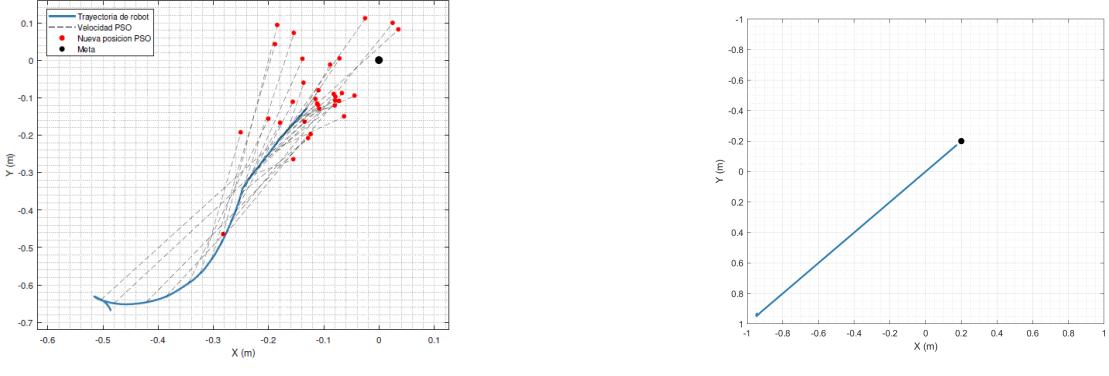
(a) MPSO [1]

(b) AS

Figura 64: Gráfica de velocidad en los motores para controlador TUC.

10.2.2. Control de pose

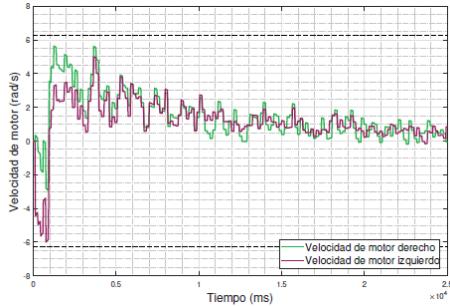
Una de las diferencias entre MPSO y AS es el camino tomado por el robot, pues el de AS es recto (Figura 65b) y el del MPSO presenta una curva (Figura 65a). La Figura 66a muestra cómo la velocidad baja conforme se acerca a la meta, pero con variaciones en la velocidad. Con la interpolación ya no es tan marcado el cambio de metas como en la Figura 58c. Puede verse en la Figura 66b cómo el robot tiene un cambio de velocidad en lo que gira para alinearse a la meta y luego queda casi de forma estable hasta descender de forma exponencial. A este controlador y al de pose de Lyapunov se les añadió otra modificación para que el paro al llegar a la meta fuera exponencial. A la entrada con el offset se le multiplicó un e^{-t} donde t es el un contador que comienza a aumentar en 0.0375 cuando el robot va por la penúltima meta. Esta constante se eligió luego de realizar pruebas de forma empírica, pues fue la que generó los mejores resultados.



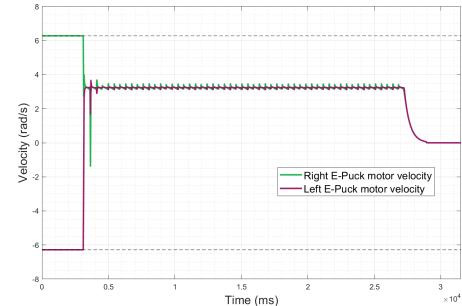
(a) MPSO [1]

(b) AS

Figura 65: Trayectoria generada con el controlador de pose.



(a) MPSO [1]

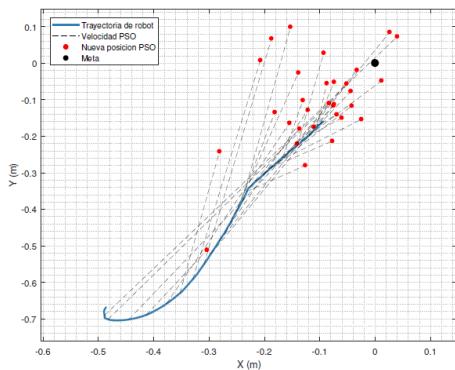


(b) AS

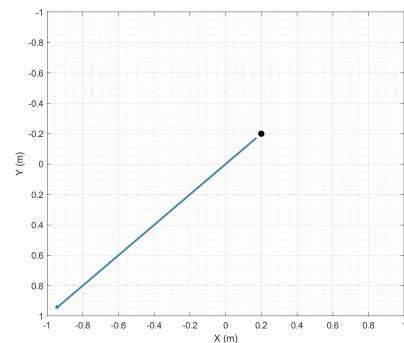
Figura 66: Gráfica de velocidad en los motores para controlador de pose.

10.2.3. Control de pose de Lyapunov

Como se mencionó en la sección 10.1.2, este controlador y el de Pose simple presentan respuestas muy parecidas. Esto puede evidenciarse tanto en el MPSO como con el AS en las figuras 65 - 68. Las trayectorias y la velocidad tienen formas muy parecidas. Incluso, es posible notar que los controladores convergen en un intervalo de tiempo similar.

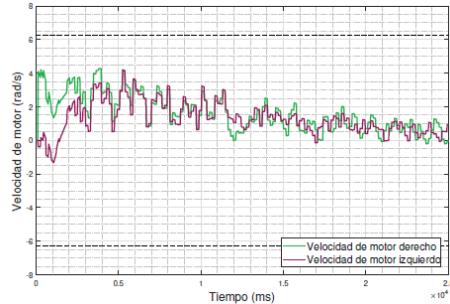


(a) MPSO [1]

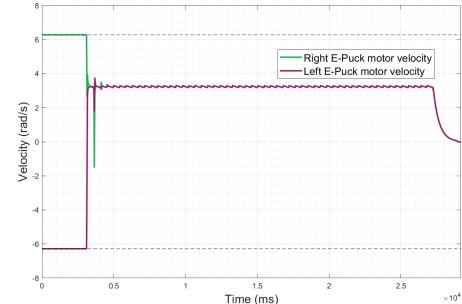


(b) AS

Figura 67: Trayectoria generada con el controlador de pose de Lyapunov.



(a) MPSO [1]



(b) AS

Figura 68: Gráfica de velocidad en los motores para controlador de pose de Lyapunov.

10.2.4. Control LQR

Este controlador con MPSO presenta una curvatura menor que las trayectorias anteriores (Figura 69a), pero su gráfica de velocidad es igual de fluctuante que las anteriores (Figura 70a). El controlador con AS acelera suavemente y permanece casi constante hasta el final (Figura 70b), produciendo un movimiento rectilíneo hasta la meta (Figura 69b).

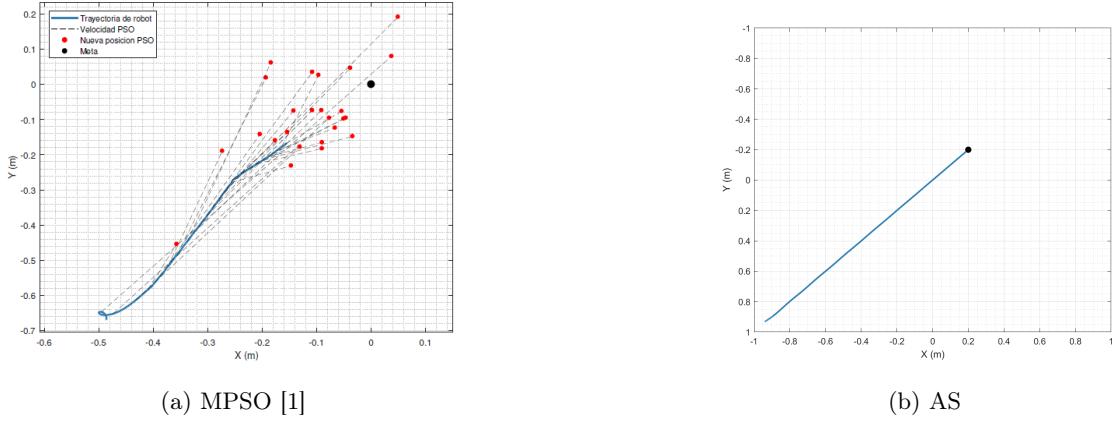


Figura 69: Trayectoria generada con el controlador LQR.

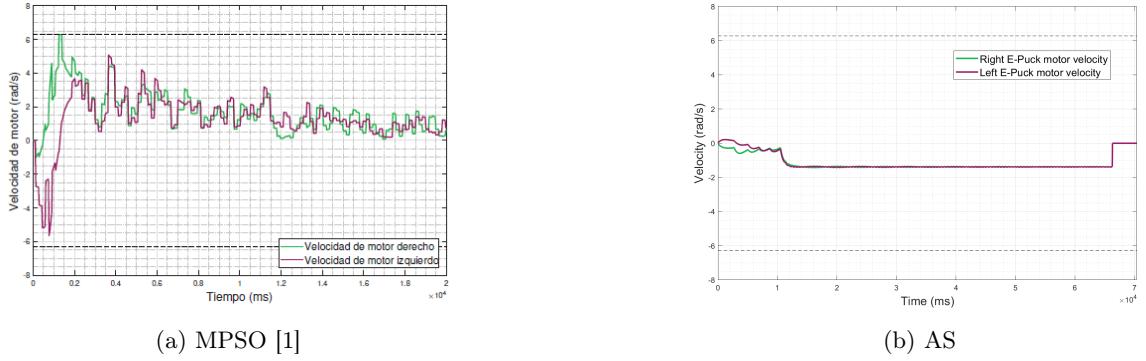


Figura 70: Gráfica de velocidad en los motores para controlador LQR.

10.2.5. Control LQI

El MPSO presenta una desaceleración suave y sin cambios bruscos en la velocidad de las ruedas, como se observa en la Figura 72a. En cambio, el AS muestra una aceleración suave hasta saturar los motores y se detiene de forma brusca al llegar a la meta (Figura 72b). En este caso el trato de las velocidades de los motores es mejor por parte del MPSO, pues no realiza un paro brusco. Sin embargo el movimiento hacia la meta del robot es un tanto curva (Figura 71a), pero no de forma tan pronunciada como con los primeros controladores. En contraste, la trayectoria con AS es completamente recta, como se observa en la Figura 71b.

Además, el ajuste de la interpolación ayudó al error de posición.

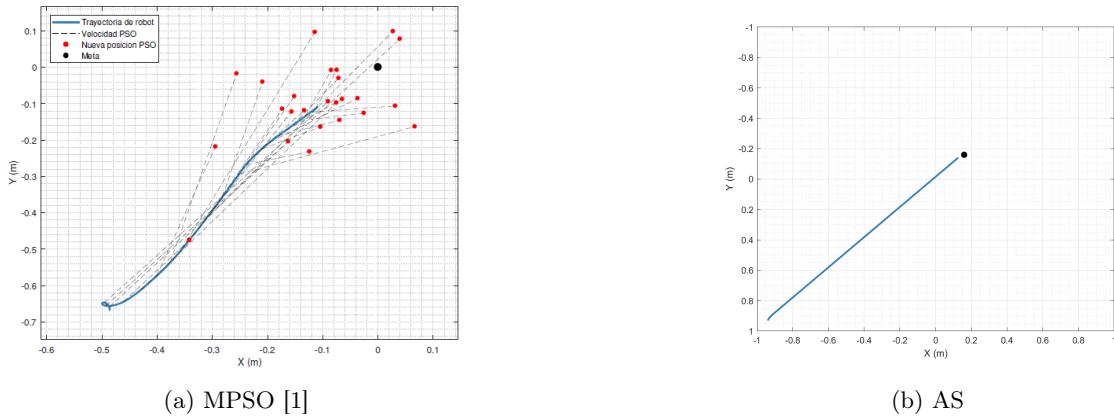


Figura 71: Trayectoria generada con el controlador LQI.

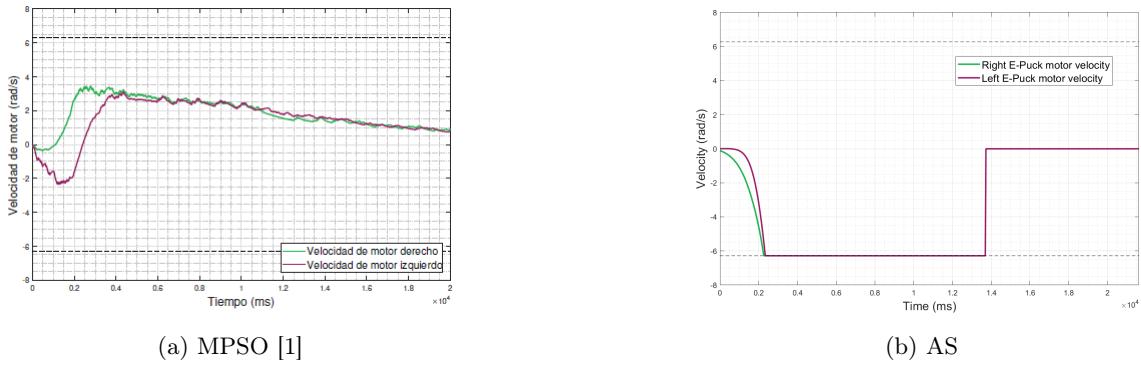


Figura 72: Gráfica de velocidad en los motores para controlador LQI.

CAPÍTULO 11

Algoritmos Genéticos (GA)

En este capítulo se muestran resultados preliminares de la implementación de algoritmos genéticos para planificar trayectorias. Lo que se busca es incursionar en el área, para conocer las posibilidades que se tienen y así poder explotarlas en el futuro. Por tanto, en esta sección no se encontrarán muchos resultados ni tan robustos como en las secciones de *Ant System*.

11.1. Minimizando funciones de costo

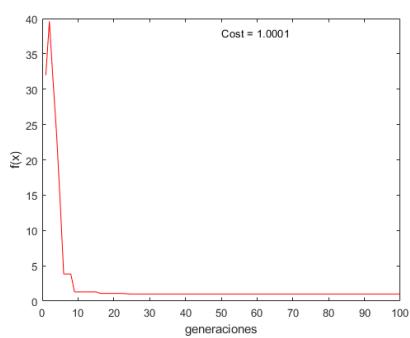
Inicialmente se consideró que el algoritmo genético podía ser utilizado como un algoritmo que encuentra caminos como el PSO al minimizar la función de costo. Por lo tanto, se propuso minimizar 4 funciones comunes de prueba descritas en el marco teórico: Rosenbrock, Ackley, Rastrigin y Booth. Para cada ejecución del programa se utilizó codificación binaria y se utilizaron 100 individuos, probabilidad de cruce de 90 %, probabilidad de mutación de 1 %, un máximo de 100 generaciones y una tolerancia de 5 % para considerar convergencia del algoritmo. El largo de los individuos es de 20 bits, para tener 10 bits para x y otros 10 bits para y. Asimismo, se desea tener 5 bits para el número y 5 bits para representar a sus decimales. En el caso de la función de Ackley es necesario aumentar los bits porque al utilizar representación binaria con 5 bits solo es posible llegar a $2^5 - 1 = 31$, pero el rango sugerido por [14] es de 32.768 positivo y negativo. En el Cuadro 31 puede observarse los rangos utilizados y la longitud de los individuos.

Función	Rango x	Rango y	Lind
Rosenbrock	[-5, 10]	[-5, 10]	20
Ackley	[-32.768, 32.768]	[-32.768, 32.768]	24
Rastrigin	[-5.12, 5.12]	[-5.12, 5.12]	20
Booth	[-10, 10]	[-10, 10]	20

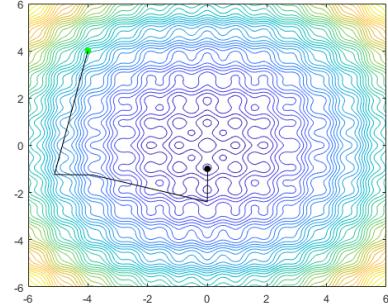
Cuadro 31: Parámetros que varían según la función de costo.

11.2. Resultados

Como se puede observar en las siguientes figuras, los caminos generados por el algoritmo genético no son para nada suaves y óptimos. Los mejores resultados pueden verse con la función de Rastrigin (Figura 73) y Ackley (Figura 75), pero aún así los resultados no son buenos a comparación con los del AS. Es posible apreciar que las trayectorias generadas con la función de Rosenbrock (Figura 74) y Booth (Figura 76) son casi aleatorias, con varios cruces extraños. Por tanto no es recomendable utilizar estas trayectorias sin un procesamiento extra, pues podría dañar los actuadores del robot.

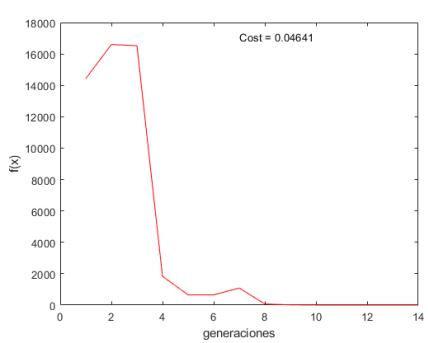


(a) Costo de la función de Rastrigin.

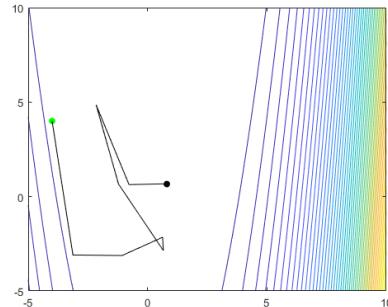


(b) Animación de algoritmo genético con la función de costo de Rastrigin.

Figura 73: Minimización de la función de Rastrigin.

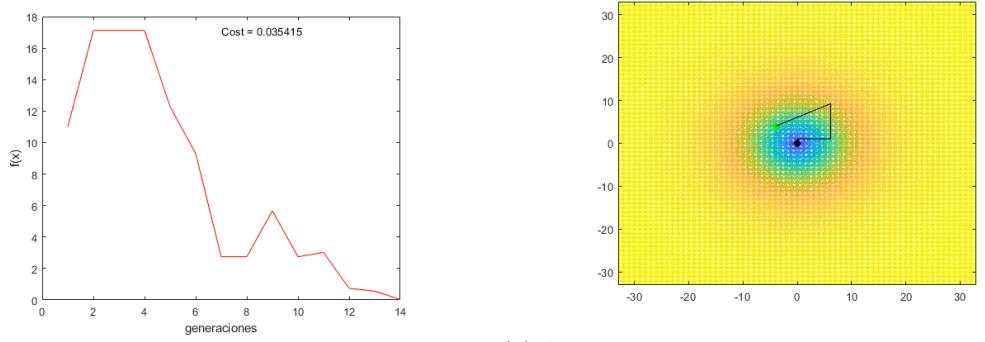


(a) Costo de la función de Rosenbrock.

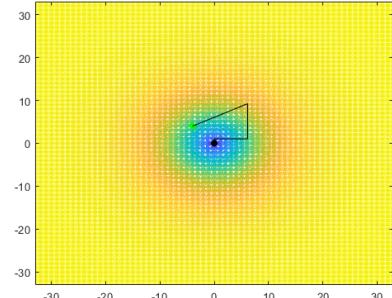


(b) Animación de algoritmo genético con la función de costo de Rosenbrock.

Figura 74: Minimización de la función de Rosenbrock.

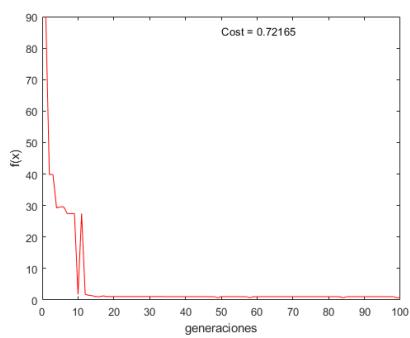


(a) Costo de la función de Ackley.

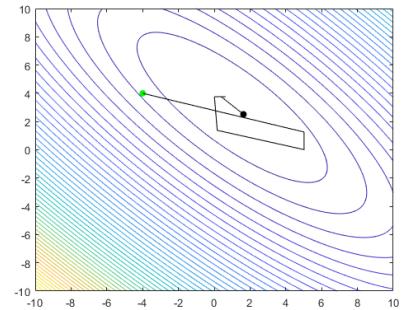


(b) Animación de algoritmo genético con la función de costo de Ackley.

Figura 75: Minimización de la función de Ackley.



(a) Costo de la función de Booth.



(b) Animación de algoritmo genético con la función de costo de Booth.

Figura 76: Minimización de la función de Booth.

Ya que estos son resultados preliminares, no es alarmante lo descubierto. De hecho, es enriquecedor pues es posible tomar en cuenta que es necesario hacer cambios en el algoritmo para que funcione como el AS o el MPSO. Posiblemente podría restringirse la variación entre los cromosomas para que el movimiento sea suave, o incluso utilizar una codificación diferente.

CAPÍTULO 12

Conclusiones

- Es posible verificar el funcionamiento del algoritmo AS, debido a que las simulaciones demuestran una correcta deposición de feromonas. De este modo las hormigas encuentran un camino entre el nodo inicial y final.
- El robot con los controladores desarrollados en la fase anterior del proyecto llega satisfactoriamente a la meta. Por tanto, los modelos de movimiento de los Bitbots desarrollados en la fase anterior del proyecto funcionan tanto en AS como en MPSO.
- Una buena opción de controlador para el algoritmo AS es el LQI si no se requiere de una precisión mayor a aproximadamente 0.1 m. De esta manera no es necesario modificar el camino encontrado por el AS en un mundo cuadriculado sin obstáculos.
- Si se desea controlar enjambres de robots de forma simultánea y *online* es mejor utilizar el MPSO sobre el AS. El algoritmo AS haría que los robots recorrieran el lugar varias veces hasta hallar la meta. En cambio, el MPSO puede llevar a varios directamente al lugar por su naturaleza vectorial.
- Dos buenas opciones de controlador con AS, para el camino recto específico que se probó, son el controlador de Pose y el controlador de Pose de Lyapunov. Ambos controladores presentan una velocidad suave y casi constante en la mayor parte del tiempo, protegiendo así los actuadores. Sin embargo, como un caso general, no podría concluirse.
- En contraste con el algoritmo sin paralelización, este algoritmo es en promedio 2.6 veces más rápido utilizando 4 núcleos.
- La paralelización mejora el desempeño del algoritmo AS, pero tiene un límite. Esto quiere decir que no todo el algoritmo puede ser paralelizado.
- Utilizar algoritmos genéticos directamente para planificar trayectorias no genera buenos resultados. Las trayectorias generadas no son suaves y en algunos casos no son lógicas.

- Los métodos de planificación de trayectorias con grafos generan trayectorias más rectas, pero son computacionalmente más demandantes. En cambio, los métodos sin grafos son más rápidos, pero no generan trayectorias tan rectas.
- La interpolación y el filtro suavizador funcionan para trayectorias rectas. La interpolación devuelve puntos de una función, por lo que la trayectoria también debe de cumplir los requisitos de una función matemática.

CAPÍTULO 13

Recomendaciones

- En [7] se plantea la posibilidad de utilizar el algoritmo Ant Colony para optimizar el *Floorplanning* en aplicaciones de VLSI. Por tanto, en el futuro sería interesante abrir una línea de investigación de algoritmos de inteligencia artificial aplicada a VLSI.
- Ya que solo se probó en mundos sin obstáculos, sería recomendable que en la siguiente fase se incorporen obstáculos para visualizar la evolución del algoritmo.
- También podría compararse este tipo de algoritmos de inteligencia computacional contra los métodos tradicionales como A*, Dijkstra, Breathfire.
- Además del SACO y AS, existen otras variantes como el Ant max-min y el AntQ. El último incluye una modificación parecida a la usada en Q Learning.
- Podría intentarse la mezcla entre entre algoritmos genéticos y PSO o algoritmos genéticos y ACO.
- Ya que el programa en Matlab requería de una computadora de alto rendimiento, se podría transcribir a Python, C/C++ u otro (junto con la paralelización), para comprobar si se puede hacer que el algoritmo se ejecute más rápido.
- Sería recomendable probar los algoritmos en el Robotarium del Tecnológico de Georgia como alternativa a la simulación, pues ellos cuentan con robots disponibles para todo el mundo.
- Podría aplicarse *Q Learning*, *Deep Learning* u otros métodos inteligentes para ajustar los parámetros de la mejor manera sin necesidad de una computadora de alto rendimiento.
- En [17] comparan el desempeño del TSP en utilizando distintos tipos de cruza y mutación en algoritmos genéticos, por lo que se podría hacer lo mismo para nuestro problema.
- Podría intentarse usar algoritmos genéticos combinados con APF.

- Probar el AS con otros escenarios (mayor cuadrícula y caminos no necesariamente rectos o diagonales). Esto para asegurar el funcionamiento del mismo desde varias perspectivas, y no solo una.
- Comparar el barrido de núcleos con el AS paralelizado contra otros algoritmos paralelizados (como Dijkstra, A*, etc.). De este modo será posible comparar si el comportamiento es el mismo o varía según el algoritmo elegido.

CAPÍTULO 14

Bibliografía

- [1] A. S. A. Nadalini, «Algoritmo Modificado de Optimización de Enjambre de Partículas (MPSO)», Universidad del Valle de Guatemala, UVG, nov. de 2019.
- [2] J. P. C. Pérez, «Implementación de enjambre de robots en operaciones de búsqueda y rescate», Universidad del Valle de Guatemala, UVG, nov. de 2019.
- [3] *Robotarium / Institute for Robotics and Intelligent Machines*. dirección: <http://www.robotics.gatech.edu/robotarium>.
- [4] *Programmable Robot Swarms*, en-US, ago. de 2016. dirección: <https://wyss.harvard.edu/technology/programmable-robot-swarms/> (visitado 06-04-2020).
- [5] S. P. ROUL, «Application of Ant Colony Optimization for finding Navigational Path of mobile robot», National Institute of Technology, Rourkela, 2011.
- [6] *Comparative Analysis of Ant Colony and Particle Swarm Optimization Techniques* V.Selvi Lecturer, Department of Computer Science, Nehru Memorial College,
- [7] A. P. Engelbrecht, *Computational intelligence: an introduction*. Wiley, 2008.
- [8] M. N. A. Wahab, S. Nefti-Meziani y A. Atyabi, «A Comprehensive Review of Swarm Optimization Algorithms», *PLoS ONE*, vol. 10, n.º 5, 2015. DOI: <https://doi.org/10.1371/journal.pone.0122827>.
- [9] Y. Zhang, S. Wang y G. Ji, «A Comprehensive Survey on Particle Swarm Optimization Algorithm and Its Applications», *Hindawi*, vol. 2015, n.º 931256, 2015. DOI: <http://dx.doi.org/10.1155/2015/931256>.
- [10] M. Dorigo y T. Stützle, *Ant colony optimization*, en. Cambridge, Mass: MIT Press, 2004, ISBN: 978-0-262-04219-2.
- [11] U. Sydney, *Genetic Algorithm:A Learning Experience*. dirección: http://www.cse.unsw.edu.au/~cs9417ml/GA2/encoding_other.html (visitado 27-07-2020).
- [12] D. Samanta, «Encoding Techniques in Genetic Algorithms», en, pág. 42,
- [13] M. Obitko, *Encoding - Introduction to Genetic Algorithms - Tutorial with Interactive Java Applets*, 1998. dirección: <https://www.obitko.com/tutorials/genetic-algorithms/encoding.php> (visitado 27-07-2020).

- [14] D. Bingham. (2017). Optimization Test Problems, dirección: <https://www.sfu.ca/~ssurjano/optimization.html>.
- [15] K. Rodríguez Vázquez, *Cómputo evolutivo - Inicio*, es, Library Catalog: www.coursera.org. dirección: <https://www.coursera.org/learn/computo-evolutivo/home/welcome> (visitado 27-07-2020).
- [16] H. Mühlenbein y D. Schlierkamp-Voosen, «Predictive Models for the Breeder Genetic Algorithm I. Continuous Parameter Optimization», en, *Evolutionary Computation*, vol. 1, n.º 1, págs. 25-49, mar. de 1993, ISSN: 1063-6560, 1530-9304. DOI: 10.1162/evco.1993.1.1.25. dirección: <http://www.mitpressjournals.org/doi/10.1162/evco.1993.1.1.25> (visitado 27-07-2020).
- [17] L. P., K. C.M.H., M. R.H., I. I. y D. S., «Genetic Algorithms for the Travelling Salesman Problem: A Review of Representations and Operators», en, pág. 42,
- [18] *Genetic Algorithms - Mutation - Tutorialspoint*. dirección: https://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_mutation.htm (visitado 08-08-2020).
- [19] C. A. Maeso, *Métodos basados en grafos*. dirección: http://pdg.cnb.uam.es/pazos/cursos/bionet_UAM/Grafos_CAguirre.pdf.
- [20] K. Thulasiraman y M. N. S. Swamy, *Graphs: Theory and Algorithms*, en. John Wiley & Sons, mar. de 2011, Google-Books-ID: rFH7eQffQNkC, ISBN: 978-1-118-03025-7.
- [21] K. " Thulasiraman, S. Arumugam, A. Brandstädt y T. Nishizeki, eds., *Handbook of Graph Theory, Combinatorial Optimization, and Algorithms*, en, 0.^a ed. Chapman y Hall/CRC, ene. de 2016, ISBN: 978-0-429-15023-4. DOI: 10.1201/b19163. dirección: <https://www.taylorfrancis.com/books/9781420011074> (visitado 12-07-2020).
- [22] *Graph with undirected edges - MATLAB - MathWorks América Latina*. dirección: <https://la.mathworks.com/help/matlab/ref/graph.html> (visitado 12-07-2020).
- [23] D. Phillips, *Python 3 object oriented programming: harness the power of Python 3 objects*, en, ép. Community experience distilled. Birmingham: Packt Publ, 2010, OCLC: 802342008, ISBN: 978-1-84951-126-1.
- [24] *Clases - MATLAB & Simulink - MathWorks América Latina*. dirección: <https://la.mathworks.com/help/matlab/object-oriented-programming.html> (visitado 12-07-2020).
- [25] T. Mathieu, G. Hernandez y A. Gupta, «Parallel Computing with MATLAB», en, pág. 56,
- [26] *Using parfor Loops: Getting Up and Running*, Library Catalog: blogs.mathworks.com. dirección: <https://blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/> (visitado 13-07-2020).
- [27] *Programming Patterns: Maximizing Code Performance by Optimizing Memory Access*, en, Library Catalog: la.mathworks.com. dirección: <https://la.mathworks.com/company/newsletters/articles/programming-patterns-maximizing-code-performance-by-optimizing-memory-access.html> (visitado 13-07-2020).
- [28] *Techniques to Improve Performance - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/matlab_prog/techniques-for-improving-performance.html (visitado 13-07-2020).

- [29] *Vectorization - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/matlab_prog/vectorization.html (visitado 13-07-2020).
- [30] *Buscar cuellos de botella de código - MATLAB & Simulink - MathWorks América Latina*. dirección: https://la.mathworks.com/help/matlab/creating_plots/assessing-performance.html (visitado 13-07-2020).
- [31] K. M. Lynch y F. C. Park, *Modern robotics: mechanics, planning, and control*, en. Cambridge, UK: Cambridge University Press, 2017, OCLC: ocn983881868, ISBN: 978-1-107-15630-2.
- [32] M. Zea, *MT3005 Lección 11 - modelado de robots móviles*. dirección: <https://www.youtube.com/watch?v=5AqXC8ivZ8U&list=PLWIV1mZv4WUbgPWFhWgEucu7tTKollf8K&index=2> (visitado 28-08-2020).
- [33] P. Corke, *Robotics Toolbox*, abr. de 2020. dirección: <https://petercorke.com/toolboxes/robotics-toolbox/>.
- [34] *Procesador Intel® Xeon® Oro 6152 (caché de 30.25 M, 2.10 GHz) Especificaciones de productos*, es. dirección: <https://ark.intel.com/content/www/es/es/ark/products/120491/intel-xeon-gold-6152-processor-30-25m-cache-2-10-ghz.html> (visitado 13-08-2020).

CAPÍTULO 15

Anexos

15.1. Especificaciones de computadora de alto desempeño



Figura 77: Procesador Intel Xeon Gold [34]

La computadora es una DELL Precision Tower 3620 con 2 procesadores Intel Xeon Gold 6152 de 2.1 Ghz y 3.7 Ghz Turbo, 3 Ultra Path Interconnect, 30MB de Cache y 140W de energía promedio que el procesador disipa al operar en la frecuencia básica con todas los núcleos activos. La tecnología de este procesador es de 14 nm. Cada procesador tiene 22 núcleos, por lo que en total se cuenta con 44 núcleos para procesamiento. Además, esta computadora cuenta con 384GB de memoria DDR4 con una frecuencia de 2666MHz, un disco duro de M.2 2TB de estado sólido y 3 discos de 12 TB 7200rpm SATA Enterprise Hard Drive [34]. Matlab fue instalado en una partición de Windows, pues su sistema operativo principal es CentOS 7.

CAPÍTULO 16

Glosario

AS: Algoritmo de inteligencia de enjambre basado en hormigas; Ant System. 1

estocástico: Aleatorio o al azar. 15

grafo de visibilidad: Grafo que toma las esquinas de sus obstáculos como nodos y une entre sí los que pueden verse (sin obstáculo de por medio) para conformar sus aristas. 20

inteligencia de enjambre: Rama de la inteligencia artificial que estudia el comportamiento de los seres que al estar en grupo realizan tareas extraordinarias. 7

metaheurística: Método inteligente y funcional, pero no necesariamente formal. 9

MPSO: Optimización de enjambre de partículas modificado o Modified Particle Swarm Optimization. Algoritmo PSO modificado en la fase II del proyecto robotat por [1]. 1

PRM: Mapas probabilísticos o Probabilistic Road Maps. 21

PSO: Optimización de enjambre de partículas o Particle Swarm Optimization. Algoritmo de inteligencia de enjambre basado en peces y aves. 1

RRT: Rapidly Exploring Random Trees. 21

Toolbox: Paquete extra de Matlab con funciones, estructuras y clases definidas. 18