Universidad de Costa Rica
Facultad de Ingeniería
Escuela de Ingeniería Eléctrica

IE0499 – Proyecto Eléctrico

Implementación y simulación de algoritmos de evasión de obstáculos y seguimiento de trayectorias para un robot autónomo terrestre

por

Daniel Díaz Molina

Ciudad Universitaria Rodrigo Facio

Agosto de 2017

Implementación y simulación de algoritmos de evasión de obstáculos y seguimiento de trayectorias para un robot autónomo terrestre

por

Daniel Díaz Molina

B22245

IE0499 – Proyecto Eléctrico

Aprobado por

Ing. Leonardo Marín Paniagua, Ph. D.
Profesor guía

Ing. José David Rojas Fernández, Ph. D.
Profesor lector

Ricardo Román Brenes, M. Sc.
Profesor lector

Agosto de 2017

Resumen

Implementación y simulación de algoritmos de evasión de obstáculos y seguimiento de trayectorias para un robot autónomo terrestre

por

Daniel Díaz Molina

Universidad de Costa Rica
Escuela de Ingeniería Eléctrica
Profesor guía: Ing. Leonardo Marín Paniagua, Ph. D.
Agosto de 2017

Este trabajo detalla la implementación de varios algoritmos de evasión de obstáculos para robots móviles.

Se incluye una investigación bibliográfica de la teoría relevante al trabajo, en donde se menciona la clasificación y modelado de robots móviles, y el efecto de la configuración de las ruedas en la cinemática del robot. Se explica de manera general el funcionamiento de algunos sensores que se pueden usar para la detección de obstáculos. Además se incluye una descripción teórica de 3 algoritmos de evasión: vehículos de Braitenberg, *Vector Field Histogram* (VFH) y VFH+.

El objetivo principal del proyecto fue la implementación de los algoritmos, misma que se hizo en el lenguaje de programación Python. Se hizo una implementación funcional de un algoritmo basado en los vehículos de Braitenberg, el algoritmo VFH original y el VFH+. El código completo se incluye como un anexo al trabajo.

Los algoritmos luego fueron probados mediante el uso de una herramienta de simulación: V-REP. La comunicación con V-REP mediante la API Remota, que provee V-REP para el control de la simulación con aplicaciones externas, fue una parte considerable del trabajo desarrollado. El código y las escenas de simulación también se incluyen como parte de los resultados.

Como resultado principal se incluyen las trayectorias seguidas por el robot usando los algoritmos implementados, para tres cursos de obstáculos diferentes, además de una tabla comparativa que resume el desempeño de cada algoritmo en cada pista. El algoritmo VFH+ fue el que mostró el mejor desempeño.

***Palabras claves**: Robótica, robots autónomos, evasión de obstáculos, planeamiento local, V-REP, Python, AI, Inteligencia artificial, Vehículo de Braitenberg, VFH, VFH+.*

Acerca de IE0499 – Proyecto Eléctrico

El Proyecto Eléctrico es un curso semestral bajo la modalidad de trabajo individual supervisado, con el propósito de aplicar estrategias de diseño y análisis a un problema de temática abierta de la ingeniería eléctrica. Es un requisito de graduación para el grado de bachiller en Ingeniería Eléctrica de la Universidad de Costa Rica.

Abstract

Implementación y simulación de algoritmos de evasión de obstáculos y seguimiento de trayectorias para un robot autónomo terrestre

Original in Spanish. Translated as: "Implementation and Simulation of Obstacle Avoidance Algorithms with Trajectory Tracking for Autonomous Indoor Vehicles"

by

Daniel Díaz Molina

University of Costa Rica
Department of Electrical Engineering
Tutor: Ing. Leonardo Marín Paniagua, Ph. D.
August of 2017

This report details the implementation of several obstacle avoidance algorithms for mobile robots.

A bibliographical research was conducted in relevant areas of robotics, covering the topics of robot classification and modeling, as well as the effect of wheel configuration on the robot's kinematics. A general explanation of several sensors that could be used for obstacle detection is included. Last, a theoretical description of 3 algorithms is presented: one based on Braitenberg's vehicles, the *Vector Field Histogram* (VFH), and VFH+.

The primary focus of the project was to implement the algorithms, this implementation was done in the Python Programming Language. Functional implementations of an algorithm based on Braitenberg vehicles, the original VFH algorithm, and the VFH+ algorithm were developed. The complete source code is available as an appendix.

The algorithms were put to test on a simulation, for this the software V-REP was used. Communication between the implemented code and V-REP was done using the Remote API provided by V-REP for control of the simulation by external applications. This was a big chunk of the work done, all code and files used are included as part of the results.

The main results presented in this report are the trajectories followed by the robot while using the algorithms on three different obstacle courses, as well as summary table that compares all three algorithms on all three courses. VFH+ achieved a better overall performance.

***Keywords**: Robotics, Autonomous robots, Obstacle Avoidance, Local Path Planning, V-REP, Python, AI, Artificial Intelligence, Braitenberg Vehicle, VFH, VFH+.*

About IE0499 – Proyecto Eléctrico ("Electrical Project")

The "Electrical Project" is a course of supervised individual work of one semester, with the purpose of applying design and analysis strategies to a problem in an open topic in electrical engineering. It is a requisite of graduation for the Bachelor of Science in Electrical Engineering, granted by the University of Costa Rica.

Dedicado a mi mamá y a mis amigos

Agradecimientos

Quiero agradecer de todo corazón a mi mamá porque gracias a ella es que estoy ahora a punto de graduarme. Su ayuda y consejo me han servido durante toda la carrera, en los buenos y malos momentos, en las decisiones importantes, en poder entender lo que quiero de mi carrera y en poder balancear mi vida y ser feliz. Espero que este trabajo demuestre tener la calidad y atención a los detalles que mi mamá siempre demostró y me inculcó.
También quiero agradecer a mis amigos y al DS, por ser una fuente constante de alegría en especial en los momentos de más estrés. Quiero agradecer especialmente a Ricardo Quirós y Michelle Cersósimo por su apoyo incondicional. Por último me gustaría agradecer a mi profesor tutor, por aguantarme todo el semestre y más, por exigirme excelencia y hacer de este proyecto algo de lo que me pueda sentir orgulloso de haber completado. Además quiero agradecer a todas las personas que me ayudaron con la corrección del trabajo, y a todos los compañeros de carrera que ayudaron de alguna forma u otra a llegar a este punto de mi carrera.

Índice general

Índice de figuras

Índice de tablas

Índice de algoritmos

Capítulo 1

Introducción

1.1. Motivación

La evasión de obstáculos en tiempo real es un aspecto clave para el éxito de aplicaciones de robótica móvil. Este es parte de un problema de más alto nivel, la navegación autónoma, que involucra simultáneamente dirigir al robot a un objetivo predeterminado y evadir obstáculos que se presenten en tiempo real, y de los cuales no necesariamente se tiene información de antemano. Así, hay un aspecto de planeamiento de rutas global para alcanzar el objetivo, y un planeamiento local para la evasión en tiempo real [**?**]. Este planeamiento local será el problema que se abordará en el desarrollo del proyecto.

Se pretende hacer un estudio comparativo de posibles soluciones al problema de la evasión de obstáculos en robots autónomos. Se hará un enfoque del problema aplicado a *Autonomous Indoor Vehicles* o AIV. Estos son robots móviles autónomos diseñados para ambientes interiores [**?**]. Este tipo de robots móviles tiene aplicaciones en industria, seguridad, salud, e incluso en el mercado de consumidores, como es el caso de los Roomba®. En general son robots que utilizan diferentes configuraciones de ruedas motorizadas para el control de su postura y traslación, en capítulos posteriores se hará una descripción más detallada de los mismos.

Se plantea una situación en la que un AIV, al momento de trasladarse mientra sigue una trayectoria dada, se encuentra con uno o más obstáculos que le impiden seguir el camino original, y debe recalcular o modificar su trayectoria a partir de la información suministrada por los sensores que disponga para la detección de estos obstáculos.

El proyecto constará de una investigación bibliográfica, en la que se elaborará un marco teórico del problema y se describirán al menos tres algoritmos de evasión de obstáculos, además se estudiará la forma en que estos pueden acoplarse a los algoritmos de seguimiento de trayectorias, y si la naturaleza de cada algoritmo permite esta posibilidad, esto corresponde al contenido del Capítulo 2.

Una vez se haya desarrollado el marco teórico sólido del problema, se procederá a implementar los tres algoritmos seleccionados en un lenguaje de programación de alto nivel. La documentación y detalles de la implementación se elaborarán en el Capítulo 3.

Posteriormente se usará un software de simulación para observar el funcionamiento de los algoritmos, se definirán los aspectos a evaluar y las diferentes pistas de prueba, se presentará el procedimiento y los datos obtenidos en el Capítulo 4.

Finalmente, en el Capítulo 5 se hará una comparación de los algoritmos a partir de los datos obtenidos, desempeño, ventajas y desventajas de cada algoritmo, y se presentarán las conclusiones del proyecto. Se pretende que los resultados de este proyecto, y en especial las implementaciones propuestas, eventualmente se puedan usar de base para manejar la evasión de obstáculos en un sistema complejo que incluya además otros aspectos relevantes a la navegación autónoma.

Como anexos se incluirá el texto completo del código desarrollado, así como un manual de instalación para repetir las simulaciones propuestas.

1.2. Alcances y Delimitación

Está dentro del alcance del proyecto:

- Descripción teórica del problema de la evasión de obstáculos en el contexto de un AIV.
- Descripción teórica de 3 o más algoritmos de evasión.
- Implementación de los algoritmos de evasión.
- Uso de la herramienta de simulación y conexión del ambiente de simulación con los algoritmos implementados.

Está fuera del alcance:

- Modelado de los actuadores del robot.
- Elaboración del sistema de control para los actuadores del robot.
- Implementación de algoritmos de planeamiento de trayectorias en un robot real.
- Construcción de los sistemas propuestos.

1.3. Justificación

El presente trabajo de investigación tiene relevancia en muchas áreas dentro y fuera de la robótica, en especial para aplicaciones que requieren de automatización. Los algoritmos desarrollados en este proyecto se podrían usar en todo tipo de aplicaciones: cualquier dispositivo que requiera moverse sin control humano ocupa ser capaz de evadir obstáculos. Se podrían usar los algoritmos desarrollados en ambientes industriales por ejemplo para bodegaje, para desarrollar un sistema de robots que sean capaces de mover cargas pesadas sin intervención humana. La evasión de obstáculos también es necesaria en sistemas de piloto automático, por ejemplo en el desarrollo de carros inteligentes.

En cuanto al valor de esta investigación para la Escuela de Ingeniería Eléctrica, hay varios laboratorios con investigaciones en el área de robótica y autonomía. Implementar al menos un algoritmo de evasión sería casi un requerimiento para cualquier robot verdaderamente autónomo que se pretenda desarrollar. Así el desarrollo de este proyecto es de utilidad para todos los laboratorios relacionados con esta área de investigación.

1.4. Objetivos

1.4.1. Objetivo General

- Realizar un estudio comparativo de distintos algoritmos de evasión de obstáculos con seguimiento de trayectorias para el control de la evolución de la postura en robots móviles navegando en un plano horizontal, destacando sus distintas ventajas y desventajas al realizar la implementación de los mismos en un software de simulación de robots móviles.

1.4.2. Objetivos Específicos

1. Realizar una investigación bibliográfica detallada sobre los distintos algoritmos de evasión de obstáculos con y sin seguimiento de trayectorias para el control de la evolución de la postura en robots móviles navegando en un plano horizontal.
2. Describir los algoritmos investigados y analizados mediante el uso de pseudocódigos para facilitar su comprensión e implementación.
3. Implementar al menos tres algoritmos de evasión de obstáculos en un lenguaje de programación de alto nivel.
4. Simular el sistema de evasión de obstáculos, usando los algoritmos mencionados, mediante un software de simulación.
5. Validar los algoritmos investigados mediante la simulación del movimiento del robot móvil en distintos entornos virtuales para medir la evolución de su postura en múltiples pruebas navegación con diversos obstáculos.
6. Realizar un estudio comparativo de las ventajas y desventajas de los distintos algoritmos estudiados, según las pruebas realizadas.
7. Documentar la implementación de los algoritmos y el uso del simulador, así como el código utilizado en las diversas pruebas de navegación, para su uso en trabajos futuros.

1.5. Metodología

Para el desarrollo del proyecto se considerarán cuatro áreas de enfoque en las que se dividirá el trabajo a realizar. Estás son el aspecto teórico, la implementación de los algoritmos, la simulación y el análisis de los resultados. A continuación se listan las actividades que se llevarán a cabo para cada una de estas áreas, entre paréntesis se indican los objetivos específicos con los que están directamente relacionadas.

- **Investigación bibliográfica y teórica:**
  - Reseña histórica de los robots terrestres autónomos. (1)
  - Descripción del robot y los sensores relevantes a la detección de obstáculos. (1)

  - Resumen del modelo matemático del sistema robot-obstáculo pertinente al desarrollo del proyecto, definición formal del problema. (1)
  - Reseña histórica y descripción teórica de los algoritmos de evasión de obstáculos (2).
  - Descripción en pseudocódigo de los algoritmos. (2)
  - Análisis teórico de la complejidad temporal y espacial de los algoritmos. (1, 2)

- **Implementación de los algoritmos:**

  - Definición de la representación de los datos de entrada y salida. (2, 3)
  - Implementación, en C++ o Python. (3)
  - Uso de librerías externas. (3)
  - Documentación del código. (7)
  - *Debugging*, unit tests. (3)
  - Integración con la herramienta de simulación. (3, 4, 5)

- **Simulación y recolección de datos**

  - Reseña de la herramienta de simulación. (7)
  - Integración del modelo del robot con el sensor y los algoritmos de evasión. (4, 5)
  - Definir los entornos de las pruebas. (4, 5)
  - Definir cantidad de pruebas a realizar por algoritmo y entorno. (5)
  - Definir criterios de evaluación y metodología de comparación. (5, 6)
  - Definir cantidad de pruebas a realizar por algoritmo y entorno. (5, 6)

- **Análisis de resultados**

  - Análisis del desempeño de los algoritmos, eficacia y desempeño. (6)
  - Comparación cuantitativa: elaboración de tablas y gráficos comparativos a partir de los indicadores seleccionados en la definición de las pruebas. (6)
  - Comparación cualitativa: ventajas y desventajas, facilidad de implementación, requerimientos, limitantes, uso del software de simulación. (6, 7)
  - Elaboración de una guía de usuario. (7)

Capítulo 2

Marco Teórico

2.1. Robots autónomos y definición del problema

Primero es relevante definir qué es un robot autónomo terrestre.

Se puede empezar aclarando que los robots móviles son aquellos diseñados con la capacidad de moverse por cierto ambiente. Estos en contraposición a los brazos robóticos que dominan las aplicaciones industriales, un robot móvil ofrece una flexibilidad mucho mayor en comparación, pues en dado caso sería capaz de aplicar sus habilidades en toda la planta de producción, y no solo en el lugar en donde fue instalado [**?**]. Por supuesto, la movilidad también aumenta considerablemente la complejidad del robot. En este aspecto, se pueden clasificar según el ambiente para el que fueron diseñados como se hace en [**?**]:

- Vehículos terrestres o *UGVs*. A su vez se pueden subdividir según el mecanismo de traslación:
  - Robots con piernas.
  - Robots con ruedas.
  - Robots con orugas.
- Vehículos aéreos o *UAVs*.
- Vehículos submarinos o *AUVs*.

Este trabajo se enfoca en los robots terrestres que usan ruedas como mecanismo de traslación, llamados *WMR* por sus siglas en inglés (*Wheeled Mobile Robots*).

Los robots también se pueden clasificar según su grado de autonomía. Existen los **robots teleoperados**, estos son aquellos que son controlados por un ser humano de forma remota. Cualquier sistema "inteligente" en estos robots se limita a facilitar al operador humano el control del robot, ya sea suministrando información del ambiente o mejorando la respuesta de los controles [**?**]. Los **robots automatizados guiados** son capaces de seguir una ruta mediante el uso de sensores o por medios mecánicos, por ejemplo seguir un camino pintado en el piso. Estos son análogos a un tren que sigue la vía férrea, su autonomía está restringida al camino que deben seguir. Finalmente, los **robots autónomos**

o *AMRs* pueden detectar obstáculos en su camino hacia un objetivo y planear su propio camino [**?**], el presente proyecto se enfoca en *WMRs* que son a su vez *AMRs*.

Según [**?**], para cumplir los requisitos de autonomía un robot debe tener las siguientes características y habilidades:

- **Movilidad**
- **Adaptabilidad a situaciones desconocidas**
- **Percepción del ambiente**
- **Adquisición de conocimiento**
- **Interacción con un operario u otros agentes**
- **Seguridad**
- **Procesamiento en tiempo real**

El problema de evasión de obstáculos involucra varios de estos aspectos. Para lograr verdadera movilidad autónoma, un robot debe ser capaz de responder a las siguientes preguntas [**?**]:

- ¿Cómo me muevo? ⟶ Cinématica y control de los actuadores.
- ¿Cómo es el medio ambiente? ⟶ Percepción y construcción de mapas.
- ¿Dónde estoy? ⟶ Localización.
- ¿Cómo llego a mi objetivo? ⟶ Planificación de movimiento.

La evasión de obstáculos es entonces un subproblema del problema más general de la navegación autónoma. En el contexto de este proyecto, se asume que los problemas de localización y control de los actuadores están resueltos por algún módulo externo que es parte del sistema robótico. Además en cuanto a la planificación de movimiento se debe hacer la distinción entre **planeamiento global** y **planeamiento local**, [**?**].

El primero se encarga de encontrar un camino adecuado de un punto de inicio a un punto objetivo, tal que se cumplan ciertas restricciones u optimizaciones. Los algoritmos de planeamiento global suelen describir la ruta obtenida como una serie de puntos que el robot es capaz de seguir. Además, se suelen construir a partir de alguna representación global del ambiente del robot, es decir se dispone de al menos cierta información *a priori*.

El planeamiento local por otro lado se encarga de generar un ruta tomando en cuenta el ambiente inmediato alrededor del robot, ambiente que no necesariamente se conoce *a priori*, por lo que es parte de la solución construir una representación de la localidad del robot a partir de los datos suministrados por los sensores. Es claro que los algoritmos de evasión caen en este segundo tipo de planeamiento, y el proyecto se limita a estudiar este tipo de algoritmos. Una desventaja inherente de los algoritmos

locales es que tienden a quedarse atrapados en mínimos locales (por ejemplo en "calles sin salida"). Para atacar estos casos resulta conveniente poder combinar planeamiento local y global: el algoritmo global dirige al robot por una ruta que llegue al objetivo mientras el algoritmo local reacciona en tiempo real a cualquier obstáculo que se presente en el camino.

Finalmente, se puede definir formalmente las condiciones del problema que se desea resolver con los algoritmos que se desarrollan en secciones posteriores.

2.1.1. Problema

Dado un robot autónomo que no necesariamente tiene información[1] de su ambiente y detecta un obstáculo en su cercanía, este debe ser capaz de construir una representación interna del obstáculo para recalcular su trayectoria en tiempo real y evitar una colisión. Además en caso de que el robot se encontrara siguiendo una trayectoria, este debe ser capaz de evadirlo mientras se dirige hacia un punto objetivo, o bien retomar la trayectoria una vez que no haya obstáculos impidiendo directamente su camino.

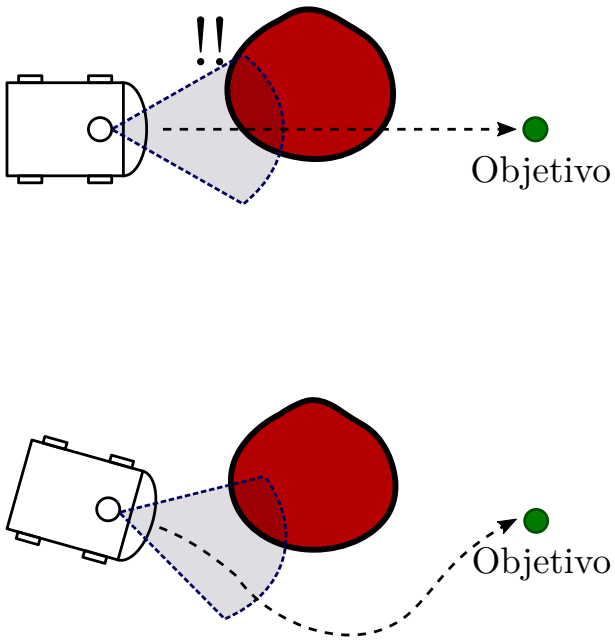


Figura 2.1: Problema que pretende resolver el proyecto. Un robot móvil detecta un obstáculo y modifica su trayectoria.

[1]Esta información puede ser por ejemplo un mapa que describa la ubicación de obstáculos, paredes, pasillos, etc.

Supuestos

1. La postura del robot (x, y, θ) respecto a un marco de referencia global es conocida con relativa certeza en todo momento.
2. El robot cuenta con sensores capaces de detectar obstáculos en la vecindad del robot en tiempo real.
3. Las lecturas de los sensores se representan como una serie de n puntos $\boldsymbol{p} = (r, \theta)$ dados en coordenadas polares respecto al marco de referencia local del robot.
4. El objetivo hacia el cuál debe dirigirse el robot (si lo hubiese) se representa simplemente como un punto (x, y) en el marco de referencia global.
5. El algoritmo de evasión genera como salida comandos de movimiento a alto nivel, ya sea rapidez y dirección (v, θ) o bien rapidez y velocidad angular (v, ω).
6. A partir de estos comandos, el robot es capaz de generar las respuestas apropiadas para sus motores mediante otros módulos.

2.2. Modelado

2.2.1. Marcos de referencia y transformaciones

En general la *postura* de un robot en un marco de referencia cartesiano tridimensional se puede describir con una séxtupla $(x, y, z, \alpha, \beta, \theta)$, que describe la posición y orientación del robot respecto a un marco de referencia absoluto [**?**]. En general, un vector de posición respecto a un marco de referencia arbitrario O se puede expresar respecto al marco de referencia absoluto B mediante la ecuación (2.2), donde ${}^{O}_{B}R(\alpha, \beta, \theta)$ es la matriz de rotación de O a B. La distinción entre marco de referencia local y absoluto del robot se puede apreciar en la figura 2.3.

$$\vec{\boldsymbol{u}}_{\boldsymbol{r}} = (x_r, y_r, z_r, \alpha_r, \beta_r, \theta_r) \tag{2.1}$$

$${}^{B}\vec{\boldsymbol{u}} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} + {}^{O}_{B}R(\alpha, \beta, \theta)\, {}^{O}\vec{\boldsymbol{u}} \tag{2.2}$$

Para un robot que se mueve sobre un plano bidimensional se puede simplificar la descripción de la postura a (x, y, θ) en el caso de coordenadas Eulerianas (z, α, y β son constantes). En este caso la matriz de rotación tridimensional respecto a θ es:

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{2.3}$$

$$\vec{\boldsymbol{u}}_{\boldsymbol{r2D}} = (x_r, y_r, \theta_r) \tag{2.4}$$

Una transformación relevante al posterior desarrollo de los algoritmos es la que se describe en la ecuación (2.5). Para un punto $\vec{\boldsymbol{p}}_{\boldsymbol{r}} = (r, \phi)$ expresado en coordenadas polares respecto al marco de referencia del robot, obtener el mismo punto $\vec{\boldsymbol{p}}_{\boldsymbol{0}} = (x, y)$ en el marco de referencia absoluto conociendo $\vec{\boldsymbol{u}}_{\boldsymbol{r2D}}$.

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} x_r \\ y_r \end{pmatrix} + r \begin{pmatrix} \cos(\theta_r + \phi) \\ \sin(\theta_r + \phi) \end{pmatrix} \tag{2.5}$$

El estudio de transformaciones es muy extenso, en esta sección se presentan únicamente las ecuaciones más relevantes al desarrollo e implementación del proyecto. Para más información de cómo se obtienen estas ecuaciones se recomienda consultar [?].

2.2.2. Restricciones al movimiento impuestas por las ruedas

Uno de los principales aspectos a considerar al modelar un robot es la configuración y tipo de ruedas que utiliza. En general se habla de 4 o 5 tipos diferentes de rueda en la literatura de *AIVs*: ruedas comunes fijas u orientables (giratorias), ruedas tipo *castor*, ruedas *Mecanum* o Suecas y ruedas esféricas ([?], [?], [?]). La figura 2.2 muestra estos diferentes tipos de ruedas.

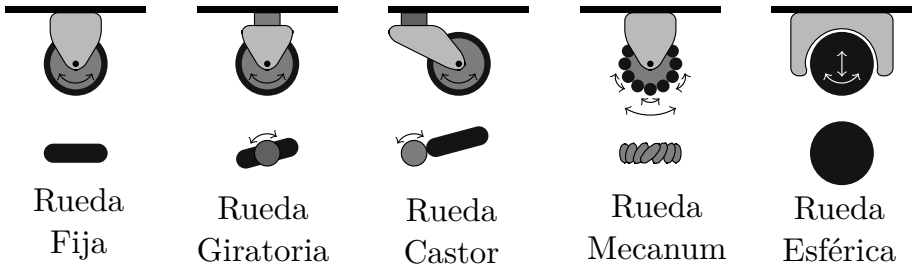


Figura 2.2: Diferentes tipos de ruedas usadas en AIVs, tomado de [?]

La configuración de las ruedas de un robot impone restricciones a su movimiento. Estas restricciones suelen estar directamente relacionadas al deslizamiento de las ruedas, por ejemplo una rueda fija no se puede deslizar lateralmente respecto a su eje de rotación. En [?] se describen con detalle las restricciones que impone cada tipo de rueda en el movimiento del robot. A partir de estas restricciones se puede desarrollar el modelo cinemático de un *AIV*, a su vez también se pueden calcular el *grado de movilidad* δ_m y el *grado de direccionabilidad* δ_s. Finalmente, el *grado de maniobrabilidad* δ_M se define como la suma de los dos anteriores, ecuación (2.6). La tabla 2.1 resume la clasificación de los diferentes tipos de robots según sus grados de movilidad y direccionabilidad (δ_m, δ_s), nótese que solo es relevante la cantidad de ruedas fijas y orientables, esto se deduce del hecho que los otros tipos de ruedas no imponen restricciones sobre el movimiento del robot (son omnidireccionales).

$$\delta_M = \delta_m + \delta_s \tag{2.6}$$

Tabla 2.1: Los 5 tipos genéricos de vehículos con ruedas [?].

Tipo (δ_m, δ_s)	Configuración de las ruedas	Ejemplo
(3, 0)	No hay ruedas comunes, tiene únicamente ruedas omnidireccionales (castor, mecanum o esféricas).	Robot Mecanum
(2, 0)	No hay ruedas orientables, y hay al menos una rueda fija. Si hay más de una rueda fija estas tienen el mismo eje de rotación.	Robot diferencial
(2, 1)	No hay ruedas fijas y hay al menos una rueda orientable. Si hay más de una rueda orientable estas están coordinadas tal que $\delta_s = 1$.	*Synchronous Drive*
(1, 1)	Hay al menos una rueda fija y al menos una rueda orientable. Si hay más de una rueda orientable estas están coordinadas tal que $\delta_s = 1$.	Vehículo Ackerman, triciclo
(1, 2)	No hay ruedas fijas y hay al menos dos ruedas orientables independientes. Si hay más de dos ruedas orientables estas están coordinadas tal que $\delta_s = 2$.	Configuración poco común en la práctica

Las restricciones cinemáticas descritas por estos grados son importantes por la forma en que describen la facilidad con que un robot se puede mover en su ambiente. Físicamente, el grado de movilidad indica los grados de libertad instantáneos en el movimiento de un robot, es decir un robot con $\delta_m = 3$ puede modificar sus x, y, γ instantáneamente y de manera independiente. El grado de direccionabilidad indica grados de libertad adicionales no instantáneos, que el robot puede controlar mediante la velocidad de rotación de sus ruedas orientables [?]. El ejemplo más ilustrativo probablemente es el vehículo Ackerman (la configuración usada en la mayoría de automóviles). Este solo se puede mover en una dirección en dado momento, hacia adelante o hacia atrás. Sin embargo, al cambiar la orientación de sus ruedas giratorias con el tiempo, obtiene un segundo grado de libertad en su movimiento que le permite modificar su orientación y moverse "lateralmente" (de manera dependiente una de la otra).

Entendiendo los conceptos de movilidad, direccionabilidad y maniobrabilidad, se puede explicar la distinción que se hace en robótica de robots holonómicos y omnidireccionales. Un robot (3, 0) no tiene ninguna restricción en su movimiento debido a sus ruedas, es capaz de moverse en cualquier dirección independientemente de su orientación γ, se dice que es un robot omnidireccional u *holonómico*. En robótica se entiende que un robot es holonómico si el total de grados de libertad de movimiento del robot es igual al número de grados de libertad controlables. Según esta definición un robot holonómico es aquel para el que $\delta_M = \delta_m = 3$ en el caso de movimiento en un plano horizontal. Cabe resaltar que esto difiere del sentido estricto de holonomía geométrica, decir que un robot de este tipo es omnidirec-

cional sería más correcto, pero los términos se suelen usar indistintamente [?]. En general los robots holonómicos suelen ser más complejos que los no-holonómicos, en especial por los controladores que requieren las ruedas omnidireccionales.

2.2.3. Robot Diferencial

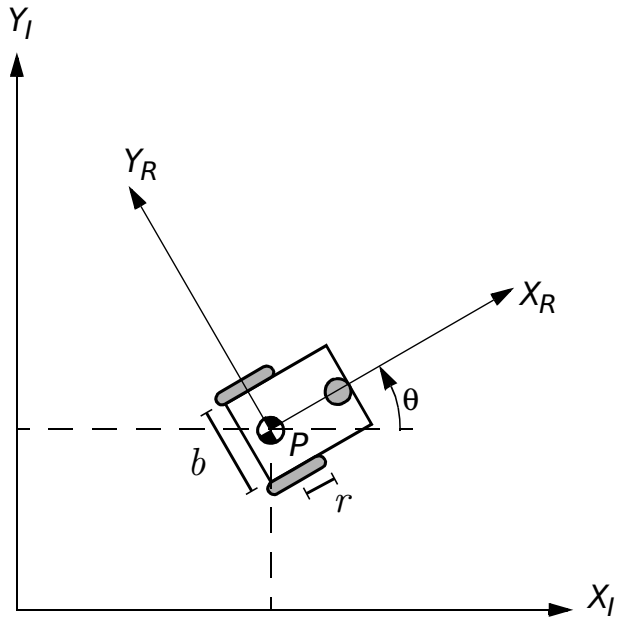


Figura 2.3: Diagrama básico de un robot diferencial, tomado de [?].

Para el desarrollo de este proyecto se decidió usar un robot diferencial en las simulaciones. A pesar de que los algoritmos son generales y no dependen directamente del tipo de robot, era necesario escoger algún modelo para las pruebas y el robot diferencial es probablemente el más sencillo. La Figura 2.3 muestra la geometría del robot, consta de dos ruedas comunes fijas de radio r separadas una distancia b. Cada rueda es controlada por un motor que la hace girar a una cierta velocidad angular. Además, suele tener una o más ruedas omnidireccionales que sirven como punto de apoyo para estabilizar el robot. Luego, el modelo cinemático del robot es:

$$\begin{bmatrix} v \\ w \end{bmatrix} = \begin{bmatrix} \frac{1}{2} & \frac{1}{2} \\ -\frac{1}{2b} & \frac{1}{2b} \end{bmatrix} \begin{bmatrix} v_L \\ v_R \end{bmatrix} \tag{2.7}$$

El modelo cinemático inverso es:

$$\begin{bmatrix} v_L \\ v_R \end{bmatrix} = \begin{bmatrix} 1 & -b \\ 1 & b \end{bmatrix} \begin{bmatrix} v \\ w \end{bmatrix} \tag{2.8}$$

Donde

$$v_L = r \cdot \phi_L \tag{2.9a}$$

$$v_R = r \cdot \phi_R \tag{2.9b}$$

Para ϕ_L y ϕ_R las velocidades angulares de la rueda izquierda y derecha respectivamente.

Todas estas ecuaciones se presentan a manera de resultado, el modelado general de robots se sale del alcance del proyecto. Para una explicación detallada de cómo obtener estas ecuaciones a partir de las restricciones cinemáticas de la geometría y configuración de las ruedas del robot se puede consultar [**?**] y [**?**].

2.3. Sensores

Hay una gran cantidad de sensores disponibles en el mercado para aplicaciones robóticas, que difieren considerablemente en su funcionamiento, forma de medición e interfaz con el robot o controlador. Se suelen clasificar de la siguiente manera (según [**?**] y [**?**]):

1. **Internos** o **externos**, dependiendo de si miden un estado propio del robot o bien si la medición se basa en una señal de referencia externa obtenida del ambiente del robot. Se suelen llamar también **propioceptivos** y **exteroceptivos** o **locales** y **globales** respectivamente.

2. **Activos** o **pasivos**. Los sensores activos interactúan con el ambiente para obtener la medición, por ejemplo emitiendo un haz de luz infrarroja o un ultrasonido, y midiendo la respuesta del ambiente. Los pasivos por otro lado miden una señal que ya está presente, pueden ser por ejemplo cámaras o micrófonos.

Debido a la gran cantidad de sensores que existen, esta sección se limita a describir algunos sensores usados en la detección de obstáculos, en particular los llamados *sensores de distancia*. Este tipo de sensores suelen ser activos y en general están basados en la detección de un haz que es rebotado en el objeto u obstáculo.

La energía de la señal medida depende de:

- La superficie de sensado B en el receptor.

- La superficie A del objeto impactado por el haz.

- La reflexión $\rho(\alpha)$ del haz sobre el objeto, que depende el ángulo de incidencia α y la reflectividad del material.

- La intensidad del haz (ya sea de luz o ultrasonido) en el objeto $f(R)$ que es una función de la distancia R.

2.3.1. Sonar

Los sensores de ultrasonido han sido históricamente de los más comunes en cuanto a detección de obstáculos [?]. Funcionan con el principio de "tiempo de vuelo", el sensor emite un pulso ultrasónico y cuenta el tiempo que le toma reflejarse en un objeto y volver al receptor y con esto calcula la distancia, la velocidad del pulso es conocida y constante para el rango de funcionamiento. Las frecuencias ultrasónicas sufren una gran atenuación al pasar por el aire, el límite para una señal de 50 kHz es de 11 m [?]. La mayoría de sensores ultrasónicos usados en robots móviles tiene un rango de entre 12 cm y 5 m, y alcanzan una resolución de aproximadamente 2 cm [?]. Estos sensores sufren de varias desventajas en la detección: son susceptibles a efectos de eco y reflexiones con paredes, interferencia o *crosstalk* si se usan varios sensores a la vez, y fallan en la detección de materiales suaves que absorben el sonido [?].

2.3.2. Lidar

Los sensores láser funcionan con un principio de tiempo de vuelo similar a los ultrasónicos. Sin embargo, para poder medir directamente el tiempo de vuelo se requiere precisión de picosegundos, esto es relativamente costoso. Si bien hay sensores que hacen precisamente esto, otros sensores usan un método más sencillo de determinar el tiempo de vuelo a partir del cambio de fase de la luz reflejada, figura 2.4. El sensor transmite luz modulada a una frecuencia conocida, esta sigue la ecuación (2.10). Luego la distancia al objecto detectado está dada por la ecuación (2.11), donde θ es la diferencia de fase entre el haz transmitido y el reflejado [?].

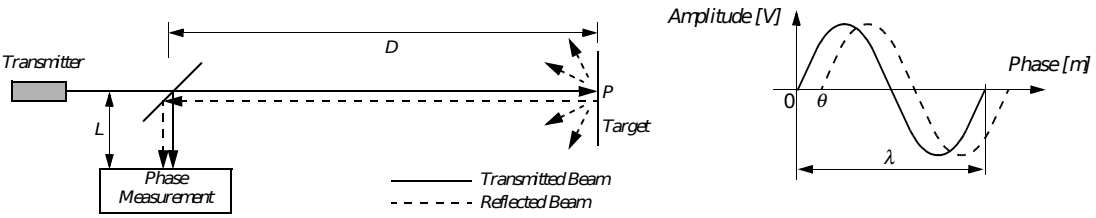


Figura 2.4: Medición de distancia a partir del cambio de fase, tomado de [?].

Si bien podría existir ambigüedad en la medición de distancia en caso de que la diferencia de fase fuera mayor a 360°, en la práctica la longitud de onda de la señal moduladora es mucho mayor que el rango del sensor debido a la atenuación del aire. Este tipo de sensores tiene un espejo giratorio que les permite escanear un plano, además tienen una resolución angular mucho mejor que el sonar. Se decidió usar este tipo de sensor en las simulaciones, pues por sus características se suele usar en la detección de obstáculos. Este tipo de sensores suele dar un conjunto de puntos de distancia ordenados $(r_i,\ \phi_i)$ respecto al marco de referencia del vehículo ([?]), como se planteó en la sección 2.1.1.

$$c = f \cdot \lambda \tag{2.10}$$

$$D = \frac{\lambda}{4\pi}\theta \tag{2.11}$$

2.3.3. Sensores infrarrojos de triangulación

Este tipo de sensores funciona con un principio diferente al "tiempo de vuelo" de los sensores láser y ultrasónicos. Consta de un diodo que emite un haz colimado de luz infrarroja, pero en este caso el receptor consta de un lente que enfoca la luz reflejada sobre un dispositivo de carga acoplada o *CCD*. Sabiendo la geometría del arreglo emisor-receptor se puede calcular la distancia a la que se encuentra el objeto. La ecuación (2.12) describe la relación entre la distancia al objeto D, la distancia l entre el LED y el CCD, la posición x del haz medido en el CCD y la distancia focal f del lente del receptor, figura 2.5.

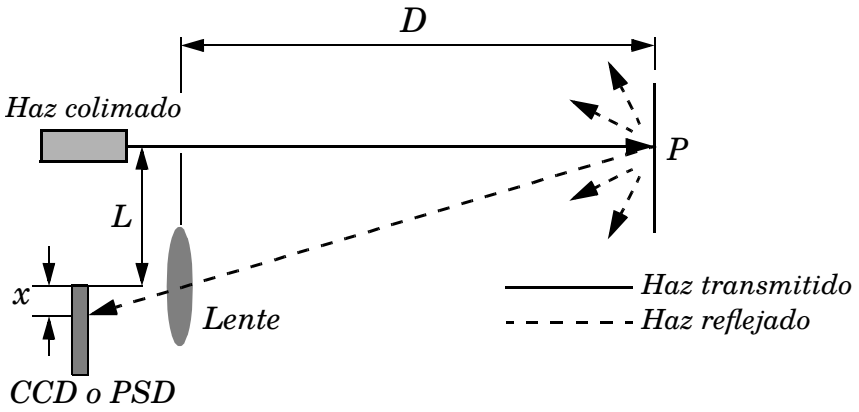


Figura 2.5: Sensor infrarrojo de distancia, tomado de [?].

$$d = f\frac{l}{x} \tag{2.12}$$

2.4. Algoritmos de Evasión

2.4.1. Vehículos de Braitenberg

Uno de los precursores de los robots autónomos móviles son los vehículos propuestos por Braitenberg en [?]. Estos no se exponen a manera de propuestas matemáticas o computacionales formales, sino más como un experimento mental, *ciencia ficticia* en palabras del autor, para demostrar cómo reacciones relativamente sencillas a estímulos pueden dar la impresión de un comportamiento inteligente. Incluso, Braitenberg llega a clasificar los vehículos según la "emoción" que parecen sentir en su respuesta al ambiente.

El vehículo más sencillo, llamado simplemente vehículo 1, cuenta con un sensor y un motor, conectados de forma muy rudimentaria: el motor irá más rápido entre mayor sea la lectura del sensor (independientemente de lo que se esté sensando). El vehículo puede moverse únicamente hacia delante y su movimiento será errático, dependerá en gran parte de las perturbaciones causadas por el ambiente en el que se mueva. Sin embargo, este vehículo demuestra la idea detrás de un vehículo de Braitenberg, una serie de sensores y motores interconectados, de modo que un sensor modifica el comportamiento de un motor, ya sea que vaya más rápido entre mayor sea la excitación del sensor, o por el contrario, una

mayor exitación del sensor inhiba el movimiento del motor. A lo largo de la obra [**?**] se elaboran comportamientos cada vez más complejos a partir de estas conexiones relativamente simples. Este concepto es reminiscente a las redes neuronales.

El vehículo 2b cuenta con dos sensores y dos motores con una conexión cruzada como se muestra en la Figura 2.6. Supóngase que se tengan sensores de luz. Si una fuente de luz se coloca directamente frente al vehículo este se moverá hacia adelante, a mayor velocidad entre más cerca esté. Ahora supóngase que la fuente de luz se encuentra ligeramente a la izquierda. El sensor del lado izquierdo tendrá una lectura mayor, esto hará que el motor derecho se mueva a mayor velocidad que el izquierdo, y a su vez que el robot gire en dirección a la fuente de luz. Efectivamente, este vehículo tiende a moverse "agresivamente" hacia las fuentes del sensor [**?**]. Supóngase ahora que se sustituyen los sensores de luz por sensores infrarrojos, de modo que la excitación será la distancia en línea recta desde el sensor hasta el primer objeto. Este vehículo tendrá un comportamiento tal que se verá "atraído" al espacio libre, y tenderá a evitar obstáculos en su camino. Efectivamente, el algoritmo 2.1 que describe este comportamiento se podría implementar de manera sencilla mediante ecuaciones lineales, para obtener un comportamiento de evasión de obstáculos.

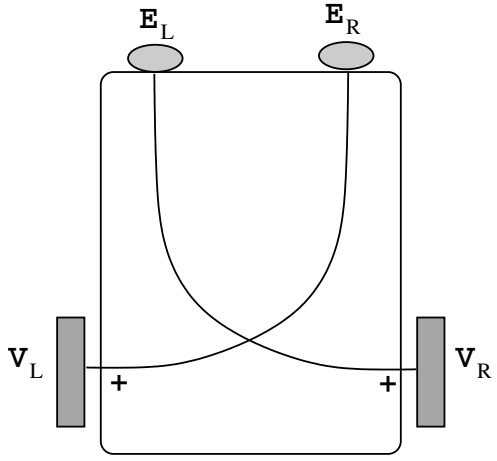


Figura 2.6: Vehículo 2b con conexión cruzada, tomado de [**?**].

El vehículo 3a tiene un comportamiento similar en cuanto el seguimiento, pero más bien tiende a disminuir su velocidad cuando se acerca a la fuente. La figura 2.7 muestra la conexión de sensores y motores, nótese que en este caso la lectura del sensor inhibe la acción del motor que se encuentra del mismo lado, como describe el algoritmo 2.2. Así, una diferencia en el nivel de estímulo de los sensores hace que el vehículo gire.

Si bien en [**?**] no se hace un análisis matemático formal de los vehículos propuestos, otros autores sí han hecho estudios matemáticamente rigurosos sobre el comportamiento de los diferentes vehículos. Por ejemplo, en [**?**] se hace un análisis de la convergencia de trayectorias de un vehículo 3a bajo la influencia de un estímulo parabólico. Este trabajo demuestra cómo se pueden obtener trayectorias estables, inestables y periódicas según las condiciones iniciales del robot.

Algoritmo 2.1: Algoritmo de vehículo 2b.

Entrada: Se tiene un robot como el de la figura 2.6, con un motor y un sensor del lado izquierdo, y otro motor y sensor del derecho.
Salida: La velocidad de ambos motores.

```
Si la lectura de un sensor aumenta, entonces:
    aumentar la velocidad del motor del lado contrario

Si la lectura de un sensor disminuye, entonces:
    disminuir la velocidad del motor del lado contrario
```

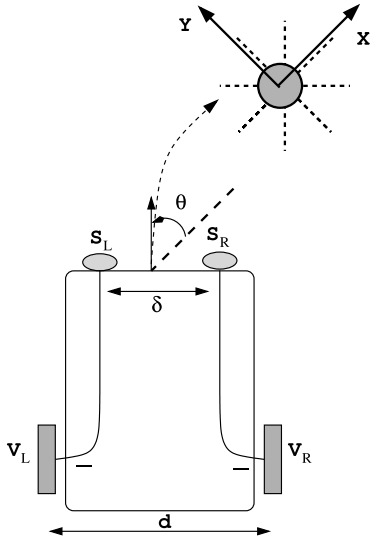


Figura 2.7: Vehículo 3a con acción inhibitoria de los sensores, tomado de [?].

Algoritmo 2.2: Algoritmo de vehículo 3a.

Entrada: Se tiene un robot como el de la figura 2.7, con un motor y un sensor del lado izquierdo, y otro motor y sensor del derecho.
Salida: La velocidad de ambos motores.

```
1 Si la lectura de un sensor aumenta, entonces:
2     disminuir la velocidad del motor del mismo lado.
3
4 Si la lectura de un sensor disminuye, entonces:
5     aumentar la velocidad del motor del mismo lado.
```

2.4.2. Vector Field Histogram

El algoritmo *VFH* fue desarrollado por Johann Borenstein en colaboración con otros autores. La versión original del algoritmo se presenta en [?]. Este desarrolla el concepto de histograma de obstáculos para representar la densidad probabilística de obstáculos alrededor del robot. A partir de este se construye un histograma polar de obstáculos que en última instancia se usa para determinar la dirección en la que debe dirigirse el robot. Borenstein presentó varias mejoras posteriores al algoritmo original. Primero el *VFH+* [?], en este se presentan 4 modificaciones puntuales que mejoran la respuesta del algoritmo considerablemente al tomar en cuenta las dimensiones y la dinámica del robot, reduciendo en gran medida la necesidad de ajustar parámetros. Estas se presentan más adelante en la sección 2.4.3. Posteriormente el *VFH** [?] añade características de planeamiento global al algoritmo.

A continuación se elaboran los elementos más importantes del algoritmo según se plantean en [?].

Cuadrícula de Certeza

La primera representación de datos que usa *VFH* es una cuadrícula $\boldsymbol{C}$ (llamada *Certainty Grid* en inglés), que se usa como modelo del medio ambiente del robot. En esta se construye una pseudo representación probabilística de los obstáculos que el *AIV* va sensando. Cada espacio $\boldsymbol{c_{i,j}}$ guarda un valor entero que representa la certeza de que algún objeto se encuentre realmente en ese espacio de la cuadrícula. La forma en que se actualizan los valores de $\boldsymbol{C}$ es muy sencilla: cada lectura de un sensor aumenta el valor de una única celda en uno. La ventaja de este método es que si los sensores realizan mediciones continuamente a una alta taza de muestreo, se logra una representación bastante certera de los obstáculos mientas el robot se traslada, y a un muy bajo coste computacional.

Una ventana que se mueve con el *AIV* define la región activa $\boldsymbol{C}^*$ de la cuadrícula $\boldsymbol{C}$ sobre la cual se realizan los cálculos. Esta ventana encuadra una región de $w_s \times w_s$ celdas centrada en la posición del robot, en la que cada celda $\boldsymbol{c_{i,j}}$ ahora representa un vector de obstáculos, donde su dirección β se determina de la celda al centro del vehículo

$$\beta_{i,j} = \tan^{-1} \frac{y_j - y_0}{x_i - x_0} \tag{2.13}$$

y la magnitud está dada por

$$m_{i,j} = c_{i,j}^2 \left(a - b d_{i,j}^2\right) \tag{2.14}$$

donde

$x_0,\ y_0$	son las coordenadas del centro del robot
$x_i,\ y_j$	son las coordenadas de la celda activa $c_{i,j}$
$c_{i,j}$	es el valor de certidumbre de la celda activa $c_{i,j}$
$d_{i,j}$	es la distancia del centro del robot a la celda activa $c_{i,j}$
$a,\ b$	son constantes positivas

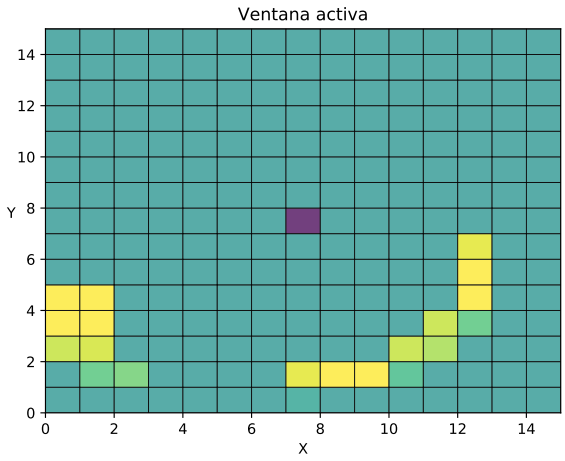


Figura 2.8: Ventana activa $\boldsymbol{C}^*$. Los colores más amarillentos representan valores más altos de certidumbre, el robot ocupa la celda morada. Obtenida en las simulaciones realizadas.

Las constantes a y b se eligen de tal forma que la celda activa más lejana tenga una magnitud de $c^2_{i,j}$. Es decir

$$a - b\,\frac{(w_s - 1)^2}{2} = 1 \tag{2.15}$$

Histograma Polar

Una vez se tiene el ángulo y la magnitud de todos los vectores de la ventana activa, estos se mapean al histograma polar $\boldsymbol{H}$. Este tiene una resolución arbitraria α tal que $360/\alpha$ resulta en un número entero de sectores. Se establece una correspondencia entre cada cada sector k y vector a partir su ángulo β

$$k = int\left(\beta_{i,j}/\alpha\right) \tag{2.16}$$

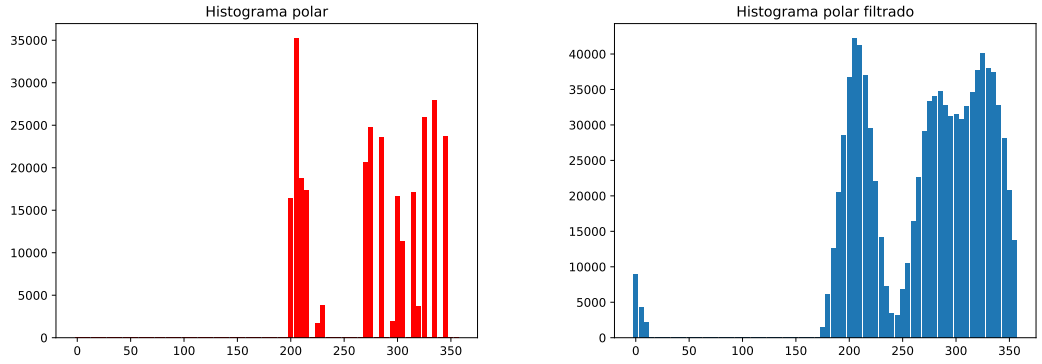
Luego, para cada sector k de $\boldsymbol{H}$, se define la densidad h_k

$$h_k = \sum m_{i,j} \tag{2.17}$$

Finalmente, se aplica un filtro a $\boldsymbol{H}$ tal que

$$h'_k = \frac{h_{k-l} + 2h_{k-l+1} + \cdots + lh_k + \cdots + 2h_{k+l-1} + h_{k+l}}{2l + 1} \tag{2.18}$$

La figura muestra la representación de un histograma polar filtrado que se podría obtener a partir de un curso de obstáculos sencillo.



(a) Histograma polar.

(b) Histograma polar filtrado.

Figura 2.9: Histogramas polares obtenido a partir de la ventana activa de la figura 2.8.

Control de la dirección

La dirección de giro del robot se obtiene a partir de un histograma polar como podría ser el de la figura 2.9. El histograma se divide en "valles" y "picos", que son respectivamente las secciones de menor y mayor densidad de obstáculos. La distinción se hace respecto a un cierto valor de umbral que se determina como un parámetro. Los grupos de sectores que se encuentren por debajo este umbral son los valles candidatos a ser la nueva dirección del movimiento. El algoritmo selecciona el valle que sea más cercano a la dirección del punto objetivo.

Borenstein además hace una distinción entre valles anchos y angostos, dependiendo de si constan de más de s_{max} sectores consecutivos. Sin embargo, la dirección se calcula de igual manera para ambos tipos. Sea k_{targ} el sector de la dirección al punto objetivo, se define k_n como el sector del valle más cercano a k_{targ} y k_f como el sector más lejano. Luego la dirección del robot es

$$\theta = \alpha \frac{\left(k_n + k_f\right)}{2} \tag{2.19}$$

De esta forma el robot sigue una trayectoria centrada entre los obstáculos. El algoritmo es poco sensible a la elección de y s_{max}, estos tienen poco efecto sobre el correcto funcionamiento del algoritmo excepto en casos de extremo desajuste [?].

Control de velocidad

Para determinar la velocidad del robot se toma en consideración la densidad de obstáculos h'_c en la dirección actual del movimiento del robot. Se define la velocidad máxima V_{max}, a partir de esta la densidad se calcula la velocidad deseada.

$$V' = V_{max}\left(1 - \frac{h''_c}{h_m}\right) \tag{2.20}$$

donde

$$h_c'' = \text{mín}(h_c', h_m) \tag{2.21}$$

Esto garantiza que el robot desacelere cuando tenga un obstáculo muy grande o muy cercano al frente. La constante h_m se debe determinar de manera empírica para la aplicación, de modo que ocasione una reducción suficiente de la velocidad.

Además del control de velocidad dado por (2.20) y (2.21), se puede hacer una reducción adicional tomando en consideración la tasa de viraje Ω del robot.

$$V = V' \left(1 - \frac{\Omega}{\Omega_{max}}\right) + V_{min} \tag{2.22}$$

donde Ω_{max} es la máxima velocidad angular permitida para el robot. Además V_{min} evita que el robot se detenga por completo.

El algoritmo 2.3 resume el planteamiento teórico de VFH, más adelante se verá que la implementación realizada difiere ligeramente de la teoría.

Algoritmo 2.3: Algoritmo de evasión VFH

Entrada: Lectura de los sensores , postura del robot, posición del objetivo.
Salida: Velocidad y dirección (v, θ)

```
1  Obtener la posición y dirección actual.
2  Leer los datos de los sensores.
3  Actualizar la cuadrícula de certeza C
4  Calcular los vectores de obstáculo de la ventana activa C*, según las
       ecuaciones (2.13) y (2.14).
5  Actualizar el histograma polar H según las ecuaciones (2.16) y (2.17).
6  Actualizar el histograma polar filtrado H´ según la ecuación (2.18).
7  Determinar los valles a partir del valor de s_max.
8  Determinar la dirección del robot θ a partir de la ecuación (2.19).
9  Calcular la velocidad v según las ecuaciones (2.20), (2.21) y (2.22).
10 Modificar la dirección del robot según (v, θ).
```

2.4.3. VFH+

El algoritmo VFH+ es una mejora del algoritmo VFH original propuesta por Borenstein en [**?**]. Al igual que en VFH se parte de una cuadrícula de certeza $\boldsymbol{C}$ con una ventana activa $\boldsymbol{C_a}$ que se mueve con el robot y contiene vectores de obstáculo, las ecuaciones (2.13), (2.14) y (2.15) se mantienen.

Histograma polar principal

El primer cambio en el algoritmo es la definición de un radio de ensanchamiento para las celdas de $\boldsymbol{C_a}$, que viene a compensar el ancho del robot. Este radio está definido por la ecuación (2.23), donde r_r es el radio del robot (que se define como la distancia del centro del robot al punto más alejado de su perímetro) y d_s es la distancia mínima entre el robot y un obstáculo. Este método de compensación viene a sustituir el filtro usado en VFH, ecuación (2.18), que de por sí era difícil de sintonizar para un funcionamiento óptimo. Luego, para cada celda se define el ángulo de ensanchamiento $\gamma_{i,j}$ en la ecuación (2.24).

$$r_{r+s} = r_r + d_s \tag{2.23}$$

$$\gamma_{i,j} = \arcsin \frac{r_{r+s}}{d_{i,j}} \tag{2.24}$$

Para cada sector k del histograma polar, la densidad de obstáculos se calcula como:

$$H_k^p = \sum_{i,j \in C_a} m_{i,j} \cdot h'_{i,j} \tag{2.25}$$

donde

$$\begin{cases} h'_{i,j} = 1 & \text{si } k \cdot \alpha \in \left[\beta_{i,j} - \gamma_{i,j}\,, \beta_{i,j} + \gamma_{i,j}\right] \\ h'_{i,j} = 0 & \text{de lo contrario} \end{cases} \tag{2.26}$$

La función h' sirve de filtro pasobajo y elimina la necesidad de filtrar el histograma polar $\boldsymbol{H^p}$, con la ventaja de que h' se determina analíticamente a partir de las dimensiones del robot y no empíricamente por prueba y error como era el caso en VFH.

Histograma polar binario

El algoritmo VFH original por lo general resultaba en trayectorias suaves. Sin embargo, a veces el valor de umbral T causaba problemas en ambientes con varios valles angostos. En este tipo de situación, sucedía a veces que el robot alternaba constantemente entre dos de estos valles, resultando en un comportamiento indeciso. Este problema se reduce fácilmente si se aplica una histéresis basada en dos valores de umbral τ_{low} y τ_{high}. A partir de $\boldsymbol{H^p}$ y estos valores de umbral, se construye un *histograma polar binario* $\boldsymbol{H^b}$, en vez de guardar valores de densidad de obstáculos, los sectores de $\boldsymbol{H^b}$ se encuentran simplemente *libres* (0) u *ocupados* (1). En cualquier instante de tiempo discreto n, el histogram binario se actualiza según las reglas de la ecuación (2.27).

$$\begin{cases} H_{k,n}^b = 1 & \text{si } H_{k,n}^p > \tau_{high} \\ H_{k,n}^b = 0 & \text{si } H_{k,n}^p < \tau_{low} \\ H_{k,n}^b = H_{k,n-1}^b & \text{en otro caso} \end{cases} \tag{2.27}$$

Histograma polar mascarado

El algoritmo VFH original obvia la dinámica del robot al asumir implícitamente que este puede cambiar de dirección de manera instantánea. Este no es el caso en la práctica, por esta razón se aproxima la trayectoria del robot mediante curvaturas $\kappa = 1/r$. La máxima curvatura suele ser una función de la velocidad del robot. Los radios de curvatura mínimos en función de la velocidad del robot serán $r_r = 1/\kappa_r$ y $r_l = 1/\kappa_l$. Con estos parámetros y la ventana activa se pueden determinar sectores adicionales que están bloqueados por la trayectoria del robot. Nuevamente las celdas con obstáculos se ensanchan un radio r_{r+s}, si un círculo de una celda ocupada traslapa con un círculo de trayectoria todas las direcciones desde el obstáculo hacia la dirección contraria al movimiento actual son bloqueadas.

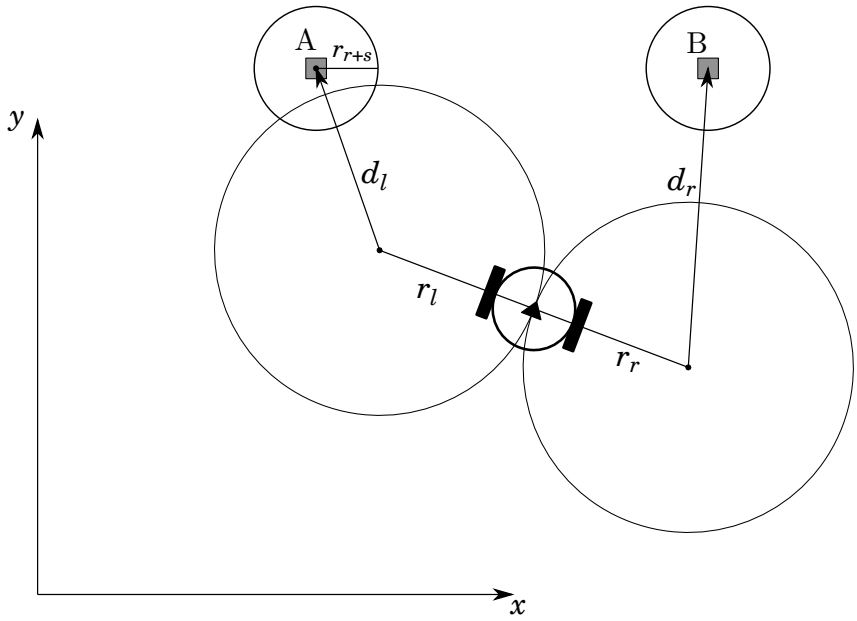


Figura 2.10: La celda A bloquea las direcciones a su izquierda.

La figura 2.10 ilustra esta situación. Para una celda que se encuentra ocupada se calcula la distancia desde el centro del mínimo radio de giro hasta la celda. Si esta distancia es menor a la suma del radio de giro y r_{r+s}, entonces la celda bloquea todas las direcciones posteriores. La posición del centro del círculo de giro respecto al marco de referencia de la ventana activa se puede calcular según la ecuación (2.28) ó (2.29), donde w_s es el tamaño de la ventana activa.

$$x_r = \frac{w_s}{2} + r_r \cdot \cos(\theta - 90) \qquad y_r = \frac{w_s}{2} + r_r \cdot \sin(\theta - 90) \tag{2.28}$$

$$x_l = \frac{w_s}{2} + r_l \cdot \cos(\theta + 90) \qquad y_l = \frac{w_s}{2} + r_l \cdot \sin(\theta + 90) \tag{2.29}$$

$$d_r^2 = (c_x - x_r)^2 + \left(c_y - y_r\right)^2 \tag{2.30}$$

$$d_l^2 = (c_x - x_l)^2 + \left(c_y - y_l\right)^2 \tag{2.31}$$

Una celda de C_a bloquea las direcciones a su derecha si:

$$d_r^2 < (r_r + r_{r+s})^2 \tag{2.32}$$

Una celda de C_a bloquea las direcciones a su izquierda si:

$$d_l^2 < (r_l + r_{r+s})^2 \tag{2.33}$$

Verificando estas condiciones para todas las celdas de la ventana activa se obtienen dos ángulos límite: ϕ_r y ϕ_l para ángulos a la derecha e izquierda respectivamente. Además se define $\phi_b = \theta + 180°$ como la dirección opuesta al movimiento actual. Esta etapa se puede implementar eficientemente con un algoritmo que verifique únicamente las celdas que puedan cambiar ϕ_l y ϕ_r. El fragmento de código 2.4 describe un algoritmo así.

Finalmente se puede construir el *diagrama polar mascarado* $\boldsymbol{H^m}$:

$$\begin{cases} H_k^m = 0 & \text{si } H_k^b = 0 \text{ y } (k \cdot \alpha) \in \{[\phi_r, \theta], [\theta, \phi_l]\} \\ H_k^m = 1 & \text{en caso contrario} \end{cases} \tag{2.34}$$

Control de la dirección

Al igual que en VFH original se hace una distinción entre valles angostos y anchos, si ocupan más de s_{max} sectores consecutivos. En el caso de valles angostos solo hay una dirección candidata que lleva la robot por el espacio en medio de los obstáculos.

$$c_d = \frac{k_r + k_l}{2} \tag{2.35}$$

Para valles anchos hay al menos dos direcciones candidatas, c_r a la derecha y c_l a la izquierda de la apertura. La dirección del objetivo también puede ser una dirección candidata si se encuentra entre las dos anteriores.

$$c_r = k_r + \frac{s_{max}}{2} \tag{2.36}$$

$$c_l = k_l - \frac{s_{max}}{2} \tag{2.37}$$

$$c_t = k_t \qquad \text{si } k_t \in [c_r, c_l] \tag{2.38}$$

Luego, para elegir la nueva dirección de entre las direcciones candidatas se aplica una función de costo, ecuación (2.39):

$$g(c) = \mu_1 \cdot \Delta\left(c, k_t\right) + \mu_2 \cdot \Delta\left(c, \theta_n\right) + \mu_3 \cdot \Delta\left(c, k_{d,n-1}\right) \tag{2.39}$$

Algoritmo 2.4: Algoritmo para determinar ϕ_l y ϕ_r.

Entrada: Postura del robot (x, y, θ), ventana activa, histograma polar mascarado, histograma polar binario, radios de giro r_{der} y r_{izq}.
Salida: Histograma polar mascarado.
Parámetros: r_{r+s}.

```
1  φ_b = θ + 180°
2  φ_r = φ_b
3  φ_l = φ_b
4
5  Para i de 0 a window_size:
6      Para j de 0 a window_size:
7          Si m_{i,j} está libre entonces:
8              Continuar a la siguiente celda.
9
10         Si β_{i,j} ∈ [θ, φ_l[ entonces:
11             Calcular la distancia, ecuación (2.31).
12             Si la distancia es menor a r_izq + r_{r+s} entonces:
13                 φ_l = β_{i,j}
14             Fin si
15         Si no, si β_{i,j} ∈ [φ_r, θ[ entonces:
16             Calcular la distancia, ecuación (2.30).
17             Si la distancia es menor a r_der + r_{r+s} entonces:
18                 φ_r = β_{i,j}
19             Fin si
20         Fin si
21     Fin para
22 Fin para
23
24 Retornar φ_l y φ_r
```

donde

$\Delta(\alpha, \beta)$	es una función que calcula el valor absoluto de la diferencia de ángulos entre α y β
c	es la dirección candidata
k_t	es la dirección del objetivo
θ_n	es la dirección actual del robot
$k_{d,n-1}$	es la dirección de giro que fue seleccionada previamente
μ_1 μ_2 μ_3	son parámetros que le dan peso a cada término asociado. Para que el robot tenga un comportamiento de seguir el objetivo se debe cumplir que $\mu_1 > \mu_2 + \mu_3$

Esta función de costo también permite alterar el comportamiento general del robot de forma más sutil, al cambiar los pesos de los parámetros. Finalmente, el algoritmo 2.5 resume el funcionamiento completo de VFH+, según se describió en esta sección.

Algoritmo 2.5: Algoritmo de evasión VFH+

Entrada: Lectura de los sensores , postura del robot, posición del objetivo, radios de giro.
Salida: Dirección (v, θ)

```
1  Obtener la posición y dirección actual.
2  Leer los datos de los sensores.
3  Actualizar la cuadrícula de certeza C
4  Calcular los vectores de obstáculo de la ventana activa C_a, según las
       ecuaciones (2.13) y (2.14).
5  Actualizar el histograma polar H^p según las ecuaciones (2.25) y (2.26).
6  Actualizar el histograma polar binario H^b según la ecuación (2.27).
7  Actualizar el histograma polar mascarado H^m según la ecuación (2.34) y
       el algoritmo 2.4.
8  Si todas las direcciones de H^m están bloqueadas, disminuir la velocidad
9  Determinar los valles a partir de s_max.
10 Determinar las direcciones candidatas, ecuaciones (2.35), (2.36), (2.37) y
       (2.38).
11 Determinar la dirección de menor costo, ecuación (2.39).
```

Capítulo 3

Implementación de los algoritmos seleccionados

3.1. Generalidades

Para la implementación de los algoritmos se tomó la decisión de usar el lenguaje de programación Python. Las siguientes fueron razones de peso en esta decisión:

- **Compatibilidad con el software de simulación:** El software de simulación que se usó para validar los algoritmos cuenta con soporte oficial de una API para Python.
- **Facilidad y rapidez de prototipado:** Python es un lenguaje interpretado de alto nivel, que permite una gran flexibilidad a la hora de programar, no requiere manejo explícito de memoria, además cuenta con un *shell* interactivo y está ampliamente documentado. Todo esto acelera considerablemente el proceso de desarrollo y reduce la cantidad de líneas de código necesarias, hasta la mitad de horas de desarrollo y líneas de código en comparación a lenguajes compilados como Java y C++ [**?**]. Este es un factor decisivo considerando el cronograma ajustado del proyecto eléctrico.
- **Disponibilidad de paquetes externos:** Python cuenta con varios paquetes para aplicaciones científicas y de robótica. En particular, en este proyecto se usaron extensivamente *NumPy* y *Matplotlib*.
- **Compatibilidad con ROS:** Python cuenta con una API oficial para ROS. A pesar de que ROS no se usó directamente en el desarrollo de este proyecto, sí es parte de los objetivos permitir compatibilidad e integración de los algoritmos a futuro en sistemas más complejos, y el CERLab y varios otros laboratorios de EIE hacen uso de ROS en los robots que desarrollan.

El código y la documentación del mismo se incluyen anexados al final del documento.
La estructura del proyecto es la siguiente:

- Py: En esta carpeta se encuentran los módulos de los algoritmos de evasión y del modelo del robot diferencial, en general el código "reutilizable" o "abstraíble" en otros sistemas. Este compone el grueso de las implementaciones desarrolladas.

  - `VFH.py`: Implementa el algoritmo de evasión VFH con la clase `VFHModel`.
  - `VFHP.py`: Implementa el algoritmo de evasión VFH+ con la clase `VFHPModel`.
  - `Braitenberg.py`: Implementa el algoritmo de evasión basado en Vehículos de Braitenberg con la clase `BraitModel`.
  - `DiffRobot.py`: Modela un robot diferencial, sus sensores y actuadores relevantes a la evasión de obstáculos.
  - `PID.py`: Una implementación sencilla de un controlador PID, se use dentro del modelo del robot diferencial para controlar la orientación θ (ecuación (2.4)).

- `Vrep`: En esta carpeta se encuentran los *scripts* y el código usado para la simulación con V-REP. Es decir, los programas que se corren desde el simulador usando la API de V-REP, y que instancian los algoritmos y modelos definidos en `Py`. Estos se detallan como parte de la simulación en el capítulo 4.

  - `HokuyoRob.py`: Programa que instancia los algoritmos de evasión y se conecta al simulador.
  - `ePuck.lua`: Script del robot ePuck.
  - `Hokuyo.lua`: Script del sensor Hokuyo.
  - `Pista1.tttt`: Escena de simulación del primer curso de obstáculos.
  - `Pista2.tttt`: Escena de simulación del segundo curso de obstáculos.
  - `Pista3.tttt`: Escena de simulación del segundo curso de obstáculos.

Los tres algoritmos de evasión se implementaron como clases, siguiendo un paradigma de programación orientada a objetos. Además, una cuarta clase modela el robot diferencial, e instancia los algoritmos de evasión. La figura 3.1 muestra un diagrama de clases del conjunto.

La documentación del código se incluye anexada al final de reporte. La misma se generó mediante el uso de la herramienta Sphinx. Se incluye además en formato de página web, en la copia digital del proyecto.

3.2. Algoritmos de Braitenberg

Para la implementación del algoritmo de evasión basado en los vehículos de Braitenberg se usó una combinación de los vehículos 2b y 3a según se describen en [**?**]. El funcionamiento a alto nivel del controlador se describe en el algoritmo 3.1.

La tabla 3.1 resume los parámetros usados por el controlador, estos se implementaron como atributos de clase. El controlador puede funcionar con 3 modos diferentes de sensado, mismos se resumen en la tabla 3.2. Se asume que se usa un sensor de distancia tipo Lidar, tal que el estímulo para cada lado se obtiene a partir de las mediciones o puntos que se encuentren en cierto sector angular. El estímulo izquierdo se obtiene a partir de las mediciones en el rango $[0, \alpha]$, mientras que el derecho se obtiene de las mediciones en el sector $[-\alpha, 0]$, ambos rangos respecto al marco de referencia del robot. Esto se implementó mediante el algoritmo 3.2 con $\alpha = \pi/6$ rad.

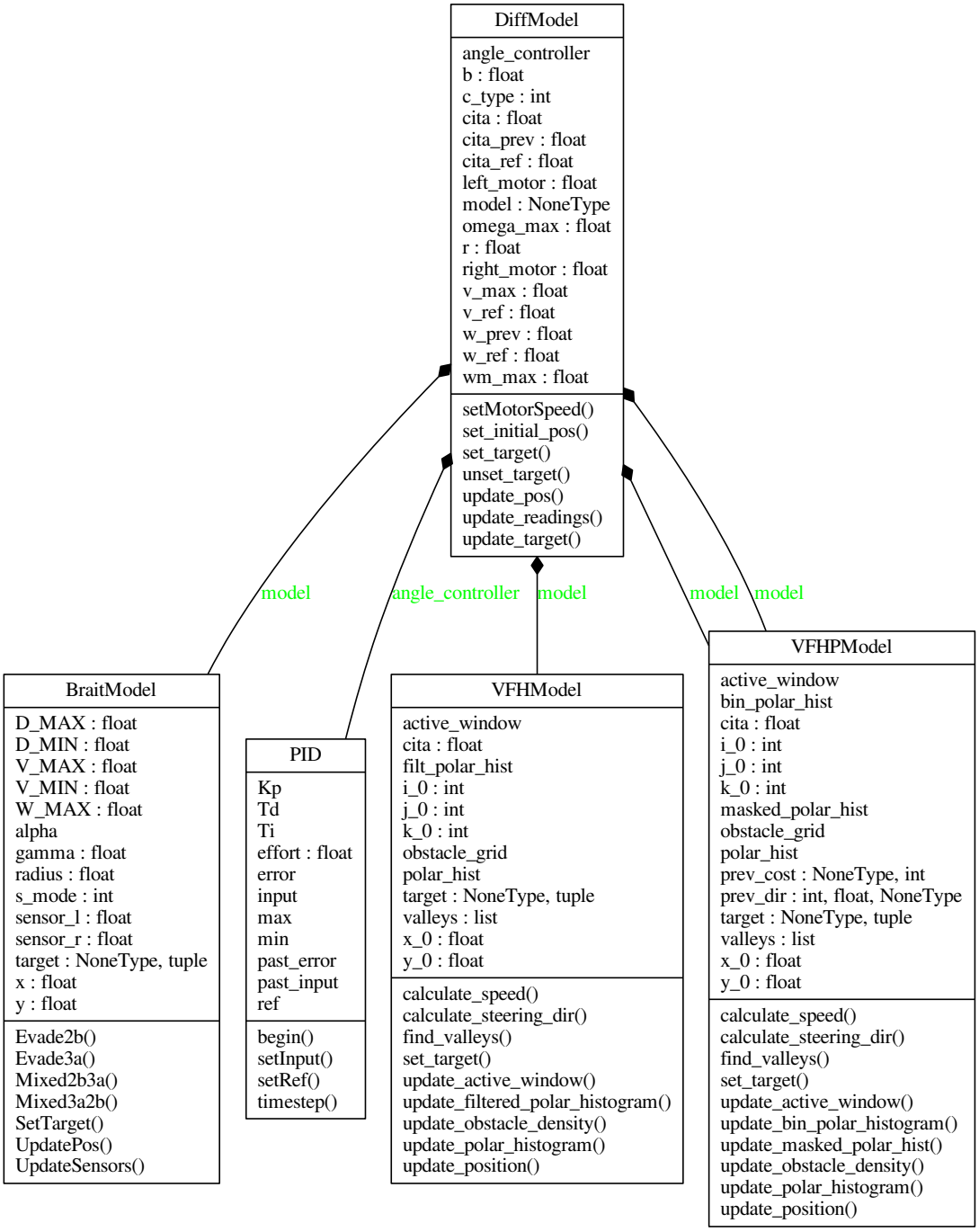


Figura 3.1: Diagrama de clases.

Algoritmo 3.1: Algoritmo de evasión según vehículos de Braitenberg.

Entrada: Datos del sensor, información de la posición del robot y el punto objetivo (si hay).
Salida: Una respuesta en forma de velocidad lineal y angular del robot (v, ω).
Parámetros: El tipo de evasión y seguimiento (2b o 3a).

```
Obtener las lecturas de los sensores
Calcular los estímulos para cada lado a partir de las lecturas
Obtener la respuesta de evasión según el vehículo 2b (o 3a)

si no hay objetivo entonces:
    Calcular la respuesta del robot a partir de la evasión
    retornar la respuesta total

si hay objetivo entonces:
    Calcular la distancia al objetivo
    Obtener la respuesta de seguimiento según el vehículo 3a (o 2b)
    Calcular la respuesta total a partir de la evasión y el seguimiento
    retornar la respuesta total
```

Tabla 3.1: Parámetros del algoritmo de Braitenberg.

Nombre	Descripción
`radius`	Radio del robot (en m).
`D_MIN`	Distancia mínima a los obstáculos (en m). Cualquier lectura menor de los sensores de distancia es equivalente a `D_MIN`.
`D_MAX`	Distancia máxima de detección de obstáculos (en m). Cualquier lectura mayor es ignorada.
`V_MIN`	Velocidad mínima del robot (en m/s).
`V_MAX`	Velocidad máxima del robot (en m/s).
`W_MAX`	Velocidad angular máxima del robot (en rad/s).
`alpha`	Rango angular de visión del robot (en radianes).
`smode`	Indica el modo de sensado según la tabla 3.2.
`MIN_READ`	Número mínimo lecturas en el modo `S_MODE_FULL`.

Para la evasión de obstáculos se hizo una implementación basada en transformaciones lineales. La idea es que mediante transformaciones lineales sencillas se relacionen los estímulos obtenidos de los sensores con la respuesta del robot siguiendo la lógica de los vehículos de Braitenberg. Para esto se define una función de *mapeo lineal*, según la ecuación (3.1), nótese que es la ecuación de una recta, pero con y acotado dentro de un cierto rango.

Algoritmo 3.2: Obtención de estímulos a partir de las medidas del sensor

Entrada: Los datos del sensor, una lista de puntos (r, ϕ).
Salida: Los estímulos $sensor_l$ y $sensor_r$
Parámetros: El modo de sensado `s_mode`, el rango de visión α

```
funcion ObtenerEstimulos(datos):

    definir dos arreglos vacíos: datos_izq, datos_der

    para cada par (r, φ) en datos hacer:
        si r es 0 y φ es 0 entonces:
            continuar con el siguiente dato
        si no, si -α < φ ≤ 0 entonces:
            agregar r a datos_izq
        si no, si 0 < φ < α entonces:
            agregar r a datos_der
        fin si
    fin para

    si s_mode es SMODE_MIN entonces:
        sensor_l = mínimo valor en datos_izq o D_MAX si está vacío
        sensor_r = mínimo valor en datos_der o D_MAX si está vacío

    si no, si s_mode es SMODE_AVG entonces:
        sensor_l = promedio de valores en datos_izq o D_MAX si está vacío
        sensor_r = promedio de valores en datos_der o D_MAX si está vacío

    si no, si s_mode es SMODE_FULL entonces:
        definir un número mínimo de datos requerido por cada lado N (depende del
            sensor)
        agregar a datos_izq valores D_MAX hasta que tenga al menos N elementos
        agregar a datos_der valores D_MAX hasta que tenga al menos N elementos
        sensor_l = promedio de valores en datos_izq
        sensor_r = promedio de valores en datos_der
    fin si

    retornar sensor_r, sensor_l
```

Tabla 3.2: Modos de operación del sensor.

Modo	Descripción
`SMODE_MIN`	El estímulo corresponde al mínimo valor de distancia detectado dentro del rango respectivo.
`SMODE_AVG`	El estímulo corresponde al promedio de los valores de distancia no nulos dentro del rango detectado. Si no hay ningún valor detectado, se asigna el valor máximo de distancia detectable.
`SMODE_FULL`	El estímulo corresponde al promedio de valores de distancia no nulos y se debe tener al menos n cantidad de mediciones no nulas. Si no se alcanza este número, se suman mediciones con el máximo valor de distancia hasta alcanzar el mínimo número de puntos, y se calcula el promedio. El valor de n está dado por el parámetro `MIN_READ`.

$$\texttt{Map}(x, x_1, x_2, y_1, y_2) = \begin{cases} y_2 & \text{si } x > x_2 \\ \frac{y_2 - y_1}{x_2 - x_1}(x - x_1) + y_1 & \text{si } x_1 \leq x \leq x_2 \\ y_1 & \text{si } x < x_1 \end{cases} \tag{3.1}$$

La clase define 4 métodos para la evasión:

- `Evade2b`: Evasión tipo 2b sin seguimiento de trayectoria algoritmo 3.3 .
- `Evade3a`: Evasión tipo 3a sin seguimiento de trayectoria algoritmo 3.4 .
- `Mixed2b3a`: Evasión tipo 2b con seguimiento de trayectoria tipo 3a algoritmo 3.5 .
- `Mixed3a2b`: Evasión tipo 3a con seguimiento de trayectoria tipo 2b algoritmo 3.6 .

Los algoritmos de evasión funcionan mediante transformaciones lineales. Se determina una rapidez y velocidad angular "normalizadas" en los rangos $[0,\ 1]$ y $[-0{,}5,\ 0{,}5]$ respectivamente a partir de los estímulos obtenidos de los sensores. Luego, estas se transforman linealmente al rango de operación del robot, dado por los parámetros v_{min}, v_{max} y ω_{max}. De este modo, la salida del controlador es un par (v, ω) tal que $v \in [v_{min},\ v_{max}]$ y $\omega \in [-\omega_{max},\ \omega_{max}]$.

El funcionamiento de los algoritmos mixtos es algo diferente, pues se debe definir también un estímulo para el comportamiento de dirigirse hacia el objetivo. Para esto se define un radio del robot r_R, y los puntos $\vec{\boldsymbol{u}}_r$ $\vec{\boldsymbol{u}}_l$ a la derecha e izquierda del centro del robot. Estos son perpendiculares a la dirección del robot θ y se encuentran a una distancia r_R del centro. La distancia $(d_r\ d_l)$ medida desde cada uno de estos puntos al objetivo será la base del estímulo de "cercanía".

$$\begin{pmatrix} x_{left} \\ y_{left} \end{pmatrix} = \begin{pmatrix} x_r \\ y_r \end{pmatrix} + r_{ROB} \begin{pmatrix} \cos(\theta + \frac{\pi}{2}) \\ \sin(\theta + \frac{\pi}{2}) \end{pmatrix} \tag{3.2}$$

Algoritmo 3.3: Evasión con vehículo 2b

Entrada: Estímulos s_l y s_r
Salida: Acción de control v, ω
Parámetros: D_MIN, D_MAX, V_MIN, V_MAX, W_MAX.

```
funcion Evasion2b(s_l, s_r):
    v_izq = Map(s_r, D_MIN, D_MAX, 0, 1)
    v_der = Map(s_l, D_MIN, D_MAX, 0, 1)

    v_norm = (v_der + v_izq)/2
    w_norm = (v_der - v_izq)/2

    v = Map(v, 0, 1, V_MIN, V_MAX)
    w = Map(w, -0.5, 0.5, -W_MAX, W_MAX)

    retornar v, w
```

Algoritmo 3.4: Evasión con vehículo 3a

Entrada: Lectura de los sensores s_l y s_r
Salida: Acción de control v, ω
Parámetros: D_MIN, D_MAX, V_MIN, V_MAX, W_MAX.

```
funcion Evasion3a(s_l, s_r):
    v_izq = Map(s_l, D_MIN, D_MAX, 1, 0)
    v_der = Map(s_r, D_MIN, D_MAX, 1, 0)

    v_norm = (v_der + v_izq)/2
    w_norm = (v_der - v_izq)/2

    v = Map(v, 0, 1, V_MIN, V_MAX)
    w = Map(w, -0.5, 0.5, -W_MAX, W_MAX)

    return v, w
```

$$\begin{pmatrix} x_{right} \\ y_{right} \end{pmatrix} = \begin{pmatrix} x_r \\ y_r \end{pmatrix} + r_{ROB} \begin{pmatrix} \cos(\theta - \frac{\pi}{2}) \\ \sin(\theta - \frac{\pi}{2}) \end{pmatrix} \tag{3.3}$$

El algoritmo 3.5 usa una combinación de la evasión por 2b y seguimiento 3a. Para calcular el estímulo de seguimiento se define una cercanía proporcional al inverso de la distancia al objetivo para cada lado, según lo que se mencionó. para que el efecto se mantenga aún a distancias muy lejanas se acotan estos valores. Si la distancia del lado más alejado es mayor que $2 \cdot r_{ROB}$ entonces se resta la diferencia a ambas distancias. A partir de la cercanía se calcula la velocidad normalizada para el algoritmo 3a, y luego se suman las respuestas de evasión y seguimiento, para obtener la respuesta final del robot. Un proceso análogo se sigue en el algoritmo 3.6

3.2.1. Elección de parámetros

Este controlador resultó muy fácil de parametrizar, tiene muy pocos parámetros y la mayoría simplemente son características físicas del robot. Los siguientes son algunos consejos para sintonizar los parámetros que quedan a criterio del usuario:

- `D_MIN` y `D_MAX` se pueden elegir según las limitaciones de distancia máxima y mínima de los sensores que use el robot. Sin embargo, también puede ayudar al desempeño limitarlos un poco más, en especial `D_MAX` si la distancia máxima del sensor es muy grande, de modo que el robot solo considere los obstáculos más cercanos.
- El rango de visión `alpha` no debe ser muy angosto ni demasiado amplio, un valor de 30° ($\frac{\pi}{6}$ rad) funciona bien.
- Los parámetros `V_MIN` y `V_MAX` afectan la velocidad promedio del robot. En las pruebas realizadas se observó que el principal limitante para el algortimo mixto 2b3a era `V_MAX`, el robot tendía a desplazarse a una velocidad cercana la mayor parte del tiempo. Además, aumentar el valor de `V_MIN` podía aumentar también la velocidad promedio (aunque en menor medida), principalmente en ambientes con gran cantidad de obstáculos, mas con la desventaja de que el robot vira hacia los espacios libres más lentamente. Por esta razón se recomienda que `V_MIN` sea un valor cercano a cero.
- Probablemente el parámetro que más efecto tiene sobre el comportamiento del robot es el modo de sensado `s_mode`, la elección depende enteramente de la naturaleza de los datos que provee el sensor. El modo que dio mejores resultados para el sensor Lidar usado en las simulaciones fue `SMODE_FULL`, con un valor de `MIN_READ` tal que fuera aproximadamente el máximo número de mediciones posibles según la resolución del sensor, tomando en cuenta `alpha`.

Algoritmo 3.5: Evasión con vehículo 2b y seguimiento con vehículo 3a.

Entrada: Lectura de los sensores s_l, s_r, postura del robot (x_r, y_r, θ_r), posición del objetivo (x_t, y_t) y radio del robot `R_ROB`
Salida: Acción de control v, ω
Parámetros: `D_MIN`, `D_MAX`, `V_MIN`, `V_MAX`, `W_MAX`, `R_ROB`.

```
funcion Mixta2b3a(s_l, s_r, x_r, y_r, cita, x_t, y_t):

    calcular las velocidades normalizadas para la respuesta de
       evasión
    v_izq_eva = Map(s_r, D_MIN, D_MAX, 0, 1)
    v_der_eva = Map(s_l, D_MIN, D_MAX, 0, 1)


    calcular las distancias d_l y d_r a partir de las ecuaciones
        (3.2) y (3.3)
    d_max = 2*r_ROB
    si d_l o d_r > d_max entonces:
        dif = máximo(d_l, d_r) - d_max
        d_l = d_l - diff
        d_r = d_r - diff
    fin si

    stim_l = r_ROB/d_l
    stim_r = r_ROB/d_r

    v_izq_seg = Map(s_l, D_MIN, D_MAX, 1, 0)
    v_der_seg = Map(s_r, D_MIN, D_MAX, 1, 0)

    ratio = 0.8
    v_izq = ratio*v_izq_eva + (1-ratio)*v_izq_seg
    v_der = ratio*v_der_eva + (1-ratio)*v_der_seg

    v_norm = (v_der + v_izq)/2
    w_norm = (v_der - v_izq)/2

    v = Map(v, 0, 1, V_MIN, V_MAX)
    w = Map(w, -0.5, 0.5, -W_MAX, W_MAX)

    return v, w
```

Algoritmo 3.6: Evasión con vehículo 3a y seguimiento con vehículo 2b.

Entrada: Lectura de los sensores s_l, s_r, postura del robot (x_r, y_r, θ_r), posición del objetivo (x_t, y_t) y radio del robot `R_ROB`
Salida: Acción de control v, ω
Parámetros: `D_MIN`, `D_MAX`, `V_MIN`, `V_MAX`, `W_MAX`, `R_ROB`.

```
funcion Mixta3a2b(s_l, s_r, x_r, y_r, cita, x_t, y_t):

    calcular las velocidades normalizadas para la respuesta de
       evasión
    v_izq_eva = Map(s_l, D_MIN, D_MAX, 1, 0)
    v_der_eva = Map(s_r, D_MIN, D_MAX, 1, 0)


    calcular las distancias d_l y d_r a partir de las ecuaciones
        (3.2) y (3.3)
    d_max = 2*r_ROB
    si d_l o d_r > d_max entonces:
        dif = máximo(d_l, d_r) - d_max
        d_l = d_l - diff
        d_r = d_r - diff
    fin si

    stim_l = r_ROB/d_l
    stim_r = r_ROB/d_r

    v_izq_seg = Map(s_l, D_MIN, D_MAX, 0, 1)
    v_der_seg = Map(s_r, D_MIN, D_MAX, 0, 1)

    ratio = 0.8
    v_izq = ratio*v_izq_eva + (1-ratio)*v_izq_seg
    v_der = ratio*v_der_eva + (1-ratio)*v_der_seg

    v_norm = (v_der + v_izq)/2
    w_norm = (v_der - v_izq)/2

    v = Map(v, 0, 1, V_MIN, V_MAX)
    w = Map(w, -0.5, 0.5, -W_MAX, W_MAX)

    return v, w
```

3.3. Algoritmo VFH

El algoritmo VFH se implementó mediante el módulo `VFH.py` que contiene la clase `VFHModel`. La implementación se hizo con base en la teoría desarrollada en la sección 2.4.2. Las constantes definidas como parámetros del módulo en la implementación se indican en la tabla 3.3. La clase VFHModel utiliza las variables miembro que se indican en la tabla 3.4. Las estructuras de datos más importantes de la clase son: `obstacle_grid`, `active_window`, `polar_hist`, `filt_polar_hist` y `valleys`. Estas son la representación de los datos que usa el modelo. Nótese que la ubicación del robot en la ventana activa no cambia, por lo que solo es necesario calcular las constantes $\beta_{i,j}$ y $d_{i,j}$ una única vez para cada celda, luego estas se pueden guardar en la misma estructura de datos.

Tabla 3.3: Parámetros de la implementación de VFH.

Nombre	Descripción
`GRID_SIZE`	Tamaño n de la cuadrícula de certeza $n \times n$.
`RESOLUTION`	Longitud del lado de cada celda de la cuadrícula en metros.
`WINDOW_SIZE`	Tamaño w_s de la ventana activa que viaja con el robot.
`WINDOW_CENTER`	Índice del centro de la ventana.
`ALPHA`	Resolución α del histograma polar en grados.
`HIST_SIZE`	Cantidad de sectores en el histograma polar.
`THRESH`	Valor umbral T que determina la detección de un pico o valle en el histograma polar.
`WIDE_V`	Cantidad de sectores que califican como un "valle ancho" o s_{max}.
`V_MAX`	Rapidez máxima del robot en metros por segundo.
`V_MIN`	Rapidez mínima del robot en metros por segundo.
`OMEGA_MAX`	Velocidad angular máxima del robot en radianes por segundo.
`D_max2`	Cuadrado de la distancia máxima de una celda en la ventana activa según las ecuaciones (2.14) y (2.15).
`B`	Constante b de la ecuación (2.14).
`A`	Constante a de la ecuación (2.14).

El algoritmo 3.7 es una descripción de la implementación a muy alto nivel de todo el controlador VFH. Un detalle importante es que el cálculo de la dirección a partir del valle más cercano para valles anchos difiere ligeramente del planteo original, y se asemeja más bien a las direcciones candidato que se exponen en VFH+. Esto se debe a que para valles muy anchos se tendía a seleccionar direcciones muy alejadas, pues efectivamente la mitad del sector se podría encontrar completamente al lado contrario de la dirección del objetivo. Esto se daba en varias situaciones muy comunes, como cuando solo hay un obstáculo pequeño en la cercanía del robot.

Algoritmo 3.7: Algoritmo de evasión VFH implementado.

Entrada: Lectura de los sensores , postura del robot, posición del objetivo.
Salida: Acción de control v, θ
Parámetros: Ver tabla 3.3.

```
Iniciar con todas las estructuras en cero:
Calcular constantes β_{i,j} y d_{i,j} de las ecuaciones (2.13) y (2.14), para cada
    celda (i,j) de la ventana activa

Mientras no se haya alcanzado el objetivo hacer:
    Actualizar la posición actual.
    Leer los datos de los sensores.
    Para cada lectura aumentar el valor de la celda i,j correspondiente
       de obstacle_grid
    Para cada celda de la ventana activa calcular m_{i,j} según la ecuación
       (2.14).
    Actualizar el histograma polar según las ecuaciones (2.16) y (2.17).
    Actualizar el histograma polar filtrado según la ecuación (2.18)
    Determinar los valles a partir de s_max.

    Si hay un objetivo entonces:
        Calcular la dirección del ojetivo t_dir
    Si no:
        t_dir = dirección actual
    Fin si

    Obtener el valle v_dir más cercano a t_dir.
    Si v_dir es un valle angosto (> s_max) entonces:
        Calcular la nueva dirección θ_o según ecuación (2.19)
    Si no, si t_dir está dentro del valle entonces:
        θ_o = t_dir
    Si no:
        s = sector de v_dir más cercano a t_dir
        Si s es el borde izquierdo entonces:
            θ_o = alpha*(s - s_max)/2
        Si no:
            θ_o = alpha*(s + s_max)/2
        Fin si
    Fin si

    Calcular la velocidad v_o según las ecuaciones (2.20) y (2.21)
    Pasar la acción de control (v_o, θ_o) al controlador de los motores.
Fin mientras
```

Tabla 3.4: Variables miembro de la clase VFHModel.

Nombre	Descripción
`x_0`	Posición absoluta del robot sobre el eje x (en metros).
`y_0`	Posición absoluta del robot sobre el eje y (en metros).
`cita`	Orientación absoluta del robot θ respecto al eje z (en grados).
`obstacle_grid`	Arreglo bidimesional que representa la cuadrícula de certeza $\boldsymbol{C}$.
`active_window`	Arreglo bidimensional que representa la ventana activa $\boldsymbol{C}^*$.
`polar_hist`	Arreglo que representa el histograma polar $\boldsymbol{H}$.
`filt_polar_hist`	Arreglo que representa el histograma polar filtrado $\boldsymbol{H'}$.
`valleys`	Lista de valles candidatos, cada valle es un par de sectores (s_1, s_2) que indican respectivamente el inicio y el final del valle en sentido horario.
`i_0`	Índice en la cuadrícula de certeza correspondiente a `x_0`.
`j_0`	Índice en la cuadrícula de certeza correspondiente a `y_0`.
`k_0`	Sector en el histograma polar correspondiente a `cita`.
`target`	Punto (x, y) objetivo, o `None` si no se está siguiendo una trayectoria.

3.3.1. Elección de parámetros

El proceso de elección de parámetros fue bastante largo para este algoritmo. En la publicación original apenas se menciona de forma general el efecto de los parámetros, y solo hacen descripciones muy generales de algunos de los valores que fueron usados. En la investigación realizada, no se encontró ninguna publicación científica que describiera algún método para escoger los parámetros del controlador.

Así, fue necesario seguir una metodología de observación, prueba y error. De especial importancia fue el ajuste del valor de umbral para picos y valles `THRESH`. Éste se hizo observando diferentes histogramas producidos a partir de datos tanto fabricados directamente desde el ambiente de programación, como obtenidos de las simulaciones. Este algoritmo fue el que requirió más tiempo para ajustar sus parámetros antes de obtener el funcionamiento que se expone en los resultados del proyecto. Parte de esto se debe a que muchos de los parámetros no tienen un "significado o interpretación física" obvia, por lo que no hay forma de hacer un primer estimado confiable del valor que deberían tener varios de los parámetros. Por esta razón, y como se verá más adelante por el desempeño de los algoritmos, se recomienda usar VFH+ en la mayoría de casos.

A continuación algunos consejos:

- Ajustar `RESOLUTION` tal que sea del orden de magnitud del radio del robot (o más).
- `WINDOW_SIZE` se recomiendan valores no mayores a 30 (dependiendo de `RESOLUTION`), pues tener un rango de visión muy "largo" puede confundir al robot a la hora de detectar valles, especialmente si hay "interferencia" de obstáculos muy lejanos.
- Los valores de los parámetros `A`, `B`, `D_max2` y `THRESH` están íntimamente relacionados entre sí. Hay que recordar que `D_max2` se define a partir de `RESOLUTION` y `WINDOW_SIZE`, y que `A` se define

en términos de `B` y `D_max2`, es decir que solo es necesario ajustar `B` y `THRESH`.

- Ajustar `B` y `THRESH` de manera conjunta. En general un `B` mayor disminuye el efecto de las celdas más lejanas, pero aumenta los valores absolutos de densidad que se observan en el histograma polar filtrado.
- En general, los consejos de ajuste para VFH+ dados en la sección 3.4.1 para los parámetros que comparten ambos algoritmos son válidos, aunque la correlación con las características físicas del robot y los sensores no parece ser tan directa.

3.4. Algoritmo VFH+

El algoritmo VFH+ comparte muchas similitudes en su interfaz con el VFH, como es de esperar. El controlador se implementó en el módulo `VFHP.py`, mediante la clase `VFHPModel`. La tabla 3.5 indica los parámetros del módulo, mientras la tabla 3.6 indica los miembros de la clase `VFHPModel`. A pesar de tener más parámetros que el controlador VFH, estos tienen un significado físico más obvio. El algoritmo 3.8 resume la implementación a alto nivel.

La implementación se hizo en base a la teoría y las ecuaciones descritas en la sección 2.4.3, con algunas modificaciones.

La descripción del algoritmo hecha en [?] no hace mención alguna al cálculo de la velocidad. Solo se menciona que si el histograma polar mascarado se encuentra totalmente bloqueado, es necesario disminuir la velocidad. Así, se decidió implementar el control de la velocidad a partir de la misma función de costo que se usa para seleccionar la dirección, ecuación (2.39). Nótese que la máxima diferencia entre cualesquiera dos ángulos es 180°, por lo que esta función tiene un valor máximo fácil de determinar, ecuación (3.4). La velocidad de control se obtiene según la ecuación (3.5), a partir del costo c_{dir} de la dirección seleccionada.

$$c_{max} = 180^\circ \cdot (\mu_1 + \mu_2 + \mu_3) \tag{3.4}$$

$$V = V_{max} \left(1 - \frac{c_{dir}}{c_{max}}\right) + V_{min} \tag{3.5}$$

3.4.1. Elección de parámetros

La elección de parámetros se hizo mediante prueba y error, observando el comportamiento del robot durante las simulaciones. Esto debido a que no se contaba con una fuente confiable que describiera una metodología formal de cómo parametrizar el controlador. Si bien los parámetros de VFH+ parecen tener un relación más directa con elementos físicos reales del robot y su ambiente en comparación a VFH, fue necesario un tiempo considerable para ajustar los parámetros y obtener el funcionamiento que se presenta en los resultados del trabajo.

Durante este proceso, se establecieron algunas relaciones que podrían ayudar de forma general al ajuste del controlador en diferentes situaciones, los siguientes son consejos prácticos obtenidos de esta experiencia. Se recomienda seguir este orden al hacer la sintonización de parámetros.

Algoritmo 3.8: Algoritmo de evasión VFH+ implementado.

Entrada: Lectura de los sensores , postura del robot, posición del objetivo, radios de giro.
Salida: Acción de control (v, θ)
Parámetros: Ver tabla 3.5.

```
1  Iniciar con todas las estructuras en cero:
2  Calcular constantes β_{i,j}, d_{i,j} γ_{i,j} de las ecuaciones (2.13), (2.14) y (2.24),
       para cada celda (i,j) de la ventana activa
3
4  Mientras no se haya alcanzado el objetivo hacer:
5      Actualizar la posición actual.
6      Leer los datos de los sensores.
7      Para cada lectura aumentar el valor de la celda i,j correspondiente
           de obstacle_grid
8      Para cada celda de la ventana activa calcular m_{i,j} según la ecuación
           (2.14).
9      Actualizar el histograma polar según la ecuación (2.25) y (2.26).
10     Actualizar el histograma polar binario según la ecuación (2.27).
11     Actualizar el histograma polar mascarado según la ecuación (2.34) y
           el algoritmo 2.4.
12     Determinar los valles a partir de s_max.
13
14     Si hay un objetivo entonces:
15         Calcular la dirección del objetivo t_dir
16     Si no:
17         t_dir = dirección actual
18     Fin si
19
20     Determinar las direcciones candidatas, ecuaciones (2.35), (2.36),
           (2.37) y (2.38).
21     Determinar la dirección de menor costo, ecuación (2.39).
22     Calcular la velocidad según la ecuación (3.5).
23     Pasar la acción de control (v,θ) al controlador de los motores.
24 Fin mientras
```

Tabla 3.5: Parámetros de la implementación de VFH+.

Nombre	Descripción
GRID_SIZE	Tamaño n de la cuadrícula de certeza $n \times n$.
C_MAX	Valor máximo de certeza para cada celda de C.
RESOLUTION	Longitud del lado de cada celda de la cuadrícula en metros.
WINDOW_SIZE	Tamaño w_s de la ventana activa que viaja con el robot.
WINDOW_CENTER	Índice del centro de la ventana.
ALPHA	Resolución α del histograma polar en grados.
HIST_SIZE	Cantidad de sectores en el histograma polar.
D_max2	Cuadrado de la distancia máxima de una celda en la ventana activa según las ecuaciones (2.14) y (2.15).
B	Constante b de la ecuación (2.14).
A	Constante a de la ecuación (2.14).
R_ROB	Radio del robot.
D_S	Distancia mínima de obstáculo.
T_LO	Valor de umbral, límite inferior de histéresis.
T_HI	Valor de umbral, límite superior de histéresis.
WIDE_V	Cantidad de sectores que califican como un "valle ancho" o s_{max}.
V_MAX	Rapidez máxima del robot en metros por segundo.
V_MIN	Rapidez mínima del robot en metros por segundo.
mu1	Constante μ_1 de la ecuación (2.39).
mu2	Constante μ_2 de la ecuación (2.39).
mu3	Constante μ_3 de la ecuación (2.39).
MAX_COST	Costo máximo de la ecuación (2.39).

- Ajustar RESOLUTION tal que sea del orden del radio del robot. También se puede tomar en cuenta la resolución de los sensores, las celdas no deberían tener un tamaño mucho menor que la resolución de los sensores, pues podría darse una situación en la que celdas que parecen estar libres en realidad están ocupadas.

- GRID_SIZE debe ser lo suficientemente grande como para abarcar todo el espacio que se espera que el robot recorra (tomando en cuenta el valor de RESOLUTION). Este parámetro no tiene mayor impacto en el tiempo de ejecución del algoritmo.

- WINDOW_SIZE depende de la aplicación, la densidad esperada de obstáculos y la capacidad de procesamiento del robot. En general un número entre 15 y 35 suele funcionar bien. Si la densidad de obstáculos es muy alta puede ser conveniente reducir el tamaño o hacer la resolución más fina.

- C_MAX dependerá de la frecuencia de muestreo de los sensores y la confiabilidad de las mediciones. Si las mediciones son muy confiables y/o el tiempo entre mediciones es muy grande, se

Tabla 3.6: Variables miembro de la clase VFHPModel.

Nombre	Descripción
`x_0`	Posición absoluta del robot sobre el eje x (en metros).
`y_0`	Posición absoluta del robot sobre el eje y (en metros).
`cita`	Orientación absoluta del robot θ respecto al eje z (en grados).
`obstacle_grid`	Arreglo bidimesional que representa la cuadrícula de certeza $\boldsymbol{C}$.
`active_window`	Arreglo bidimensional que representa la ventana activa $\boldsymbol{C_a}$.
`polar_hist`	Arreglo que representa el histograma polar $\boldsymbol{H^p}$.
`bin_polar_hist`	Arreglo que representa el histograma polar binario $\boldsymbol{H^b}$.
`masked_polar_hist`	Arreglo que representa el histograma polar mascarado $\boldsymbol{H^m}$.
`valleys`	Lista de valles candidatos, cada valle es un par de sectores (s_1, s_2) que indican respectivamente el inicio y el final del valle en sentido horario.
`i_0`	Índice en la cuadrícula de certeza correspondiente a `x_0`.
`j_0`	Índice en la cuadrícula de certeza correspondiente a `y_0`.
`k_0`	Sector en el histograma polar correspondiente a `cita`.
`target`	Punto (x, y) objetivo, o `None` si no se está siguiendo una trayectoria.
`prev_dir`	Última dirección de control.
`prev_cost`	Costo de la última dirección de control.

puede bajar el valor de esta constante. Si por otro lado las mediciones tienen una gran varianza o incertidumbre se recomienda aumentar este valor, para compensar por posibles mediciones erróneas. Se recomiendan valores mayores a 15.

- Para la resolución angular `ALPHA` de los histogramas se recomienda un valor entre 15° y 5°, nuevamente esto depende principalmente de la resolución de los sensores, y la densidad esperada de obstáculos.

- El valor de `WIDE_V` debería definirse en términos de `HIST_SIZE`, se recomienda un valor cercano a 90° (`HIST_SIZE`/4). Valores mayores tienden a hacer que el robot mantenga una distancia mayor al rodear los obstáculos.

- Los valores de los parámetros `A`, `B`, `D_max2`, `T_LO` y `T_HI` están íntimamente relacionados entre sí. Hay que recordar que `D_max2` se define a partir de `RESOLUTION` y `WINDOW_SIZE`, y que `A` se define en términos de `B` y `D_max2`, es decir que solo es necesario parametrizar `B`, `T_LO` y `T_HI`.

- La forma más confiable de asegurar la elección de estos valores es empíricamente, graficando el histograma polar en diferentes situaciones y ajustando los valores respecto a los valles y picos observados, tomando en cuenta el obstáculo del cual resultó cada pico. Sin embargo, se puede seguir el siguiente procedimiento como punto de partida.

- En general `B` se comporta como una pendiente que relaciona inversamente la magnitud de los vectores de obstáculo de la ventana activa con la distancia al robot. Entre mayor sea el valor de `B`, menor será la magnitud en las celdas más lejanas.
- Una vez determinados `A` y `B` se puede proceder a establecer los valores de umbral con histéresis, destacando que la magnitud máxima de cada celda varía en el rango $[c^2_{max},\ \ a \cdot c^2_{max}]$ según su distancia al robot. Se recomienda un valor de `T_LO` mayor a $a \cdot c^2_{max}$ y que la diferencia entre `T_LO` y `T_HI` sea mayor a c^2_{max}.
- Finalmente, `D_S` afecta la distancia que el robot mantiene con los obstáculos, pero además tiene el efecto de suavizar el histograma polar. Se recomienda elegir un valor mayor al radio del robot, en general un valor del doble del radio del robot funciona bien.
- Las constantes de la ecuación de costo deben elegirse tal que `mu1 > mu2 + mu3`.

Capítulo 4

Simulación de los algoritmos implementados

4.1. Software de simulación V-REP

V-REP es un software de simulación de robots con un ambiente de desarrollo integrado, desarrollado por Coppelia Robotics. Se ofrece como una alternativa flexible, portable y escalable a otras plataformas de simulación similares (como Gazeebo, Open HRP y Webots). Provee al usuario de varias técnicas de programación para controlar los robots, además permite mezclarlas en una misma simulación, y permite al usuario ejecutar código de manera síncrona o asíncrona a la simulación según requiera la aplicación [**?**].

Algunas de las funciones y características de V-REP según se indican en el manual de usuario ([**?**]) son:

- Multiplataforma (Windows, MacOS, Linux).
- Soporta 6 enfoques de programación: scripts incrustados, *plugins*, *add-ons*, nodos ROS, APIs de clientes remotos o soluciones personalizadas.
- Siete lenguajes de programación: C/C++, Python, Java, Lua, Matlab, Octave o Urbi.
- Soporta cuatro diferentes motores físicos de simulación (*physics engines*): ODE, Bullet, Vortex y Newton.
- Cálculo completo de cinemática (directa e inversa).
- Detección de colisiones y obstáculos (*Meshes*, *octrees*, *point clouds*)
- Cálculo de distancia mínima entre *Meshes*, *octrees*, *point clouds*.
- Sensores de proximidad realistas.
- Registro y visualización de datos integrada.
- Navegador de modelos con función *drag and drop* y varios modelos predefinidos.

En [?] se presenta un estudio comparativo de diferentes herramientas de software que se usan para simular robots, entre ellas V-REP y otros simuladores como Gazeebo, ARGoS, Webots, OpenRave, entre otros. Este estudio incluye una encuesta realizada a investigadores y profesores en el área de robótica de varias universidades del mundo. En esta encuesta V-REP obtuvo la calificación más positiva en cuanto a documentación, soporte y tutoriales disponibles. Además, obtuvo la calificación general más alta, y el 72 % de los encuestados que usan V-REP como herramienta principal de desarrollo indicaron que tomaron esta decisión por ser la mejor herramienta después de evaluar varias alternativas.

4.1.1. Modelos de programación

Como se mencionó anteriormente V-REP permite usar hasta 6 técnicas diferentes de programación alrededor del motor de simulación.

1. *Script* embedido: escritos en Lua usando la API Regular. Este es el método "nativo" de V-REP, la mayoría de objectos del simulador tienen un script asociado. Es el método más sencillo de programación, permite modularizar fácilmente el control de un robot según la jerarquía de sus partes. Además se pueden usar para personalizar la escena e incluso la simulación en sí.

2. *Add-on*: escritos en Lua, pueden usar las funciones de la API Regular (con ciertas restricciones). Permiten extender la funcionalidad de V-REP con código escrito por el usuario. Su uso es más generico del simulador en sí, ya sea en forma de funciones o de scripts que se cargan automáticamente y corren en segundo plano.

3. *Pluggins*: en general se usan para añadir funciones, pueden hacer uso de la API Regular ya sea en Lua o C/C++.

4. API de clientes remotos: permite enlazar aplicaciones externas directamente con V-REP mediante el uso de comandos de la API Remota, la cual está disponible en C/C++, Python, Java, Matlab, Octave, Lua y Urbi.

5. Nodo ROS: permite enlazar aplicaciones externas mediante el uso de la API de ROS.

6. Un cliente/servidor personalizados: V-REP permite hacer uso de *sockets* y *pipes*, esto en teoría permite enlazar con cualquier aplicación y cualquier lenguaje de programación, sin embargo es por mucho el método más complicado, en especial porque se debe implementar la forma de comunicación personalizada.

Para el desarrollo de este proyecto se hizo uso de la API Remota de Python y *scripts* embedidos. La Figura 4.1 muestra como los diferentes componentes del modelo de programación de V-REP se enlazan entre sí.

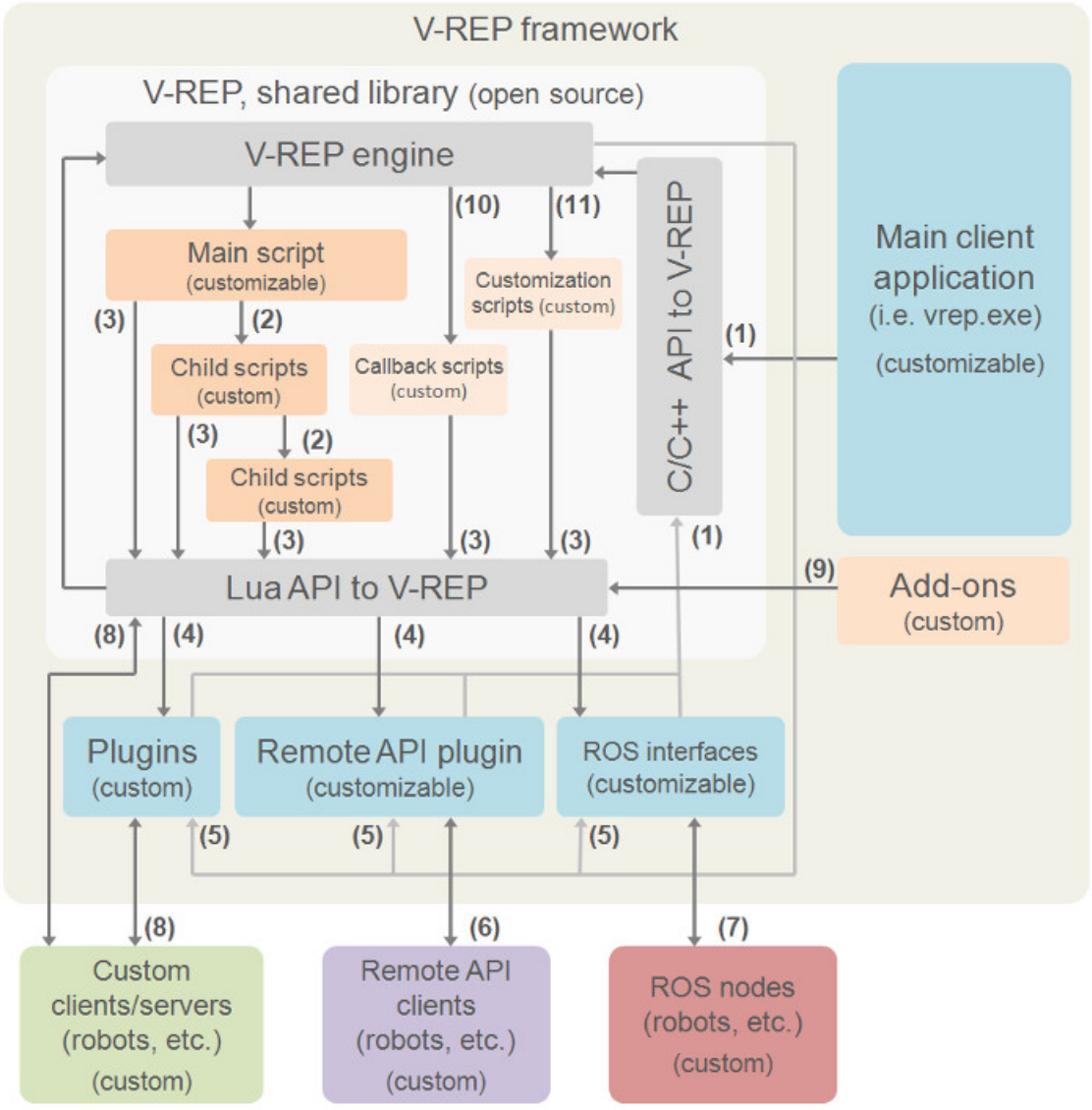
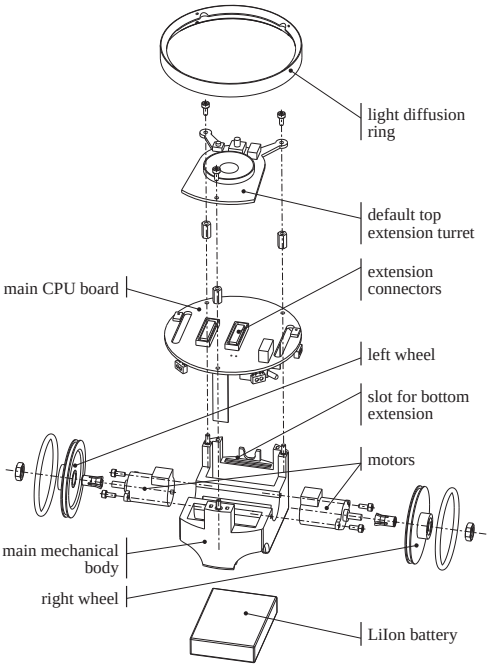


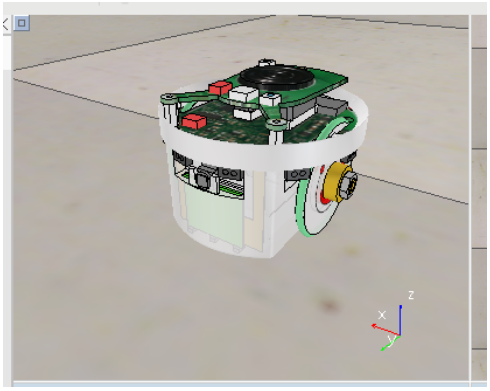
Figura 4.1: Infraestructura de V-REP, tomado de [?].

4.2. El robot e-Puck

Se usó el modelo del robot e-Puck que viene incluido en el simulador V-REP. Este es un robot diferencial de uso educacional, para más información se puede consultar el sitio web http://www.e-puck.org/. El robot tiene un diámetro de 75 mm, las ruedas tienen un diámetro de 41 mm y una separación de 53 mm [?]. La Figura 4.2 muestra un diagrama del robot y su apariencia en el simulador.



(a) Diagrama mecánico del e-Puck, tomado de [?].



(b) Modelo del e-Puck en V-REP.

Figura 4.2: El robot diferencial e-Puck.

El e-Puck incluye varios sensores de ultrasonido para detección de obstáculos. Sin embargo se decidió agregar un sensor Hokuyo sobre la parte superior del robot. El Hokuyo es un sensor Lidar con un funcionamiento como el que se describe en la sección 2.3.2. El modelo del Hokuyo es parte de los modelos que incluye V-REP, además se hicieron algunas modificaciones al *script* de control.

Es necesario aclarar que el Hokuyo no se puede conectar físicamente a un e-Puck en la vida real. Sin embargo, como el propósito de usar el e-Puck es simplemente validar el funcionamiento de los algoritmos, esto se puede obviar para motivos de este trabajo. Una implementación real se podría hacer en alguna otra plataforma robótica que sí permita usar el Hokuyo. En este caso el sensor tiene mayor relevancia que el robot en sí, pues se supone que los algoritmos son generalizables a otros modelos de robots móviles.

4.3. Integración del simulador con los algoritmos implementados

La figura 4.3 muestra la jerarquía del sistema completo usado en las simulaciones. El *script* principal del robot está asociado a la base del modelo del e-Puck. Este script lo que hace es obtener *Object Handles* para las partes del robot, llamar al programa externo de Python `HokuyoRob.py` y pasarle los *handles* como argumento. Además, el modelo del Hokuyo se incluye como un objeto hijo del e-Puck. Este contiene su propio *script*, que hace un barrido sobre un sector angular de 240° y guarda los puntos detectados en un arreglo. Este arreglo está disponible al *script* principal (y por ende al programa externo) en forma de una señal. Los dos scripts son del tipo 1 mencionado anteriormente en la sección 4.1.1.

El programa externo `HokuyoRob.py` es un script de Python que se encarga de enlazar V-REP con los controladores descritos en el capítulo 3. Hace uso de la API externa de V-REP, es decir es del tipo 4 mencionado en la sección 4.1.1. El programa hace un control de entrada de datos para verificar que se hizo una conexión exitosa con el simulador y que cuenta con todos los *Object Handles* necesarios. Posteriormente inicializa variables y crea una instancia de la clase `DiffRobot` que modela el robot e-Puck. Luego el programa entra en el lazo princpial de simulación. En este se lee el sensor Hokuyo y la posición del Robot, cada vez que hay datos nuevos estos se suministran a la clase `DiffRobot`. A su vez, esta clase contiene una instancia de los diferentes controladores de evasión de obstáculos. Internamente se ejecuta el control respectivo, se obtiene la velocidad requerida de los motores y se comunica a V-REP a través de las funciones de la API externa. La documentación completa de la API de Python se encuentra disponible en el sitio web de V-REP http://www.coppeliarobotics.com/helpFiles/en/remoteApiFunctionsPython.htm.

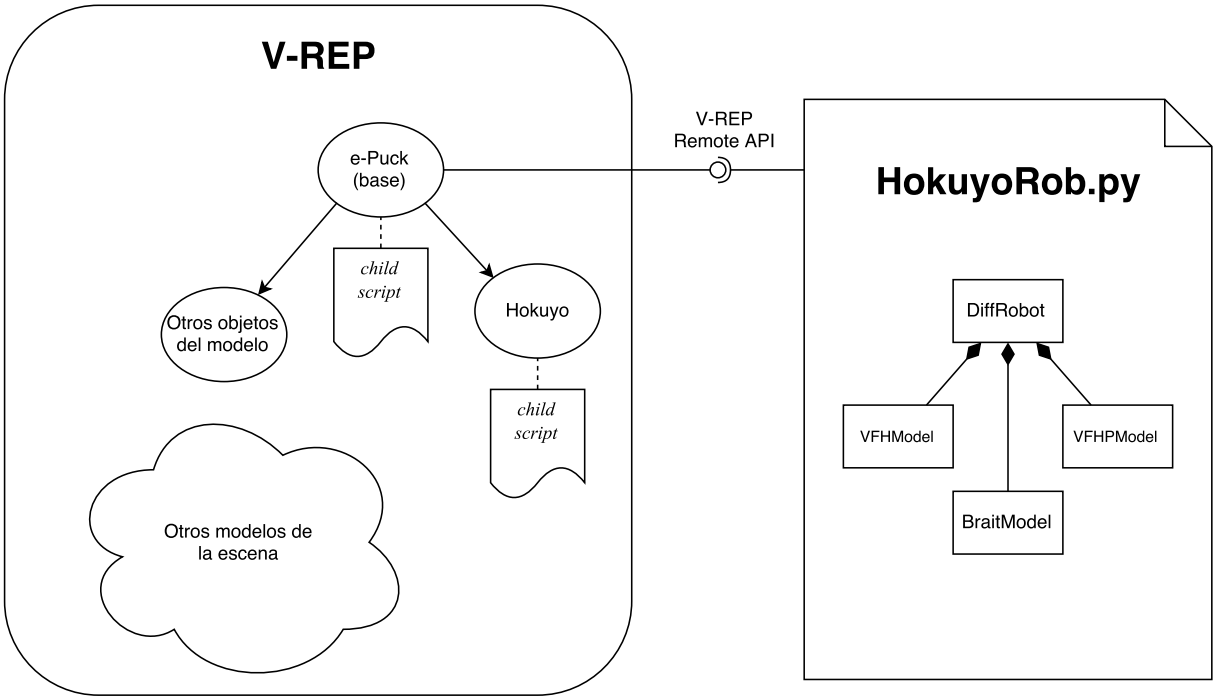


Figura 4.3: Estructura del sistema de simulado.

4.4. Cursos de obstáculos

Para validar los algoritmos se definieron tres entornos de prueba. Estos son:

1. Un obstáculo sencillo frente al robot.
2. Tres obstáculos sencillos frente al robot y ligeramente a ambos lados.
3. Obstáculos complejos en forma de "pasillo".

La figura 4.4 muestra una vista aérea de los tres cursos. Además se hicieron dos pruebas por algoritmo, una prueba con seguimiento de un objetivo y otra sin seguimiento (movimiento "libre"). Se tomaron datos de la trayectoria seguida por el robot, la distancia recorrida y el tiempo en alcanzar el objetivo, para cada algoritmo, se usó un paso de integración de **25 ms**. Los parámetros usados en las pruebas se indican en el apéndice B. A continuación se presentan algunos resultados representativos.

4.5. Resultados Obtenidos

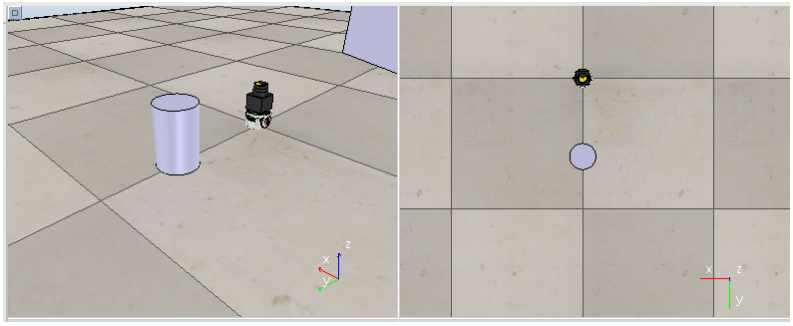
En esta sección se presentan resultados para las pruebas con seguimiento de trayectorias. Estas se prestan muy bien para comparar el desempeño de los algoritmos bajo los mismos requisitos (llegar de un punto A a un punto B). En las figuras que se encuentran más adelante, se indica el punto de partida en verde y la ubicación final del robot en rojo. Al robot se le ordena detenerse una vez se encuentre a una distancia menor a 30 mm del punto objetivo (recordar que el robot tiene un diámetro de 75 mm).

4.5.1. Pista 1

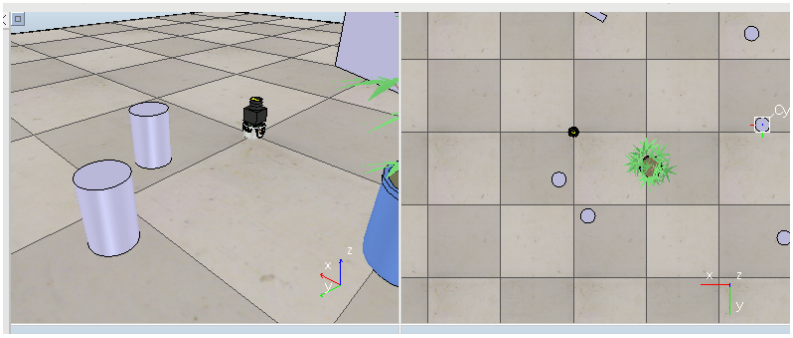
La figura 4.5a muestra la trayectoria seguida por el robot según el algoritmo usado. Por otro lado, la figura 4.5b gráfica la distancia al objetivo respecto al tiempo. Nótese que tanto para VFH como VFH+, el robot mantuvo una mayor distancia al obstáculo. Ambos algoritmos tomaron una ruta semicircular alrededor del cilindro que bloqueaba el camino del robot. Por otro lado, el algoritmo de Braitenberg tomó un camino apenas ligeramente curvo. En general el algoritmo de Braitenberg tiende a resultar en trayectorias más suaves.

En las primeras pruebas realizadas, el algoritmo de Braitenberg tuvo su peor comportamiento en esta pista, incluso llegando a rozar el obstáculo. Esto se puede explicar por el ambiente de simulación: el hecho de que el obstáculo se encuentra casi perfectamente alineado con el robot, además de tener un perímetro perfectamente circular. Por ésto y la forma en que se calculan los estímulos, estos tienden a tener valores casi idénticos, por lo que la diferencia entre ambos lados no es lo suficientemente significativa para hacer virar al robot hasta que este se encuentra casi tocando el cilindro. Sin embargo, al cambiar el modo de operación del sensor a `SMODE_FULL`, el desempeño mejoró hasta el mostrado en la figura 4.5b, superando a VFH y llegando a ser casi tan bueno como el VFH+.

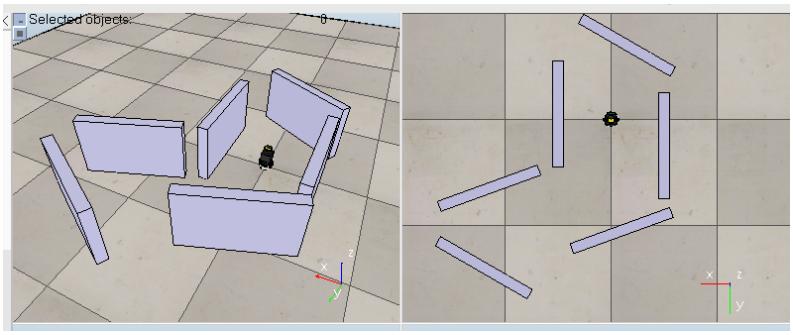
En esta pista se evidenció una de las mejoras de VFH+ respecto a VFH. Al iniciar la simulación, el Algoritmo VFH parece oscillar ligeramente entre evadir el obstáculo por la derecha o la izquierda. Al hacer esto el robot disminuye su velocidad considerablemente (pues aún tiene el obstáculo directamente



(a) Primera pista.

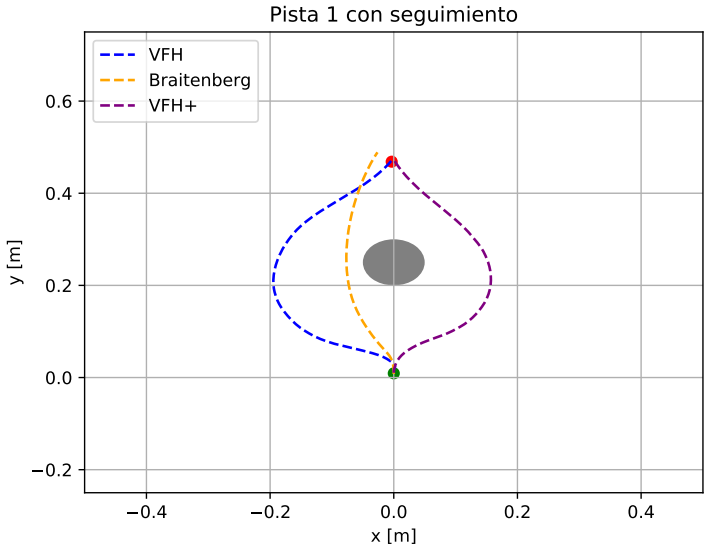


(b) Segunda pista.

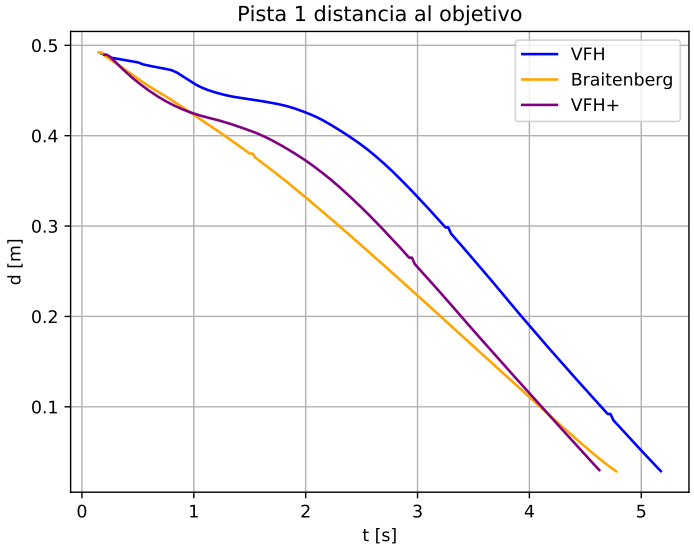


(c) Tercera pista.

Figura 4.4: Pistas de obstáculos usadas en la simulación.



(a) Trayectoria del robot, objetivo = (0, 0.5). Círculo verde inicio, rojo objetivo.



(b) Distancia al objetivo respecto al tiempo.

Figura 4.5: Pista 1, resultados de simulación.

en frente), todo esto resulta en que le tome un tiempo empezar a acercarse al objetivo, como se ve alrededor de $t = 1$ s en la figura 4.5b. Eventualmente el robot se decide por ir hacia la izquierda, y aumenta su velocidad. Esta "indecisión" a la hora de determinar la dirección del movimiento no se presenta en VFH+, gracias al uso de la función de costo. Recordando lo mencionado en la sección 2.4.3, hay un costo asociado a cambios respecto a la dirección seleccionada anteriormente. Esto hace que en situaciones como esta, en que hay dos direcciones que son casi igual de buenas, el robot no oscile entre las dos posibilidades y se "comprometa" a seguir una única dirección.

4.5.2. Pista 2

La figura 4.6a muestra la trayectoria seguida por el robot, mientras la figura 4.6b muestra la distancia al objetivo. Nuevamente el algoritmo VFH+ tiene el mejor desempeño. Nótese además que el algoritmo VFH resulta en la trayectoria más segmentada y menos suave.

Además se evidencia como VFH+ tiene un mejor comportamiento respecto a obstáculos que se encuentran a mayor distancia. Es claro que VFH tarda más que VFH+ en notar el espacio libre entre los dos obstáculos más cercanos al objetivo. Esto se puede deber al efecto del ángulo de ensanchamiento en VFH+, que se reduce para celdas más alejadas del robot y cumple el rol del filtro que se aplica al histograma polar en el caso de VFH (filtro que no distingue cercanía de las celdas). VFH+ rápidamente mantiene una dirección circundante al obstáculo que se encuentra más cerca del robot.

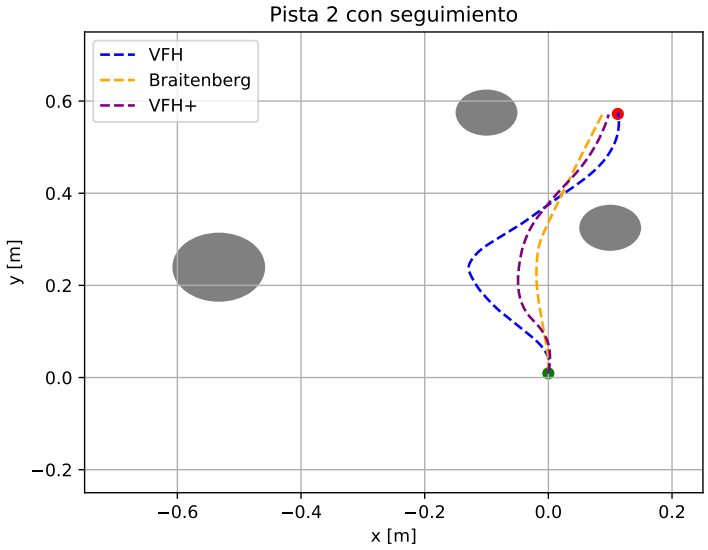
En cuanto al algoritmo de Braitenberg, se puede observar que nuevamente el robot sigue una trayectoria bastante directa al objetivo, con apenas unas pequeñas variaciones en la dirección del robot, y de hecho la trayectoria es bastante similar a la del algoritmo VFH+. El algoritmo de Braitenberg alcanza el objetivo en menos tiempo que el VFH, y pasa considerablemente más cerca de los obstáculos.

4.5.3. Pista 3

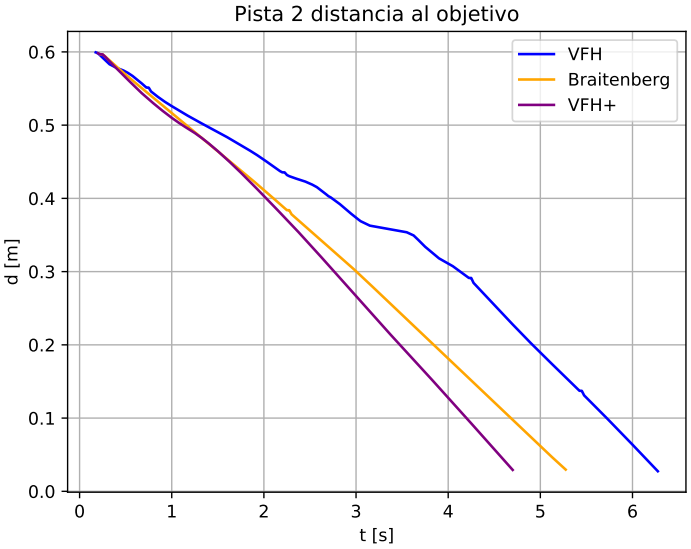
Esta es la pista con mayor densidad de obstáculos. En esta pista nuevamente VFH+ logró el menor tiempo de los tres algoritmos, y el algoritmo VFH logró llegar al objetivo más rápido que Braitenberg. Sin embargo, esta fue la pista en la que el algoritmo VFH tuvo mayores problemas, el robot se mantuvo peligrosamente cerca de una de las paredes. Durante el desarrollo de la investigación, esta pista evidenció la fragilidad del algoritmo VFH original, pues con tan solo aumentar ligeramente el tamaño de la ventana activa o cambiar en menos de un 10 % el valor de umbral de los valles, el algoritmo resulta en una ruta cíclica que no converge y queda encerrado dentro los obstáculos, incapaz de notar el pequeño espacio libre hacia el objetivo.

El problema de ajuste de parámetros no se dio con Braitenberg, siendo el caso que el robot llegó correctamente al objetivo desde la primer iteración de las pruebas. Además, modificar los parámetros tuvo poco efecto en la trayectoria general del robot, mas sí lo tuvo en el tiempo que tardó en llegar al objetivo. Al igual que en las pruebas anteriores, el algoritmo de Braitenberg resultó en una trayectoria poco curva. Una ventaja que quedó clara del algoritmo de Braitenberg, es que el robot parece responder mejor a la diferencia en distancia entre obstáculos en comparación al algoritmo VFH.

Por otro lado, el algoritmo VFH+ mostró un muy buen desempeño, manteniendo una velocidad alta a lo largo del trayecto y al mismo tiempo una distancia prudencial a los obstáculos en el camino.

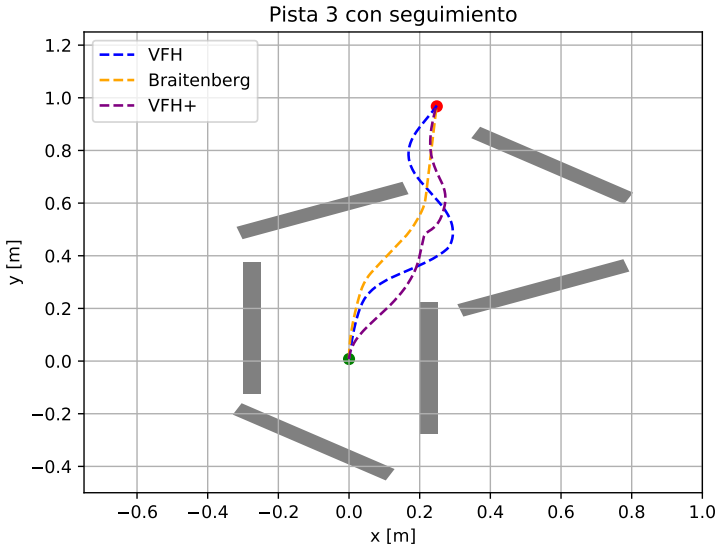


(a) Trayectoria del robot, objetivo = (0.1, 0.6). Círculo verde inicio, rojo objetivo.

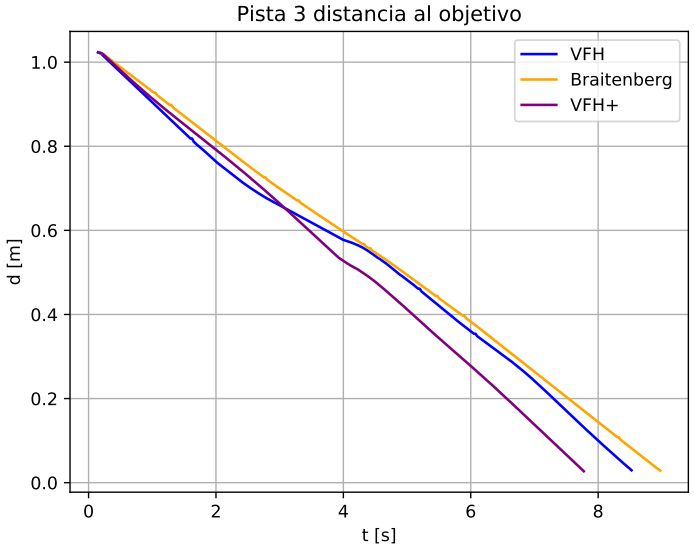


(b) Distancia al objetivo respecto al tiempo.

Figura 4.6: Pista 2, resultados de simulación.



(a) Trayectoria del robot, objetivo = (0.25, 1.0). Círculo verde inicio, rojo objetivo.

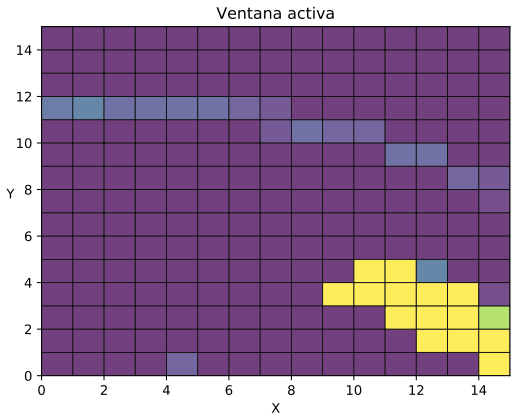


(b) Distancia al objetivo respecto al tiempo.

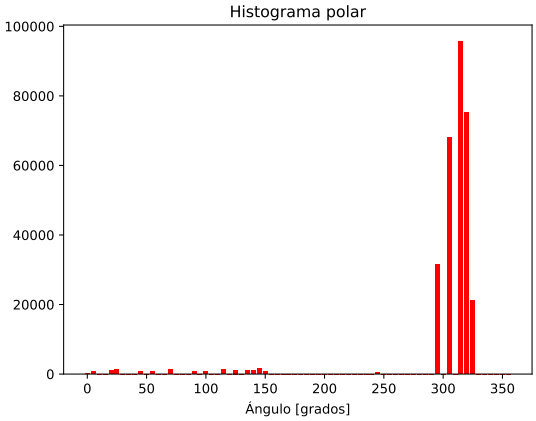
Figura 4.7: Pista 3, resultados de simulación.

También, a pesar de que inicialmente la línea de visión del robot tiene obstáculos en casi todos los sectores en la dirección general del objetivo, el algoritmo no tuvo mayor problema en dirigirse hacia la dirección en la que no había obstáculos realmente cercanos al robot. VFH+ definitivamente mejora muchas de las carencias del algoritmo VFH original en este aspecto, además de que no requiere un ajuste tan minucioso de parámetros para lograr un desempeño consistentemente mejor al de un algoritmo más sencillo de implementar como el de Braitenberg.

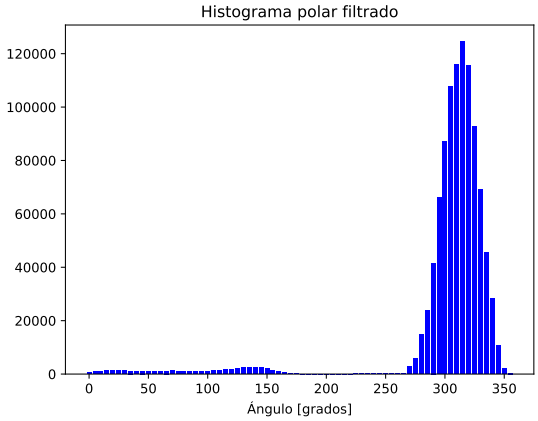
La figura 4.8 y la figura 4.9 muestran diagramas de los histogramas internos de los algoritmos VFH y VFH+ respectivamente. Nótese la forma en que ambos construyen una representación imperfecta de los obstáculos a su alrededor, sin embargo a pesar de esto logran la evasión. Una cosa que podrá notarse de estas imágenes es que en el algoritmo VFH+ se usó una ventana activa con más celdasa, aunque también con un tamaño de celda ligeramente menor. Se podría pensar que VFH se beneficiaría de tener una ventana más grande, sin embargo este no es el caso. Esta es una diferencia clave en el desemepeño de ambos algoritmos, pues por la forma de representar los obstáculos, VFH tiende a degradar su desempeño con ventanas muy grandes. En cambio, VFH+ suele manejar bien vetanas grandes, en especial por la forma en que las celdas mas alejadas tienden a contribuir menos a la magnitud del vector de obstáculos en comparación a VFH.



(a) Ventana activa VFH.

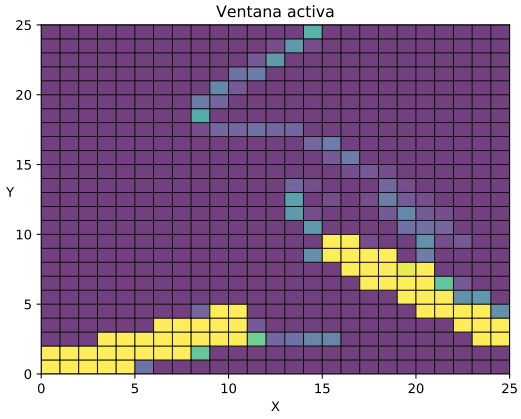


(b) Histograma polar VFH.

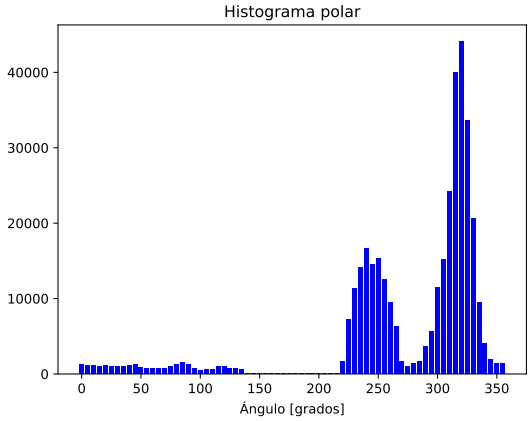


(c) Histograma polar filtrado VFH.

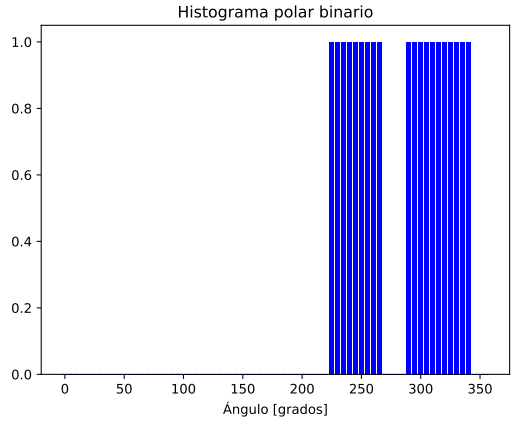
Figura 4.8: Pista 3, estado final del algoritmo VFH.



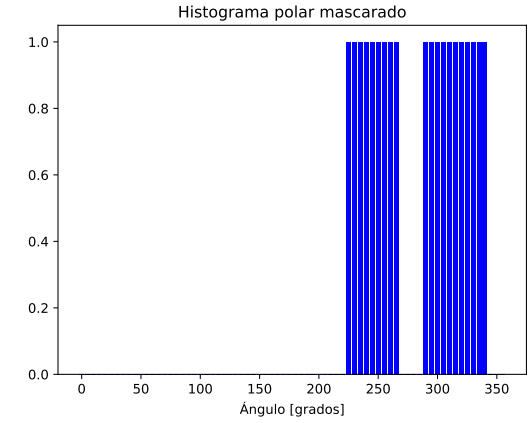
(a) Ventana activa VFH+.



(b) Histograma polar VFH+.



(c) Histograma polar binario VFH+.



(d) Histograma polar mascarado VFH+.

Figura 4.9: Pista 3, estado final del algoritmo VFH+.

4.5.4. Análisis comparativo

Tabla 4.1: Comparación del desempeño de los algoritmos. El error se calculó con base en la distancia al objetivo.

Pista	Algoritmo	recorrido total [m]	tiempo [s]	IAE	ITAE
1	VFH	0.64	5.17	1.61	3.32
1	Brait	0.50	4.78	1.27	2.27
1	VFH+	0.58	4.62	1.31	2.43
2	VFH	0.70	6.28	2.11	5.14
2	Brait	0.58	5.28	1.63	3.26
2	VFH+	0.61	4.70	1.49	2.69
3	VFH	1.14	8.53	4.45	13.84
3	Brait	1.01	8.97	4.65	15.07
3	VFH+	1.05	7.78	4.09	11.44

La tabla 4.1 resume los resultados obtenidos para cada pista y algoritmo, en cuanto a desempeño. Se calculó la IAE e ITAE con base en la distancia al objetivo mostrada en las gráficas anteriores. Además se indica el tiempo total que le tomó a cada algoritmo alcanzar su objetivo y la distancia total recorrida.

En general se puede decir que VFH+ es el más eficiente de los tres algoritmos, y el robot tiende a llegar al objetivo más rápido. El algoritmo VFH también tiene un buen desempeño en la evasión, sin embargo para su funcionamiento correcto se requiere un gran trabajo previo de ajuste de parámetros. Comparativamente, el algoritmo de Braitenberg es mucho más sencillo y robusto en la elección de parámetros para obtener un comportamiento funcional.

Una cosa que llama inmediatamente la atención es que la diferencia en complejidad de implementación entre Braitenberg y ambos algoritmos VFH y VFH+, no se refleja de forma tan marcada en el desempeño. Cabe destacar que en etapas tempranas del desarrollo del proyecto esto no era así y el desempeño de Braitenberg era mucho peor, llegando a durar hasta el doble que VFH en alcanzar el objetivo. Varios cambios en la implementación mejoraron mucho el desempeño, puntualmente el modo de sensado `SMODE_FULL` descrito en la sección 3.2 aumentó notablemente la habilidad del robot de evadir rápidamente los obstáculos.

En cuanto a la elección de parámetros que se hizo en las simulaciones, hay espacio para mejorar la eficiencia del algoritmo de Braitenberg, en particular en cuanto a la acción conjunta de evasión y seguimiento. Particularmente, se podría pensar en una función de estimulo para el seguimiento que fuera completamente diferente. En las simulaciones se puede notar que el robot parece pasar ligeramente de largo los puntos objetivos, en especial si no se encuentran directamente frente a él. Incluso para el algoritmo VFH, un análisis matemático más formal del efecto de los parámetros podría mejorar aún más el desempeño y facilitar la elección de parámetros. El algoritmo VFH tiene muchos más parámetros ajustables al de Braitenberg. Además, una ventaja marcada de VFH+ respecto a VFH, es que los parámetros del primero tienen un significado físico mucho más explícito, por lo que la elección de

parámetros se puede hacer "a ojo" sin tener que correr demasiadas pruebas.

Las fortalezas del algoritmo de Braitenberg se encuentran en su sencillez. Se desempeña mejor en situaciones en las que hay pocos obstáculos directamente frente al robot, en especial funciona muy bien para navegar en espacios como pasillos. Se puede argumentar de los resultados de las simulaciones que es tan eficaz en la evasión como el mucho más complejo VFH en la mayoría de situaciones.

Una desventaja considerable es que se puede ver afectado por la configuración y geometría de los obstáculos. Como la evasión depende de la diferencia entre las mediciones a la izquierda y derecha, si esta diferencia alcanza un mínimo local dentro de un obstáculo el robot podría quedar encerrado. El ejemplo más sencillo es que el robot se dirija hacia una esquina o algún otro obstáculo con forma de "embudo".

El algoritmo VFH requiere más trabajo para ser funcional, pero compensa esto con una mayor flexibilidad. Se desempeña especialmente bien en la evasión de obstáculos que bloquean directamente el camino del robot, es robusto a la geometría de los obstáculos y es capaz de evadir obstáculos pequeños eficientemente. El algoritmo VFH+ además demuestra una mayor velocidad en la evasión, y mejora considerablemente la fragilidad del algoritmo VFH original.

El algoritmo VFH empieza a mostrar carencias en ambientes muy congestionados. Especialmente cuando se vuelve relevante una buena elección del valor de umbral T y las constantes a y b de la ecuación (2.14) para determinar qué considera el robot como un valle, y como se ve afectado por la cercanía de los obstáculos.

Capítulo 5

Conclusiones y recomendaciones

5.1. Conclusiones

1. Se realizó una investigación bibliográfica que proporcionó un marco teórico adecuado sobre robots móviles autónomos para el este trabajo de investigación. Se describieron 3 algoritmos de evasión de obstáculos y se proporcionó una base teórica para su entendimiento.

2. Se hizo una descripción en pseudocódigo de la implementación del algoritmo de Braitenberg y VFH, para facilitar su compresión. La implementación en Python se hizo en base a este pseudocódigo.

3. Se implementaron y probaron con éxito los algoritmos de Braitenberg, VFH y VFH+.

4. Se logró usar con éxito la herramienta de simulación V-REP para modelar una situación de evasión de obstáculos a la que se puede enfrentar un robot real. Se logró simular de manera realista la obtención de datos a partir de un sensor tipo Lidar.

5. A partir del uso de la herramienta de simulación, se logró validar la implementación de los algoritmos en 3 situaciones diferentes.

6. Se hizo un estudio comparativo de las fortalezas y debilidades de cada algoritmo a partir de los resultados de la simulación.

7. El código se documentó ampliamente, y se hizo un manual de usuario para facilitar el uso de los módulos en proyectos posteriores.

8. El algoritmo VFH+ tuvo consistentemente el mejor desempeño.

9. El algoritmo VFH fue el más difícil de parametrizar, mientras que el de Braitenberg fue el más fácil.

10. El algoritmo de Braitenberg y el VFH original tuvieron un desempeño similar, pero siendo el primero mucho más sencillo tanto en implementación como en parametrización.

11. Para el algoritmo de Braitenberg con seguimiento de trayectorias se recomienda usar el modo mixto 2b3a (2b para evasión y 3a para seguimiento). Además el modo de sensado `SMODE_FULL` dio los mejores resultados en la detección y evasión de obstáculos.

5.2. Recomendaciones

Teniendo los algoritmos implementados, sería relevante hacer un estudio más profundo de la sintonización de los parámetros. Hay muchas combinaciones que se podría probar, debido a que el enfoque de este proyecto estaba en la implementación no se exploraron muchas posibilidades en la elección de parámetros. Hay además varias pequeñas modificaciones que podrían mejorar considerablemente el funcionamiento de los algoritmos en la práctica. Por ejemplo, para los algoritmos mixtos de Braitenberg, se podría determinar formalmente una función de cercanía para cada lado que proporcione un comportamiento de seguimiento más robusto.

Además, sería de gran interés implementar los sistemas aquí propuestos en un robot real. Así se podría observar por ejemplo qué tanto puede degradarse el desempeño al considerar el ruido de las mediciones, la variabilidad de los sensores, el efecto de tener que procesar los datos en tiempo real, etc.

El cuanto al código implementado, se podría pensar en hacer una clase que encapsule los algoritmos y provea una interfaz unificada a un usuario. Además algunos de los parámetros de módulo se podrían redefinir como parámetros de clase que se pasan mediante el constructor, de modo que se puedan instanciar diferentes controladores con parámetros ajustados a diferentes situaciones o incluso a diferentes robots dentro de un mismo programa, esto actualmente no es posible para los controladores VFH y VFH+. Esto sin embargo añadiría cierta complejidad al uso de los módulos, también se podría pensar en separar la configuración de parámetros para no inflar el constructor de ,argumentos, por medio de la lectura de un archivo de configuración.

Otro aspecto que sería de bastante interés es el desempeño de los algoritmos en la evasión de obstáculos móviles, cosa que sería muy importante para que un robot pueda desempeñar sus funciones en presencia de otros agentes, ya sea humanos, animales o bien otros robots. Por la forma en que se implementan los algoritmos VFH y VFH+, se podría pensar que alguna modificación es necesaria para poder evadir obstáculos en movimiento, de modo que los valores de certeza disminuyan con el tiempo, o bien cuando una celda se encuentre fuera de la ventana activa, etc.

Apéndice A

Código

A.1. Implementación del algoritmo Braitenberg

```
#! python
# coding=utf8

r"""
.. ::module Braitenberg

Este modulo define el controlador para evasión de obstáculos
por medio de vehículos de Braitenberg.

Note
----
Como parte del módulo se definen las siguientes constantes
relevantes al modo de funcionamiento del controlador:

    * `SMODE_MIN`
    * `SMODE_AVG`
    * `SMODE_FULL`
"""
import math
import numpy as np


SMODE_MIN = 0
SMODE_AVG = 1
SMODE_FULL = 2

class BraitModel(object):
    r"""Clase para el modelo de vehículo de Braitenberg

    Parameters
    ----------
    s_mode : {SMODE_MIN, SMODE_AVG, SMODE_FULL}, opcional
        Modo de sensado, define la forma de obtener los estímulos a partir
        de las lecturas de los sensores
    d_min : float, opcional
        Distancia mínima de detección de obstáculos.
    d_max : float, opcional
        Distancia máxima de detección de obstáculos.
    v_min : float, opcional
```

```
40          Velocidad lineal mínima del robot.
41      v_max : float, opcional
42          Velocidad lineal máxima del robot.
43      w_max : float, opcional
44          Velocidad angular máxima del robot (se asume que es la misma en ambas direcciones
                ).
45      alpha : float, opcional
46          Rango angular de visión del robot, dado en radianes.
47
48      Attributes
49      ----------
50      x : float
51          Posición del robot sobre el eje :math:`x`.
52      y : float
53          Posición del robot sobre el eje :math:`y`.
54      gamma : float
55          Orientación del robot resepecto el eje :math:`z`.
56      radius : float
57          Radio del robot.
58      sensor_l : float
59          Estímulo de evasión del lado izquierdo.
60      sensor_r : float
61          Estímulo de evasión del lado derecho.
62      target : tuple
63          Punto :math:`\left(x,\:y\right)` objetivo o ``None`` si
64          no se está siguiendo una trayectoria.
65      s_mode : {SMODE_MIN, SMODE_AVG, SMODE_FULL},
66          Modo de sensado, define la forma de obtener los estímulos a partir
67          de las lecturas de los sensores
68      D_MIN : float,
69          Distancia mínima de detección de obstáculos.
70      D_MAX : float,
71          Distancia máxima de detección de obstáculos.
72      V_MIN : float,
73          Velocidad lineal mínima del robot.
74      V_MAX : float,
75          Velocidad lineal máxima del robot.
76      W_MAX : float,
77          Velocidad angular máxima del robot (se asume que es la misma en ambas direcciones
                ).
78      ALPHA : float,
79          Rango angular de visión del robot, dado en radianes.
80
81      Examples
82      --------
83
84      >>> import Braitenberg as Brait
85      >>> robot = Brait.BraitModel(s_mode=SMODE_FULL)
86      >>> pseudo_readings = np.float_([[0.41, np.radians(x)] for x in range(-5,90,1)])
87      >>> robot.UpdateSensors(pseudo_readings)
88      >>> robot.UpdatePos(0.0, 0.0, np.pi/2)
89      >>> robot.SetTarget(0.0, 0.20)
90      >>> v, w = robot.Mixed2b3a()
91      """
92
93      def __init__(self, s_mode=0, d_min=0.01, d_max=0.25, v_min=-0.3, v_max=0.3, w_max
            =0.628, alpha = np.pi/6.0):
94
95          self.x = 0.
```

```
96          self.y = 0.
97          self.gamma = 0.
98          self.radius = 0.035
99
100         self.alpha = alpha
101         # Distancia minima y maxima de vision
102         # del algoritmo
103         self.D_MIN = d_min
104         self.D_MAX = d_max
105
106         self.V_MIN = v_min
107         self.V_MAX = v_max
108         self.W_MAX = w_max
109
110         self.sensor_l = d_max
111         self.sensor_r = d_max
112         self.s_mode = s_mode
113
114         self.target = None
115
116     def Evade2b(self, d_left=None, d_right=None):
117         r"""Obtiene la respuesta de evasión usando el algoritmo del vehículo 2b.
118
119         La respuesta se da en forma de velocidad lineal y
120         angular en los rangos definidos por :attr:`V_MIN`,
121         :attr:`V_MAX` y :attr:`W_MAX`.
122
123         Parameters
124         ----------
125         d_left : float, opcional
126             Estímulo de evasión del lado izquierdo, por defecto
127             toma el valor de :attr:`sensor_l`.
128         d_right : float, opcional
129             Estímulo de evasión del lado derecho, por defecto
130             toma el valor de attr:`sensor_r`.
131
132         Returns
133         -------
134         v_rob : float
135             Velocidad lineal de respuesta.
136         w_rob : float
137             Velocidad angular de respuesta (rad/s).
138         """
139
140         # Defaults to the object instance's sensor stimuli
141         # this behaviour can be overridden by providing the
142         # distances values as parameters.
143         if d_left == None:
144             d_left = self.sensor_l
145         if d_right == None:
146             d_right = self.sensor_r
147
148         v_left = MapStimulus(d_right, self.D_MIN, self.D_MAX, 0.0, 1.0)
149         v_right = MapStimulus(d_left, self.D_MIN, self.D_MAX, 0.0, 1.0)
150
151         #print "v_left = %.3f" % v_left
152         #print "v_right = %.3f" % v_right
153
154         v_norm = (v_right + v_left)/2.0
```

```
155         w_norm = (v_right - v_left)/2.0
156
157         #print "v_norm = %.3f" % v_norm
158         #print "w_norm = %.3f" % w_norm
159
160         v_rob = MapStimulus(v_norm, 0.0, 1.0, self.V_MIN, self.V_MAX)
161         w_rob = MapStimulus(w_norm, -0.5, 0.5, -self.W_MAX, self.W_MAX)
162
163         #return v_left, v_right
164         return v_rob, w_rob
165
166     def Evade3a(self, d_left=None, d_right=None):
167         r"""Obtiene la respuesta de evasión usando el algoritmo del vehículo 3a.
168
169         La respuesta se da en forma de velocidad lineal y
170         angular en los rangos definidos por :attr:`V_MIN`,
171         :attr:`V_MAX` y :attr:`W_MAX`.
172
173         Parameters
174         ----------
175         d_left : float, opcional
176             Estímulo de evasión del lado izquierdo, por defecto
177             toma el valor de :attr:`sensor_l`.
178         d_right : float, opcional
179             Estímulo de evasión del lado derecho, por defecto
180             toma el valor de :attr:`sensor_r`.
181
182         Returns
183         -------
184         v_rob : float
185             Velocidad lineal de respuesta.
186         w_rob : float
187             Velocidad angular de respuesta (rad/s).
188         """
189
190         # Defaults to the object instance's sensor stimuli
191         # this behaviour can be overriden by providing the
192         # distances values as parameters.
193         if d_left == None:
194             d_left = self.sensor_l
195         if d_right == None:
196             d_right = self.sensor_r
197
198         v_left = MapStimulus(d_left, self.D_MIN, self.D_MAX, 1.0, 0.0)
199         v_right = MapStimulus(d_right, self.D_MIN, self.D_MAX, 1.0, 0.0)
200
201         v_norm = (v_right + v_left)/2.0
202         w_norm = (v_right - v_left)/2.0
203
204         v_rob = MapStimulus(v_norm, 0.0, 1.0, self.V_MIN, self.V_MAX)
205         w_rob = MapStimulus(w_norm, -0.5, 0.5, -self.W_MAX, self.W_MAX)
206
207         #return v_left, v_right
208         return v_rob, w_rob
209
210     def Mixed2b3a(self, d_left=None, d_right=None):
211         r"""Obtiene la respuesta conjunta de evasión y
212         seguimiento, usando el algoritmo 2b para evasión y el
213         3a para seguimiento.
```

```
214
215         La respuesta se da en forma de velocidad lineal y
216         angular en los rangos definidos por :attr:`vmin`,
217         :attr:`V_MAX` y :attr:`W_MAX`.
218
219         Parameters
220         ----------
221         d_left : float, opcional
222             Estímulo de evasión del lado izquierdo, por defecto
223             toma el valor de :attr:`sensor_l`.
224         d_right : float, opcional
225             Estímulo de evasión del lado derecho, por defecto
226             toma el valor de :attr:`sensor_r`.
227
228         Returns
229         -------
230         v_rob : float
231             Velocidad lineal de respuesta.
232         w_rob : float
233             Velocidad angular de respuesta (rad/s).
234
235
236         .. warning::
237             Debe haberse definido un punto objetivo antes de
238             llamar a esta función.
239
240         """
241
242         ### Target following logic
243
244         left_x  = self.x + np.cos(self.gamma+np.pi/2)*self.radius
245         left_y  = self.y + np.sin(self.gamma+np.pi/2)*self.radius
246         right_x = self.x + np.cos(self.gamma-np.pi/2)*self.radius
247         right_y = self.y + np.sin(self.gamma-np.pi/2)*self.radius
248
249         print "left is (%.3f, %.3f)" % (left_x, left_y)
250         print "right is (%.3f, %.3f)" % (right_x, right_y)
251
252         left_d2  = np.sqrt(np.square(self.target[0]-left_x)  + np.square(self.target[1]-
                left_y))
253         right_d2 = np.sqrt(np.square(self.target[0]-right_x) + np.square(self.target[1]-
                right_y))
254         print "left = %.2f, right = %2f" % (left_d2, right_d2)
255
256         thresh = (self.radius*2)
257         if left_d2 > thresh  or right_d2 > thresh:
258             d_extra = max(left_d2,right_d2) - thresh
259
260             print "left = %.2f, right = %2f, t=%.2f" % (left_d2, right_d2, thresh)
261             left_d2 = left_d2 - d_extra
262             right_d2 = right_d2 - d_extra
263             print "Triming distance by %.2f" % d_extra
264
265         left_s  = (self.radius)/left_d2
266         right_s = (self.radius)/right_d2
267
268         close_min = 0.5
269         close_max = 1.3
270
```

```
271         v_left1 = MapStimulus(left_s, close_min, close_max, 1.0, 0.0)
272         v_right1 = MapStimulus(right_s, close_min, close_max, 1.0, 0.0)
273         print "v_left1 = %.3f" % v_left1
274         print "v_right1 = %.3f" % v_right1
275 
276         ### Obstacle avoidance logic
277 
278         if d_left == None:
279             d_left = self.sensor_l
280         if d_right == None:
281             d_right = self.sensor_r
282 
283         v_left2 = MapStimulus(d_right, self.D_MIN, self.D_MAX, 0.0, 1.0)
284         v_right2 = MapStimulus(d_left, self.D_MIN, self.D_MAX, 0.0, 1.0)
285 
286         ratio = 0.2
287         v_left = (v_left1*ratio + v_left2*(1-ratio))
288         v_right = (v_right1*ratio + v_right2*(1-ratio))
289         print "v_left = %.3f" % v_left
290         print "v_right = %.3f" % v_right
291 
292         v_norm = (v_right + v_left)/2.0
293         w_norm = (v_right - v_left)/2.0
294 
295         #print "v_norm = %.3f" % v_norm
296         #print "w_norm = %.3f" % w_norm
297 
298         v_rob = MapStimulus(v_norm, 0.0, 1.0, self.V_MIN, self.V_MAX)
299         w_rob = MapStimulus(w_norm, -0.5, 0.5, -self.W_MAX, self.W_MAX)
300 
301         #return v_left, v_right
302         return v_rob, w_rob
303 
304     def Mixed3a2b(self, d_left=None, d_right=None):
305         r"""Obtiene la respuesta conjunta de evasión y
306         seguimiento, usando el algoritmo 3a para evasión y el
307         2b para seguimiento.
308 
309         La respuesta se da en forma de velocidad lineal y
310         angular en los rangos definidos por :attr:`V_MIN`,
311         :attr:`V_MAX` y :attr:`W_MAX`.
312 
313         Parameters
314         ----------
315         d_left : float, opcional
316             Estímulo de evasión del lado izquierdo, por defecto
317             toma el valor de :attr:`sensor_l`.
318         d_right : float, opcional
319             Estímulo de evasión del lado derecho, por defecto
320             toma el valor de :attr:`sensor_r`.
321 
322         Returns
323         -------
324         v_rob : float
325             Velocidad lineal de respuesta.
326         w_rob : float
327             Velocidad angular de respuesta (rad/s).
328 
329 
```

```
330         .. warning::
331             Debe haberse definido un punto objetivo antes de
332             llamar a esta función.
333 
334         """
335         ### Target following logic
336 
337         left_x  = self.x + np.cos(self.gamma+np.pi/2)*self.radius
338         left_y  = self.y + np.sin(self.gamma+np.pi/2)*self.radius
339         right_x = self.x + np.cos(self.gamma-np.pi/2)*self.radius
340         right_y = self.y + np.sin(self.gamma-np.pi/2)*self.radius
341 
342         print "left is (%.3f, %.3f)" % (left_x, left_y)
343         print "right is (%.3f, %.3f)" % (right_x, right_y)
344 
345         left_d2  = np.sqrt(np.square(self.target[0]-left_x)  + np.square(self.target[1]-
                left_y))
346         right_d2 = np.sqrt(np.square(self.target[0]-right_x) + np.square(self.target[1]-
                right_y))
347         print "left = %.2f, right = %2f" % (left_d2, right_d2)
348 
349         thresh = (self.radius*2)
350         if left_d2 > thresh  or right_d2 > thresh:
351             d_extra = max(left_d2,right_d2) - thresh
352 
353             print "left = %.2f, right = %2f, t=%.2f" % (left_d2, right_d2, thresh)
354             left_d2 = left_d2 - d_extra
355             right_d2 = right_d2 - d_extra
356             print "Triming distance by %.2f" % d_extra
357             print "left = %.2f, right = %2f, t=%.2f" % (left_d2, right_d2, thresh)
358 
359         left_s  = (self.radius)/left_d2
360         right_s = (self.radius)/right_d2
361         print "left = %.2f, right = %2f" % (left_s, right_s)
362 
363         close_min = 0.25
364         close_max = 0.75
365 
366         v_left1 = MapStimulus(right_s, close_min, close_max, 0.0, 1.0)
367         v_right1 = MapStimulus(left_s, close_min, close_max, 0.0, 1.0)
368         print "v_left1 = %.3f" % v_left1
369         print "v_right1 = %.3f" % v_right1
370 
371         ### Obstacle avoidance logic
372 
373         if d_left == None:
374             d_left = self.sensor_l
375         if d_right == None:
376             d_right = self.sensor_r
377 
378         v_left2 = MapStimulus(d_left, self.D_MIN, self.D_MAX, 1.0, 0.0)
379         v_right2 = MapStimulus(d_right, self.D_MIN, self.D_MAX, 1.0, 0.0)
380 
381         ratio = 0.8
382         v_left = (v_left1*ratio + v_left2*(1-ratio))
383         v_right = (v_right1*ratio + v_right2*(1-ratio))
384         print "v_left = %.3f" % v_left
385         print "v_right = %.3f" % v_right
386 
```

```
387         v_norm = (v_right + v_left)/2.0
388         w_norm = (v_right - v_left)/2.0
389 
390         #print "v_norm = %.3f" % v_norm
391         #print "w_norm = %.3f" % w_norm
392 
393         v_rob = MapStimulus(v_norm, 0.0, 1.0, self.V_MIN, self.V_MAX)
394         w_rob = MapStimulus(w_norm, -0.5, 0.5, -self.W_MAX, self.W_MAX)
395 
396         #return v_left, v_right
397         return v_rob, w_rob
398 
399     def UpdatePos(self, x, y, gamma):
400         r"""Actualiza la postura actual del robot.
401 
402         Parameters
403         ----------
404         x : float
405             Posición del robot sobre el eje :math:`x`.
406         y : float
407             Posición del robot sobre el eje :math:`y`.
408         gamma : float
409             Orientación del robot resepecto el eje :math:`z`
410             en radianes.
411 
412         Note
413         -----
414         Solo es necesario usar esta función si se desea hacer
415         seguimiento de trayectorias.
416 
417         """
418 
419         self.x = x
420         self.y = y
421         self.gamma = gamma
422 
423     def SetTarget(self, x, y, unset = False):
424         r"""Define el un punto objetivo para hacer seguimiento
425         de Trayectorias.
426 
427         Permite definir un punto objetivo
428         :math:`\left(x,\:y\right)`. También se puede desactivar
429         el comportamiento por medio del parámetro `unset`.
430 
431         Parameters
432         ----------
433         x : float
434             Posición sobre el eje :math:`x`.
435         y : float
436             Posición sobre el eje :math:`y`.
437         unset : bool, opcional
438             Si se indica como verdadero desactiva el seguimiento.
439 
440         """
441         if unset:
442             self.target = None
443         else:
444             self.target = (x,y)
445 
```

```
446     def UpdateSensors(self, sensor_readings):
447         r"""Procesa las lecturas de los sensores para la
448         detección de obstáculos.
449 
450         Actualiza los valores de :attr:`sensor_r` y
451         :attr:`sensor_l` a partir de las lecturas de los
452         sensores. El comportamiento dependerá del atributo de
453         clase :attr:`s_mode` y el rango de visión :attr:`ALPHA`.
454 
455         Parameters
456         ----------
457         sensor_readings : ndarray of float
458             Lecturas del sensor, un arreglo tipo numpy.ndarray
459             que representa una nube de puntos
460             :math:`(r, \theta)` donde :math:`r` está dado en
461             metros y :math:`\theta` en radianes. Solo se toman
462             en cuenta puntos tal que
463             :math:`\theta\in [-\alpha,\alpha]`.
464 
465         Raises
466         ------
467         TypeError
468             Si ``sensor_readings`` no es de tipo
469             ``numpy.ndarray``.
470         ValueError
471             Si ``sensor_readings`` es de dimensión (2,N).
472 
473         Note
474         -----
475         La forma en que se procesan los datos para definir el
476         estímulo de cada lado del robot cambia dependiendo
477         del valor de `s_mode`, de la siguiente manera:
478 
479         `SMODE_MIN`
480             El estímulo se define como el mínimo valor de
481             distancia en el rango.
482 
483         `SMODE_AVG`
484             El estímulo se define como el promedio de los valores
485             de distancia dentro del rango.
486 
487         `SMODE_FULL`
488             El estímulo se define como el promedio de los valores
489             de distancia dentro del rango. Además se define un
490             número mínimo de puntos por rango. Si hay menos de
491             esta cantidad, se agregan puntos con el máximo valor
492             de distancia :attr:`D_MAX` antes de calcular el
493             promedio.
494 
495         """
496 
497         if (type(sensor_readings) != np.ndarray):
498             raise TypeError("Expected numpy ndarray, received %s" % type(sensor_readings)
                    )
499         elif sensor_readings.ndim != 2 or sensor_readings.shape[1] != 2:
500             raise ValueError("Expected (n, 2) array, received %s" % str(sensor_readings.
                    shape))
501 
502         left = []
```

```
503         right = []
504         for x in xrange(sensor_readings.shape[0]):
505             r, theta = sensor_readings[x,:]
506             if r == 0 and theta == 0:
507                 continue
508             if -self.alpha < theta and theta < 0:
509                 # right
510                 right.append(r)
511             elif 0 < theta and theta < self.alpha:
512                 left.append(r)
513                 #left
514 
515         if self.s_mode == SMODE_MIN:
516             if len(left) > 0:
517                 self.sensor_l = min(left)
518             else:
519                 self.sensor_l = self.D_MAX
520 
521             if len(right) > 0:
522                 self.sensor_r = min(right)
523             else:
524                 self.sensor_r = self.D_MAX
525 
526         elif self.s_mode == SMODE_AVG:
527             if len(left) == 0:
528                 self.sensor_l = self.D_MAX
529             else:
530                 self.sensor_l = sum(left)/float(len(left))
531 
532             if len(right) == 0:
533                 self.sensor_r = self.D_MAX
534             else:
535                 self.sensor_r = sum(right)/float(len(right))
536 
537         elif self.s_mode == SMODE_FULL:
538             num = 40
539 
540             sum_l = sum(left)
541             if len(left) < num:
542                 sum_l += (num-len(left))*self.D_MAX
543                 self.sensor_l = sum_l/num
544             else:
545                 self.sensor_l = sum_l/len(left)
546 
547             sum_r = sum(right)
548             if len(right) < num:
549                 sum_r += (num-len(right))*self.D_MAX
550                 self.sensor_r = sum_r/num
551             else:
552                 self.sensor_r = sum_r/len(right)
553 
554 
555 def MapStimulus(s, s_min, s_max, r_min, r_max):
556     r"""
557     Mapeo lineal de estímulos y respuesta. Aplica una
558     transformación lineal del rango de entrada al rango de
559     salida.
560 
561     Parameters
```

```
562     ----------
563     s : float
564         Valor del estímulo de entrada.
565     s_min : float
566         Cota inferior del valor de ‘s‘.
567     s_max : float
568         Cota superior del valor de ‘s‘.
569     r_min : float
570         Cota de la respuesta para un valor de entrada mínimo.
571     r_max : float
572         Cota de la respuesta para un valor de entrada máximo.
573 
574     Returns
575     -------
576     float
577         Una respuesta en el rango dado por ‘r_min‘ y ‘r_max‘.
578 
579     Note
580     -----
581     La función de mapeo no es mas que la ecuación de una recta
582     que pasa por los puntos :math:‘(s\_min,r\_min)‘ y
583     :math:‘(s\_max,r\_max)‘, por lo que es perfectamente
584     aceptable que :math:‘r\_min > r\_max‘. Esto corresponde a
585     una respuesta con acción inversa, es decir una recta con pendiente negativa.
586     """
587 
588     if s >= s_max:
589         return r_max
590     elif s > s_min:
591         #y = mx + b
592         m = (r_max - r_min)/(s_max - s_min)
593         b = r_min - (m*s_min)
594         return (m*s) + b
595     else:
596         return r_min
597 
598 def main():
599 
600     robot = BraitModel(s_mode=SMODE_FULL)
601     pseudo_readings = np.float_([[0.41, np.radians(x)] for x in range(-5,90,1)])
602     robot.UpdateSensors(pseudo_readings)
603     robot.UpdatePos(0.0,0.0,np.pi/2)
604     robot.SetTarget(0.0000, 0.20)
605     print "Left sensor reading %.2f" % robot.sensor_l
606     print "Right sensor reading %.2f" % robot.sensor_r
607 
608     print
609     print "Speed range: [%.2f, %.2f]" % (robot.V_MIN, robot.V_MAX)
610     print "Rotation range: [%.2f, %.2f]" % (-robot.W_MAX, robot.W_MAX)
611     v, w = robot.Mixed2b3a()
612     dire = "izq" if w > 0 else "der"
613     print "Result: v=%.2f , w=%.2f (%s)" % (v, w, dire)
614 
615     return 0
616 
617 
618 if __name__ == "__main__":
619     main()
```

A.2. Implementación del algoritmo VFH

```
#! /usr/bin/python
# coding=utf8

r"""

Este módulo define el controlador para evasión mediante el
algoritmo VFH, y constantes asociadas.

"""

import numpy as np
import matplotlib.pyplot as plt
import copy


########################################
###             Constantes           ####
########################################
###                                  ####
###                                  ####

# Size of the full Grid
GRID_SIZE = 125
r"""int: Tamaño de la cuadrícula de certeza.
"""

# Resolution of each cell (in m)
RESOLUTION = np.float_(0.04)
r"""float: Resolución de cada celda (en m).
"""

# Size of the active window that
# travels with the robot
WINDOW_SIZE = 15
r"""int: Tamaño de la ventana activa.

.. warning::
    La ventana activa debe tener un número impar de celdas.
"""
assert WINDOW_SIZE%2 == 1, "Window should have an odd number of cells for better results"

WINDOW_CENTER = WINDOW_SIZE/2
r"""int: Índice de la celda central en la ventana activa.

Se define automáticamente a partir de ``WINDOW_SIZE``.
"""

# Size of each polar sector
# in the polar histogram (in degrees)
ALPHA = 5
r"""int: Tamaño de cada sector del histograma polar (en grados).

.. warning::
    ``ALPHA`` debe ser un divisor de 360.
"""
if np.mod(360, ALPHA) != 0:
```

```
57      raise ValueError("Alpha must define an int amount of sectors")
58
59  HIST_SIZE = 360/ALPHA
60  r"""int: Cantidad de sectores en el histograma polar.
61  Se define automáticamente a partir de ``ALPHA``.
62  """
63
64  # Valley/Peak threshold
65  THRESH = 20000.0
66  r"""float: Valor de umbral para valles y picos.
67  """
68
69  WIDE_V = HIST_SIZE/4
70  r"""int: Tamaño límite de un valle ancho y angosto.
71  """
72  V_MAX = 0.0628
73  r"""float: Velocidad máxima del robot (m/s).
74  """
75
76  V_MIN = 0.00628
77  r"""float: Velocidad mínima del robot (m/s).
78  """
79
80  OMEGA_MAX = 1.256
81  r"""float: Velocidad angular máxima del robot (rad/s)
82  """
83
84  # Constants for virtual vector magnitude calculations
85  # D_max^2 = 2*(ws-1/2)^2
86  # A - B*D_max^2 = 1
87  D_max2 = np.square(WINDOW_SIZE-1)/2.0
88  B = np.float_(1.0)
89  r"""float: Constante :math:`b` de la ecuación de la magnitud del
90  vector de obstáculos.
91  """
92
93  A = np.float_(1+B*D_max2)
94  r"""float: Constante :math:`a` de la ecuación de la magnitud del
95  vector de obstáculos. Se define a partir de ``B`` de modo que
96  se cumpla :math:`a - b\cdot d_{max}^2 = 1`.
97  """
98
99  # Active window array indexes
100 MAG = 0
101 BETA = 1
102 DIST2 = 2
103
104 ###                                  ####
105 ###                                  ####
106 ########################################
107
108 class VFHModel:
109     r"""Clase que define el controlador para evasión mediante
110     el algoritmo VFH.
111
112     Attributes
113     ----------
114     obstacle_grid : ndarray of short
115         La cuadrícula de certeza. Cada celda tiene un valor entre
```

```
116         0 y 20 que indica que tan probable es que haya un
117         obstáculo ocupando la celda.
118
119     active_window : ndarray of float
120         La ventana activa que se mueve con el robot. Cada celda
121         contiene tres valores: la magnitud del vector de
122         obstaculos :math:`m_{i,j}`, la dirección del vector
123         :math:`\beta_{i,j}`, y la distancia al robot
124         :math:`d_{i,j}`.
125
126     polar_hist : ndarray of float
127         El histograma polar. Indica la densidad de obstáculos
128         en cada sector alrededor de la vecindad del robot.
129
130     filt_polar_hist : ndarray of float
131         El histograma polar filtrado. Se construye a aplicando
132         un filtro al histograma polar.
133
134     valleys : list of tuples
135         Contiene una lista de los valles en el histograma polar.
136         Cada valle es un par :math:`(s_1,s_2)`, donde :math:`s_1` es
137         el sector donde inicia y :math:`s_2` donde termina, en
138         sentido antihorario.
139
140     x_0 : float
141         Posición del robot sobre el eje :math:`x`.
142     y_0 : float
143         Posición del robot sobre el eje :math:`y`.
144     cita : float
145         Orientación del robot resepecto el eje :math:`z`.
146
147     i_0 : float
148         Índice de la celda que ocupa el robot sobre la cuadrícula
149         de certeza en el eje :math:`x`.
150     j_0 : float
151         Índice de la celda que ocupa el robot sobre la cuadrícula
152         de certeza en el eje :math:`y`.
153     k_0 : float
154         Sector en el histrograma polar de la dirección del robot
155         resepecto el eje :math:`z`.
156
157     target : tuple
158         Punto :math:`\left(x,\:y\right)` objetivo o ``None`` si
159         no se está siguiendo una trayectoria.
160
161     Examples
162     --------
163
164
165     >>> import VFH
166     >>> robot = VFH.VFHModel()
167     >>> robot.update_position(1.5,1.5,270.0)
168     >>> pseudo_readings = np.float_([[0.3, np.radians(x)] for x in range(0,90,1)])
169     >>> robot.update_obstacle_density(pseudo_readings)
170     >>> robot.update_active_window()
171     >>> robot.update_polar_histogram()
172     >>> robot.update_filtered_polar_histogram()
173     >>> robot.find_valleys()
174     >>> try:
```

```
175     >>>     cita = robot.calculate_steering_dir()
176     >>> except:
177     >>>     pass
178     >>> v = robot.calculate_speed(3.14/180)
179
180     """
181
182     def __init__(self):
183
184         # The Obstacle Grid keeps count of the certainty values
185         # for each cell in the grid
186         grid_dim = (GRID_SIZE, GRID_SIZE)
187         self.obstacle_grid = np.zeros( grid_dim, dtype = np.int8 )
188
189         # The Active Window has information (magnitude,
190         # direction, distance to robot) of an obstacle vector
191         # for each active cell in the grid [mij, citaij, dij]
192         window_dim = (WINDOW_SIZE, WINDOW_SIZE, 3)
193         self.active_window = np.zeros( window_dim, dtype = np.float_ )
194
195         # Initialize angles and distance (these don't change)
196
197         for i in xrange(WINDOW_SIZE):
198             for j in xrange(WINDOW_SIZE):
199                 if j == WINDOW_CENTER and i == WINDOW_CENTER:
200                     continue
201                 beta_p = np.degrees(np.arctan2(j-WINDOW_CENTER, i-WINDOW_CENTER))
202                 self.active_window[i,j,BETA] = beta_p + 360 if beta_p < 0 else beta_p
203                 # The distance is measured in terms of cells
204                 # (independent of scale/resolution of the grid)
205                 self.active_window[i,j,DIST2] = np.float_(np.square(i-WINDOW_CENTER) + np
                        .square(j-WINDOW_CENTER))
206
207
208         # The Polar Histogram maps each cell in the active window
209         # to an angular sector
210         hist_dim = HIST_SIZE
211         self.polar_hist = np.zeros( hist_dim, dtype = np.float_ )
212
213         # The Filtered Polar Histogram holds the actual data to
214         # be analized
215         self.filt_polar_hist = np.zeros( hist_dim, dtype = np.float_ )
216
217         # The valleys are stored as start-end pairs of sectors
218         # in the filtered polar histogram
219         self.valleys = []
220
221         # Real position of the robot
222         self.x_0 = 0.0
223         self.y_0 = 0.0
224         self.cita = 0.0
225
226         # Cell of the robot
227         self.i_0 = 0
228         self.j_0 = 0
229         # Sector of the robot
230         self.k_0 = 0
231
232         # (x,y) pair representing the target, or
```

```
233         # None if there is no target
234         self.target = None
235 
236     def update_position(self, x, y, cita):
237         r"""Actualiza la posición del robot.
238 
239         Parameters
240         ----------
241         x : float
242             Posición absoluta del robot sobre el eje :math:`x`.
243         y : float
244             Posición absoluta del robot sobre el eje :math:`y`.
245         cita : float
246             Orientación del robot resepecto el eje :math:`z`
247             en grados.
248 
249         Notes
250         -----
251 
252         Modifica los siguientes atributos de clase:
253 
254         .. hlist::
255 
256             * :attr:`x_0`
257             * :attr:`y_0`
258             * :attr:`cita`
259             * :attr:`i_0`
260             * :attr:`j_0`
261             * :attr:`k_0`
262 
263         """
264 
265         self.x_0 = x
266         self.y_0 = y
267         self.cita = cita if cita >= 0 else cita + 360
268 
269         self.i_0 = int(x / RESOLUTION)
270         self.j_0 = int(y / RESOLUTION)
271         self.k_0 = int(self.cita / ALPHA)%HIST_SIZE
272 
273     def set_target(self, x=None, y=None):
274         r"""Define el punto objetivo o deshabilita el seguimiento
275         de trayectorias.
276 
277         Define un punto objetivo :math:`(x,y)` para la evasión
278         con seguimiento de trayectorias, o deshabilita el
279         seguimiento si el método es invocado sin parámetros.
280 
281         Parameters
282         ----------
283         x : float, opcional
284             Posición absoluta del robot sobre el eje :math:`x`.
285         y : float, opcional
286             Posición absoluta del robot sobre el eje :math:`y`.
287 
288         Returns
289         -------
290         int
291             Retorna 1 si se estableció un punto objetivo o 0
```

```
292             si inhabilitó el seguimiento.
293
294         Raises
295         ------
296         ValueError
297             Si el objetivo dado se sale de la cuadrícula de
298             certeza.
299
300         Notes
301         -----
302         Modifica los siguientes atributos de clase:
303
304          * :attr:`target`
305
306         """
307         if x != None and y != None:
308             if x >= 0 and y >= 0 and \
309                     x < GRID_SIZE*RESOLUTION and \
310                     y < GRID_SIZE*RESOLUTION:
311                 self.target = (x,y)
312                 return 1
313             else:
314                 raise ValueError("Tried to set the target ({:d},{:d}) outside the
                        obstacle grid".format(x,y))
315         else:
316             self.target = None
317             return 0
318
319     def update_obstacle_density(self, sensor_readings):
320         r"""Actualiza la cuadrícula de certeza a partir de las
321         lecturas de un sensor.
322
323         Para cada lectura aumenta el valor de una única celda
324         en 1, hasta un máximo de 20.
325
326         Parameters
327         ----------
328         sensor_readings : ndarray
329             Una estructura de datos tipo ``numpy.ndarray`` que
330             contiene las lecturas de un sensor de distancias.
331             Debe ser un arreglo de dimensiones
332             :math:`(n\times 2)` para :math:`n` puntos o
333             lecturas, donde cada par representa una coordenada
334             polar :math:`(r,\theta)` respecto al marco de
335             referencia del robot.
336
337         Raises
338         ------
339         TypeError
340             Si ``sensor_readings`` no es de tipo
341             ``numpy.ndarray``.
342         ValueError
343             Si ``sensor_readings`` no tiene las dimensiones
344             correctas.
345
346         Notes
347         -----
348         Las coordenadas deben ser dadas en metros y radianes.
349
```

```
350         Modifica los siguientes atributos de clase:
351
352          * :attr:`obstacle_grid`
353
354         """
355         # Receives a numpy array of (r, theta) data points #
356         # r in meters, theta in radians
357         # 0,0 means no reading
358
359         if (type(sensor_readings) != np.ndarray):
360             raise TypeError("Expected numpy ndarray, received %s" % type(sensor_readings)
                )
361         elif sensor_readings.ndim != 2 or sensor_readings.shape[1] != 2:
362             raise ValueError("Expected (n, 2) array, received %s" % str(sensor_readings.
                shape))
363
364         for x in xrange(sensor_readings.shape[0]):
365             r, theta = sensor_readings[x,:]
366             if r == 0 and theta == 0:
367                 continue
368             i = int( (self.x_0 + r*np.cos(theta + np.radians(self.cita)))/RESOLUTION )
369             j = int( (self.y_0 + r*np.sin(theta + np.radians(self.cita)))/RESOLUTION )
370             if self.obstacle_grid[i,j] < 20: self.obstacle_grid[i,j] += 1
371
372     def _active_grid(self):
373
374         i_window = max(self.i_0 - (WINDOW_CENTER), 0)
375         j_window = max(self.j_0 - (WINDOW_CENTER), 0)
376         i_max = min(i_window + WINDOW_SIZE, GRID_SIZE)
377         j_max = min(j_window + WINDOW_SIZE, GRID_SIZE)
378
379         return self.obstacle_grid[i_window:i_max, j_window:j_max]
380
381     def update_active_window(self):
382         r"""Calcula la magnitud de los vectores de obstáculo para
383         las celdas de la ventana activa.
384
385         El cálculo se hace a partir de los datos guardados en la
386         :attr:`cuadrícula de certeza<obstacle_grid>`.
387
388         Notes
389         -----
390         Modifica los siguientes atributos de clase:
391
392          * :attr:`active_window`
393
394         """
395
396         i_window = self.i_0 - (WINDOW_CENTER)
397         j_window = self.j_0 - (WINDOW_CENTER)
398         for i in xrange(WINDOW_SIZE):
399             for j in xrange(WINDOW_SIZE):
400                 if j == WINDOW_CENTER and i == WINDOW_CENTER:
401                     continue
402
403                 grid_i = i_window + i
404                 grid_j = j_window + j
405
406                 if grid_i < 0 or grid_i > GRID_SIZE or grid_j < 0 or grid_j > GRID_SIZE:
```

```
407                     self.active_window[i,j,MAG] = 0.0
408                 else :
409                     cij = np.float_(self.obstacle_grid[grid_i, grid_j])
410                     mij = np.square(cij)*(A - B*self.active_window[i, j, DIST2])
411                     self.active_window[i,j,MAG] = mij
412 
413         #return self.active_window
414 
415     def update_polar_histogram(self):
416         r"""Actualiza el histograma polar de obstáculos.
417 
418         El cálculo se hace a partir de los vectores de obstáculo
419         guardados en la :attr:`ventana activa<active_window>`.
420 
421         Notes
422         -----
423         Modifica los siguientes atributos de clase:
424 
425          * :attr:`polar_hist`
426 
427         """
428 
429         for i in xrange(HIST_SIZE):
430             self.polar_hist[i] = 0
431 
432         for i in xrange(WINDOW_SIZE):
433             for j in xrange(WINDOW_SIZE):
434                 if j == WINDOW_CENTER and i == WINDOW_CENTER:
435                     continue
436                 k = int(self.active_window[i, j, BETA]/ALPHA) % HIST_SIZE
437                 assert k < HIST_SIZE and k >= 0, "Error for polar histogram index: %d on
                        i = %d, j = %d" % (k, i, j)
438                 self.polar_hist[k] += self.active_window[i, j, MAG]
439 
440         #return self.polar_hist
441 
442     def update_filtered_polar_histogram(self):
443         r"""Calcula el histograma polar filtrado a partir de el
444         :attr:`histograma polar<polar_hist>`.
445 
446         Notes
447         -----
448         Modifica los siguientes atributos de clase:
449 
450          * :attr:`filt_polar_hist`
451         """
452 
453         ## Amount of adyacent sectors to filter
454         L = 5
455         assert (2*L + 1) < HIST_SIZE
456         for i in xrange(HIST_SIZE):
457             coef = [[L - abs(L-j) + 1, (i + (j-L))%HIST_SIZE] for j in xrange(2*L+1)]
458             self.filt_polar_hist[i] = np.sum([c*self.polar_hist[k] for c, k in coef])/(2*
                    L+1)
459 
460     def find_valleys(self):
461         r"""Analiza el histograma polar filtrado y determina los
462         valles candidatos.
463 
```

```
464         Los valles se obtienen a partir del
465         :attr:`histograma polar filtrado<filt_polar_hist>` y
466         el :const:`valor de umbral<THRESH>`.
467 
468         Returns
469         -------
470         int
471              * ``-1`` si no se encuentra ningún sector con un valor
472                mayor a :const:`THRESH` en el hisotgrama filtrado.
473              * ``0`` de lo contrario.
474 
475         """
476 
477         start = None
478         for x in xrange(HIST_SIZE):
479             if self.filt_polar_hist[x] > THRESH:
480                 start = copy.copy(x)
481                 #print "Found start at {:d}".format(x)
482                 break
483 
484         # If no value was found over the threshold no action
485         # needs to be taken since there are no nearby obstacles
486         if start == None:
487             return -1
488 
489         # Else, look for valleys after 'start'
490         #print "Looking for valleys after k={:d}, c={:.1f}".format(start, start*ALPHA)
491         self.valleys = []
492         valley_found = False
493         for i in xrange(HIST_SIZE+1):
494             index = (start + i)%HIST_SIZE
495             #print "Sector {:d}, h={:.2f}".format(index, self.filt_polar_hist[index])
496             if not valley_found:
497                 if self.filt_polar_hist[index] < THRESH:
498                     v_start = index
499                     valley_found = True
500 
501             else:
502                 if self.filt_polar_hist[index] > THRESH:
503                     self.valleys.append(tuple([v_start, index-1]))
504                     valley_found = False
505         return 0
506 
507     def calculate_steering_dir(self):
508         r"""Calcula la dirección de movimiento para la evasión.
509 
510         El cálculo de la dirección para el robot se hace a
511         partir de los valles candidato, la orientación actual
512         y el objetivo. Si no hay objetivo, se toma la dirección
513         actual de movimiento como objetivo.
514 
515         Returns
516         -------
517         new_dir : float
518             La dirección del movimiento, dada en grados en el
519             rango :math:`[0°,\:360°[` .
520 
521         Raises
522         ------
```

```
523         Exception
524             Si :attr:‘valleys‘ está vacío.
525 
526         """
527 
528         # Set the target sector. If there is a target point
529         # calculate the sector from that, else it's the
530         # current orientation
531         k_t = None
532     dir_t = None
533         if self.target == None:
534             k_t = self.k_0
535         dir_t = self.cita
536         else:
537             dx = self.target[0] - self.x_0
538             dy = self.target[1] - self.y_0
539             angle = np.degrees(np.arctan2(dy, dx))
540             angle = angle + 360 if angle < 0 else angle
541             print "target dir is %.1f" % angle
542             k_t = int(angle / ALPHA)%HIST_SIZE
543         dir_t = angle
544 
545         # First we determine which valley is closest to the
546         # target or current robot orientation
547         closest_dist = None
548         closest_valley = None
549         closest_sect = None
550 
551 
552         if len(self.valleys) == 0:
553             raise Exception("No candidate valleys found to calculate a new direction")
554 
555         for v in self.valleys:
556 
557             d1, d2 = [self._dist(self.filt_polar_hist, k_t, sector) for sector in v]
558 
559             if closest_dist != None:
560                 min_dist = min(d1, d2)
561                 if min_dist < closest_dist:
562                     closest_dist = min_dist
563                     closest_valley = v
564                     if d1 < d2:
565                         closest_sect = 0
566                     else:
567                         closest_sect = 1
568 
569             else:
570                 closest_dist = min(d1, d2)
571                 closest_valley = v
572                 if d1 < d2:
573                     closest_sect = 0
574                 else:
575                     closest_sect = 1
576 
577         print "Closest valley is %s" % str(closest_valley)
578         s1, s2 = closest_valley
579         v_size = (s2 - s1) if s2 >= s1 else HIST_SIZE - (s1-s2)
580 
581         if v_size < WIDE_V :
```

```
582             # For narrow valleys move in the direction of the middle of
583             # the valley.
584             print "Crossing a narrow valley"
585             new_dir = ALPHA * (s1 + v_size/2.0)
586         else:
587             print "Crossing a wide valley"
588 
589             # For wide valleys move in the direction of travel if
590             # the closest distance is bigger than WIDE_V/2 and its
591             # inside the valley
592             if closest_dist > WIDE_V/2.0:
593                 k_inside = False
594                 if s1 < s2:
595                     k_inside = s1 < k_t and k_t < s2
596                 else:
597                     k_inside = not (s2 < k_t and k_t < s1)
598 
599                 if k_inside:
600                     print "Maintining current direction"
601                     new_dir = dir_t
602                 else:
603                     print "Current direction is blocked!"
604 
605             # If the target is closer to the edge then travel near
606             # the closer edge of the valley
607             elif closest_sect == 0:
608                 print "Staying near right"
609                 new_dir = ALPHA * (s1 + WIDE_V/2.0)
610             else:
611                 print "Staying near left"
612                 new_dir = ALPHA * (s2 - WIDE_V/2.0)
613 
614         if new_dir >= 360:
615             new_dir = new_dir - 360
616         elif new_dir < 0:
617             new_dir = new_dir + 360
618 
619         print "Setting course to target at %.2f" % new_dir
620         return new_dir
621 
622     def calculate_speed(self, omega=0):
623         r"""Calcula la velocidad del robot.
624 
625         Calcula la velocidad a partir de la densidad de
626         obstáculos en la dirección actual del movimiento.
627         El robot se mueve más despacio entre mayor sea la
628         densidad.
629 
630         Opcionalmente recibe la velocidad angular del robot como
631         parámetro. En este caso, se hace una reducción adicional
632         de la velocidad dependiendo del valor de ``omega``
633         en comparación a :const:`OMEGA_MAX`.
634 
635         Parameters
636         ----------
637         omega : float, opcional
638             Velocidad angular del robot (rad/s).
639 
640         Returns
```

```
641         -------
642         V : float
643             Velocidad lineal deseada del robot (m/s).
644         """
645 
646         # Omega in [rad/s]
647         # Obstacle density in the current direction of travel
648         H_M = THRESH*1.8
649         h_cp = self.filt_polar_hist[self.k_0]
650         h_cpp = min(h_cp, H_M)
651 
652         V_prime = V_MAX*(1 - h_cpp/H_M)
653 
654         V = V_prime*(1 - omega/OMEGA_MAX) + V_MIN
655 
656         return V
657 
658     @staticmethod
659     def _dist(array,i,j):
660         n_dist = abs(i-j)
661         return min(n_dist, len(array) - n_dist)
662 
663 
664 def main():
665     np.set_printoptions(precision=2)
666     robot = VFHModel()
667     print "Obstacle grid"
668     print robot.obstacle_grid, "\n"
669     print "Active Window angles"
670     print robot.active_window[:,:,BETA], "\n"
671     print "Active Window squared distances"
672     print robot.active_window[:,:,DIST2], "\n"
673     print "Max distance squared: %f" % D_max2
674 
675 
676     print("Updating the obstacle grid and robot position")
677 
678 
679 
680 #     robot.obstacle_grid[1,6] = 1
681 #     robot.obstacle_grid[1,5] = 2
682 #     robot.obstacle_grid[1,4] = 2
683 #     robot.obstacle_grid[1,3] = 5
684 #     robot.obstacle_grid[1,2] = 13
685 #     robot.obstacle_grid[2,2] = 3
686 #     robot.obstacle_grid[3,2] = 3
687 #     robot.obstacle_grid[4,2] = 3
688 #
689 #     robot.obstacle_grid[9,2] = 4
690 #     robot.obstacle_grid[9,3] = 5
691 #     robot.obstacle_grid[9,4] = 6
692 #     robot.obstacle_grid[9,5] = 5
693 #     robot.obstacle_grid[9,6] = 4
694 
695     print robot.i_0, robot.j_0
696     print robot._active_grid(), "\n"
697 
698     print "Simulating a set of sensor readings"
699     pseudo_readings = np.float_([[0.3, np.radians(x)] for x in range(0,90,1)])
```

```
700     robot.update_obstacle_density(pseudo_readings)
701     robot.update_obstacle_density(pseudo_readings)
702     #print int(1.2/RESOLUTION), int(1.3/RESOLUTION)
703     robot.obstacle_grid[23, 26] = 20
704     robot.obstacle_grid[23, 27] = 20
705     robot.obstacle_grid[24, 26] = 20
706     robot.obstacle_grid[24, 27] = 20
707     robot.obstacle_grid[24, 25] = 17
708     robot.obstacle_grid[23, 25] = 16
709     robot.obstacle_grid[25, 24] = 10
710     robot.obstacle_grid[24, 24] = 8
711 
712     #robot.obstacle_grid[30, 30] = -20
713 
714     print "Updating the active window"
715     robot.update_active_window()
716     print robot.active_window[:, :, MAG], "\n"
717 
718     print "Updating polar histogram"
719     robot.update_polar_histogram()
720     print robot.polar_hist, "\n"
721 
722     print "Updating filtered histogram"
723     robot.update_filtered_polar_histogram()
724     print robot.filt_polar_hist, "\n"
725 
726     print "Looking for valleys"
727     robot.find_valleys()
728     print robot.valleys, "\n"
729 
730     try:
731         print "Setting steer direction"
732         cita = robot.calculate_steering_dir()
733         print cita, "\n"
734     except:
735         pass
736 
737     ### Figuras y graficos ###
738     plt.figure(1)
739     x = [ALPHA*x for x in range(len(robot.filt_polar_hist))]
740     i = [a for a in range(len(robot.filt_polar_hist))]
741     plt.bar(x, robot.polar_hist, 4.0, 0, color='r')
742     plt.title("Histograma polar")
743 
744     plt.figure(2)
745     plt.bar(x, robot.filt_polar_hist, 4.0 ) #, 0.1, 0, color='b')
746     plt.title("Histograma polar filtrado")
747 
748     plt.figure(3)
749     plt.pcolor(robot._active_grid().T, alpha=0.75, edgecolors='k',vmin=0,vmax=20)
750     plt.xlabel("X")
751     plt.ylabel("Y", rotation='horizontal')
752     plt.title("Ventana activa")
753 
754     plt.show()
755 
756 if __name__ == "__main__":
757     main()
```

A.3. Implementación del algoritmo VFH+

```
#! /usr/bin/python
# coding=utf8

r"""

Este módulo define el controlador para evasión mediante el
algoritmo VFH+, y constantes asociadas.

"""

import numpy as np
import matplotlib.pyplot as plt



########################################
###             Constantes           ####
########################################
###                                  ####
###                                  ####

# Size of the full Grid
GRID_SIZE = 125
r"""int: Tamaño de la cuadrícula de certeza.
"""

# Maximum certantinty value
C_MAX = 20
r"""int: Valor máximo de certeza.
"""

# Resolution of each cell (in m)
RESOLUTION = np.float_(0.04)
r"""float: Resolución de cada celda (en m).
"""

# Size of the active window that
# travels with the robot
WINDOW_SIZE = 25
r"""int: Tamaño de la ventana activa.

.. warning::
    La ventana activa debe tener un número impar de celdas.
"""

assert WINDOW_SIZE%2 == 1, "Window should have an odd number of cells for better results"

WINDOW_CENTER = WINDOW_SIZE/2
r"""int: Índice de la celda central en la ventana activa.

Se define automáticamente a partir de ``WINDOW_SIZE``.
"""

# Size of each polar sector
# in the polar histogram (in degrees)
ALPHA = 5
```

```
r"""int: Tamaño de cada sector del histograma polar (en grados).

.. warning::
    ``ALPHA`` debe ser un divisor de 360.
"""
assert np.mod(360, ALPHA) == 0, "Alpha must define an int amount of sectors"

HIST_SIZE = 360/ALPHA
r"""int: Cantidad de sectores en el histograma polar.
Se define automáticamente a partir de ``ALPHA``.
"""

# Constants for virtual vector magnitude calculations
# D_max^2 = 2*(ws-1/2)^2*R^2
# A - B*D_max^2 = 1
D_max2 = np.square((WINDOW_SIZE-1)*RESOLUTION)/2.0
B = np.float_(10.0)
r"""float: Constante :math:`b` de la ecuación de la magnitud del
vector de obstáculos.
"""

A = np.float_(1+B*D_max2)
r"""float: Constante :math:`a` de la ecuación de la magnitud del
vector de obstáculos. Se define a partir de ``B`` de modo que
se cumpla :math:`a - b\cdot d_{max}^2 = 1`.
"""

# Constants for obstacle enlargement
# Robot radius
R_ROB = 0.02
r"""float: Radio del robot (en m).
"""

# Minimum obstacle distance
D_S   = 0.04
r"""float: Distancia mínima de obstáculo (en m).

"""
R_RS  = R_ROB + D_S
r"""float: Radio de compensación de obstáculos (en m).
"""

# Valley/Peak threshold
T_LO = 3000.0
"""float: Valor de umbral inferior para valles en el histograma polar.
"""
T_HI = 3500.0
"""float: Valor de umbral superior para valles en el histograma polar.
"""
WIDE_V = HIST_SIZE/8
"""int: Tamaño mínimo de valle ancho.
"""
V_MAX = 0.0628
"""float: Velocidad máxima del robot.
"""
V_MIN = 0.0 #0.00628
"""float: Velocidad mínima del robot.
"""

```

```
116 # Cost function constants
117 # Recommended mu1 > m2 + m3
118 # mu1 = target following cost
119 # mu2 = sharp steering cost
120 # mu3 = direction commiting cost
121 mu1 = 6.0
122 """float: Peso del costo de seguimiento.
123 """
124 mu2 = 2.0
125 """float: Peso del costo de cambios abruptos de dirección.
126 """
127 mu3 = 2.0
128 """float: Peso del costo de compromiso a una dirección.
129 """
130 MAX_COST = 180.0*(mu1+mu2+mu3)
131 """float: Máximo valor de la función de costo.
132 """
133
134 # Active window array indexes
135 MAG = 0
136 BETA = 1
137 DIST2 = 2
138 ABDIST = 3
139 GAMA = 4
140
141 ###                                  ####
142 ###                                  ####
143 ########################################
144
145 class VFHPModel:
146     r"""Clase que define el controlador para evasión mediante
147     el algoritmo VFH+.
148
149     Attributes
150     ----------
151     obstacle_grid : ndarray of short
152         La cuadrícula de certeza. Cada celda tiene un valor entre
153         0 y :const:`C_MAX` que indica que tan probable es que
154         haya un obstáculo ocupando la celda.
155
156     active_window : ndarray of float
157         La ventana activa que se mueve con el robot. Cada celda
158         contiene cuatro valores: la magnitud del vector de
159         obstaculos :math:`m_{i,j}`, la dirección del vector
160         :math:`\beta_{i,j}`, la distancia al robot
161         :math:`d_{i,j}` y el ángulo de ensanchamiento
162         :math:`\gamma_{i,j}` .
163
164     polar_hist : ndarray of float
165         El histograma polar. Indica la densidad de obstáculos
166         en cada sector alrededor de la vecindad del robot.
167
168     bin_polar_hist : ndarray of bool
169         El histograma polar binario. Se construye a partir del
170         histograma polar. Un valor ``True`` indica un sector
171         bloqueado.
172
173     masked_polar_hist : ndarray of bool
174         El histograma polar mascarado. Se contruye a partir del
```

```
175         histograma polar binario y los radios de giro
176         instantáneos. Un valor ``True`` indica un sector
177         bloqueado.
178
179     valleys : list of 2-tuples
180         Contiene una lista de los valles en el histograma polar.
181         Cada es un par :math:`(s_1,s_2)`, donde :math:`s_1` es
182         el sector donde inicia y :math:`s_2` donde termina, en
183         sentido antihorario.
184
185     x_0 : float
186         Posición del robot sobre el eje :math:`x`.
187     y_0 : float
188         Posición del robot sobre el eje :math:`y`.
189     cita : float
190         Orientación del robot resepecto el eje :math:`z`.
191
192     i_0 : float
193         Índice de la celda que ocupa el robot sobre la cuadrícula
194         de certeza en el eje :math:`x`.
195     j_0 : float
196         Índice de la celda que ocupa el robot sobre la cuadrícula
197         de certeza en el eje :math:`y`.
198     k_0 : float
199         Sector en el histrograma polar de la dirección del robot
200         resepecto el eje :math:`z`.
201
202     target : tuple
203         Punto :math:`\left(x,\:y\right)` objetivo o ``None`` si
204         no se está siguiendo una trayectoria.
205
206     prev_dir : float
207         Última dirección de control.
208     prev_cost : cost
209         Costo de la última dirección de control.
210
211
212
213     Examples
214     --------
215
216     >>> import VFHP
217     >>> pseudo_readings = np.float_([[0.3, np.radians(x)] for x in range(0,90,1)])
218     >>> R_ROB = 0.01
219     >>> robot = VFHP.VFHModel()
220     >>> robot.update_position(1.5,1.5,270.0)
221     >>> robot.update_obstacle_density(pseudo_readings)
222     >>> robot.update_active_window()
223     >>> robot.update_polar_histogram()
224     >>> robot.update_bin_polar_histogram()
225     >>> robot.update_masked_polar_hist(R_ROB,R_ROB)
226     >>> valles = robot.find_valleys()
227     >>> cita, v = robot.calculate_steering_dir(valles)
228
229     """
230
231     def __init__(self):
232
233         # The Obstacle Grid keeps count of the certainty values
```

```
234         # for each cell in the grid
235         grid_dim = (GRID_SIZE, GRID_SIZE)
236         self.obstacle_grid = np.zeros( grid_dim, dtype = np.int8 )
237 
238         # The Active Window has information (magnitude,
239         # direction, distance to robot) of an obstacle vector
240         # for each active cell in the grid [mij, betaij, d^2, a-bd^2_ij, gij]
241         window_dim = (WINDOW_SIZE, WINDOW_SIZE, 5)
242         self.active_window = np.zeros( window_dim, dtype = np.float_ )
243 
244         # Initialize angles and distance (these don't change)
245 
246         print "INITIALIZING"
247         for i in xrange(WINDOW_SIZE):
248             for j in xrange(WINDOW_SIZE):
249                 if j == WINDOW_CENTER and i == WINDOW_CENTER:
250                     continue
251                 beta_p = np.degrees(np.arctan2(j-WINDOW_CENTER, i-WINDOW_CENTER))
252                 self.active_window[i,j,BETA] = beta_p + 360 if beta_p < 0 else beta_p
253                 # The distance is measured in terms of cells
254                 # (independent of scale/resolution of the grid)
255                 dist2 = np.square(RESOLUTION)*np.float_(np.square(i-WINDOW_CENTER) + np.
                        square(j-WINDOW_CENTER))
256                 self.active_window[i,j,DIST2] = dist2
257                 self.active_window[i,j,ABDIST] = A - B*dist2
258                 self.active_window[i,j,GAMA] = np.degrees(np.arcsin(np.float_(R_RS)/np.
                        sqrt(dist2)))
259                 if np.isnan(self.active_window[i,j,GAMA]):
260                     print "cell ({:d},{:d}) gamma = ".format(i,j), self.active_window[i,j
                            ,GAMA]
261                     print "setting to 90.0"
262                     #print dist2, type(dist2)
263                     #print R_RS, type(R_RS)
264                     #print np.sqrt(dist2)
265                     #print np.float_(R_RS)/np.sqrt(dist2)
266                     #print "arcoseno ", np.arcsin(np.float_(R_RS)/np.sqrt(dist2))
267                     self.active_window[i,j,GAMA] = np.float_(90.0)
268 
269         # The Polar Histogram maps each cell in the active window
270         # to one or more angular sector
271         hist_dim = HIST_SIZE
272         self.polar_hist = np.zeros( hist_dim, dtype = np.float_ )
273 
274         # The Binary Polar Histogram defines free and blocked sectors
275         self.bin_polar_hist = np.zeros( hist_dim, dtype = np.bool_ )
276 
277         # The Masked Polar Histogram further restricts the free sectors
278         # depending on vehicle dynamics
279         self.masked_polar_hist = np.zeros( hist_dim, dtype = np.bool_ )
280 
281         # The valleys are stored as start-end pairs of sectors
282         # in the filtered polar histogram
283         self.valleys = []
284 
285         # Real position of the robot
286         self.x_0 = 0.0
287         self.y_0 = 0.0
288         self.cita = 0.0
289 
```

```
290         # Cell of the robot
291         self.i_0 = 0
292         self.j_0 = 0
293         # Sector of the robot
294         self.k_0 = 0
295 
296         self.target = None
297         self.prev_dir = 0
298         self.prev_cost = 0
299 
300     def set_target(self, x=None, y=None):
301         r"""Define el punto objetivo o deshabilita el seguimiento
302         de trayectorias.
303 
304         Define un punto objetivo :math:`(x,y)` para la evasión
305         con seguimiento de trayectorias, o deshabilita el
306         seguimiento si el método es invocado sin parámetros.
307 
308         Parameters
309         ----------
310         x : float, opcional
311             Posición absoluta del objetivo sobre el eje :math:`x`.
312         y : float, opcional
313             Posición absoluta del objetivo sobre el eje :math:`y`.
314 
315         Returns
316         -------
317         int
318             Retorna 1 si se estableció un punto objetivo o 0
319             si inhabilitó el seguimiento.
320 
321         Raises
322         ------
323         ValueError
324             Si el objetivo dado se sale de la cuadrícula de
325             certeza.
326 
327         Notes
328         -----
329 
330         Modifica los siguientes atributos de clase:
331          * :attr:`target`
332 
333         """
334         if x != None and y != None:
335             if x >= 0 and y >= 0 and \
336                     x < GRID_SIZE*RESOLUTION and \
337                     y < GRID_SIZE*RESOLUTION:
338                 self.target = (x,y)
339                 return 1
340             else:
341                 raise ValueError("Tried to set the target ({:d},{:d}) outside the
                        obstacle grid".format(x,y))
342         else:
343             self.target = None
344             return 0
345 
346     def update_position(self, x, y, cita):
347         r"""Actualiza la posición del robot.
```

```
348
349         Parameters
350         ----------
351         x : float
352             Posición absoluta del robot sobre el eje :math:‘x‘.
353         y : float
354             Posición absoluta del robot sobre el eje :math:‘y‘.
355         cita : float
356             Orientación del robot resepecto el eje :math:‘z‘
357             en grados.
358
359         Notes
360         -----
361
362         Modifica los siguientes atributos de clase:
363
364         .. hlist::
365
366             * :attr:‘x_0‘
367             * :attr:‘y_0‘
368             * :attr:‘cita‘
369             * :attr:‘i_0‘
370             * :attr:‘j_0‘
371             * :attr:‘k_0‘
372
373         """
374         self.x_0 = x
375         self.y_0 = y
376         self.cita = cita if cita >= 0 else cita + 360
377
378         self.i_0 = int(x / RESOLUTION)
379         self.j_0 = int(y / RESOLUTION)
380         self.k_0 = int(self.cita / ALPHA)%HIST_SIZE
381
382
383     def update_obstacle_density(self, sensor_readings):
384         r"""Actualiza la cuadrícula de certeza a partir de las
385         lecturas de un sensor.
386
387         Para cada lectura aumenta el valor de una única celda
388         en 1, hasta un máximo de :const:‘C_MAX‘.
389
390         Parameters
391         ----------
392         sensor_readings : ndarray
393             Una estructura de datos tipo ‘‘numpy.ndarray‘‘ que
394             contiene las lecturas de un sensor de distancias.
395             Debe ser un arreglo de dimensiones
396             :math:‘(n\times 2)‘ para :math:‘n‘ puntos o
397             lecturas, donde cada par representa una coordenada
398             polar :math:‘(r,\theta)‘ respecto al marco de
399             referencia del robot.
400
401         Raises
402         ------
403         TypeError
404             Si ‘‘sensor_readings‘‘ no es de tipo
405             ‘‘numpy.ndarray‘‘.
406         ValueError
```

```
407             Si ``sensor_readings`` no tiene las dimensiones
408             correctas.
409
410         Notes
411         -----
412         Las coordenadas deben ser dadas en metros y radianes.
413
414         Modifica los siguientes atributos de clase:
415          * :attr:`obstacle_grid`
416
417         """
418         # Receives a numpy array of (r, theta) data points #
419         # r in meters, theta in radians
420         # 0,0 means no reading
421
422         if (type(sensor_readings) != np.ndarray):
423             raise TypeError("Expected numpy ndarray, received %s" % type(sensor_readings)
                    )
424         elif sensor_readings.ndim != 2 or sensor_readings.shape[1] != 2:
425             raise ValueError("Expected (n, 2) array, received %s" % str(sensor_readings.
                    shape))
426
427         for x in xrange(sensor_readings.shape[0]):
428             r, theta = sensor_readings[x,:]
429             if r == 0 and theta == 0:
430                 continue
431             i = int( (self.x_0 + r*np.cos(theta + np.radians(self.cita)))/RESOLUTION )
432             j = int( (self.y_0 + r*np.sin(theta + np.radians(self.cita)))/RESOLUTION )
433             if self.obstacle_grid[i,j] < C_MAX: self.obstacle_grid[i,j] += 1
434
435     def _active_grid(self):
436
437         i_window = max(self.i_0 - (WINDOW_CENTER), 0)
438         j_window = max(self.j_0 - (WINDOW_CENTER), 0)
439         i_max = min(i_window + WINDOW_SIZE, GRID_SIZE)
440         j_max = min(j_window + WINDOW_SIZE, GRID_SIZE)
441
442         return self.obstacle_grid[i_window:i_max, j_window:j_max]
443
444     def update_active_window(self):
445         r"""Calcula la magnitud de los vectores de obstáculo para
446         las celdas de la ventana activa.
447
448         El cálculo se hace a partir de los datos guardados en la
449         :attr:`cuadrícula de certeza<obstacle_grid>`.
450
451         Notes
452         -----
453         Modifica los siguientes atributos de clase:
454          * :attr:`active_window`
455
456         """
457
458         i_window = self.i_0 - (WINDOW_CENTER)
459         j_window = self.j_0 - (WINDOW_CENTER)
460         for i in xrange(WINDOW_SIZE):
461             for j in xrange(WINDOW_SIZE):
462                 if j == WINDOW_CENTER and i == WINDOW_CENTER:
463                     continue
```

```
464
465                 grid_i = i_window + i
466                 grid_j = j_window + j
467
468                 if grid_i < 0 or grid_i > GRID_SIZE or grid_j < 0 or grid_j > GRID_SIZE:
469                     self.active_window[i,j,MAG] = 0.0
470                 else :
471                     cij = np.float_(self.obstacle_grid[grid_i, grid_j])
472                     self.active_window[i,j,MAG] = np.square(cij)*self.active_window[i,j,
                        ABDIST]
473
474         #return self.active_window
475
476     def update_polar_histogram(self):
477         r"""Actualiza el histograma polar de obstáculos.
478
479         El cálculo se hace a partir de los vectores de obstáculo
480         guardados en la :attr:`ventana activa<active_window>`.
481         Cada celda puede afectar más de un sector angular,
482         esto se determina mediante el radio de compensación
483         :const:`R_RS`.
484
485         Notes
486         -----
487         Modifica los siguientes atributos de clase:
488          * :attr:`polar_hist`
489
490         """
491
492         for i in xrange(HIST_SIZE):
493             self.polar_hist[i] = 0
494
495         for i in xrange(WINDOW_SIZE):
496             for j in xrange(WINDOW_SIZE):
497                 if j == WINDOW_CENTER and i == WINDOW_CENTER:
498                     continue
499
500                 #print "Determine ranges for cell ({:d}, {:d}".format(i,j)
501                 beta = self.active_window[i,j,BETA]
502                 gama = self.active_window[i,j,GAMA]
503                 alfa = ALPHA
504                 #print beta, type(beta), gama, type(gama), alfa, type(alfa)
505
506                 # Determine the range of histogram sectors that needs to be updated
507                 low = int(np.ceil((self.active_window[i,j,BETA] - self.active_window[i,j,
                        GAMA])/ALPHA))
508                 #print low
509                 high = int(np.floor((self.active_window[i,j,BETA] + self.active_window[i,
                        j,GAMA])/ALPHA))
510                 #print high
511                 #print "Updating sectors [{:d} {:d}] for cell ({:d}, {:d})".format(low,
                        high, i, j)
512                 k_range = [x%HIST_SIZE for x in np.linspace(low, high, high-low+1, True,
                        dtype = int)]
513                 #print k_range
514                 for k in k_range:
515                     assert k < HIST_SIZE and k >= 0, "Error for polar histogram index: %d
                            on i = %d, j = %d" % (k, i, j)
516                     self.polar_hist[k] += self.active_window[i, j, MAG]
```

```
517
518         #return self.polar_hist
519
520     def update_bin_polar_histogram(self):
521         r"""Calcula el histograma polar binario.
522
523         El cálculo se hace a partir del
524         :attr:`histograma polar<polar_hist>` y los valores
525         de umbral :const:`T_HI` y :const:`T_LO`. Cada sector
526         se indica como bloqueado (``True``) o libre (``False``).
527
528         Modifica los siguientes atributos de clase:
529          * :attr:`bin_polar_hist`
530
531         """
532
533         for k in xrange(HIST_SIZE):
534             if self.polar_hist[k] > T_HI:
535                 blocked = True
536             elif self.polar_hist[k] < T_LO:
537                 blocked = False
538             else:
539                 blocked = self.bin_polar_hist[k]
540
541             self.bin_polar_hist[k] = blocked
542
543     def update_masked_polar_hist(self,steer_l,steer_r):
544         r"""Calcula el histograma polar mascarado.
545
546         El cálculo se hace a partir del
547         :attr:`histograma polar binario<bin_polar_hist>` y
548         el radio de giro mínimo en el momento del cálculo.
549         Las celdas ocupadas de la ventana activa bloquean
550         sectores adicionales dependiendo de la dirección actual
551         del robot y su capacidad de giro.
552
553         Parameters
554         ----------
555         steer_l : float
556             Radio de giro mínimo del robot hacia la izquierda.
557         steer_r : float
558             Radio de giro mínimo del robot hacia la derecha.
559
560         Notes
561         -----
562         Modifica los siguientes atributos de clase:
563          * :attr:`masked_polar_hist`
564         """
565
566
567         phi_back = self.cita + 180.0
568         phi_back = phi_back - 360.0 if phi_back > 360.0 else phi_back
569
570         phi_left = phi_back
571         phi_right = phi_back
572
573         X_r = WINDOW_SIZE*RESOLUTION/2.0
574         Y_r = WINDOW_SIZE*RESOLUTION/2.0
575
```

```
576         print "INFO: Current dir is {:.3f}".format(self.cita)
577         print "INFO: phi_back is {:.3f}".format(phi_back)
578         for i in xrange(WINDOW_SIZE):
579             for j in xrange(WINDOW_SIZE):
580
581                 if self.active_window[i,j,MAG] <= 2*(C_MAX**2) or (i == WINDOW_CENTER and
                        j == WINDOW_CENTER):
582                     #print "Skipped cel ({},{}) = {:.3f}".format(i,j,self.polar_hist[k])
583                     continue
584
585                 if self.active_window[i,j,BETA] == self.cita:
586                     #print "Entering cell ({:d},{:d}) dead ahead".format(i,j)
587
588                     # position of the cell
589                     cij_x = (i+0.5)*RESOLUTION
590                     cij_y = (j+0.5)*RESOLUTION
591
592                     #### Left steering radius
593                     r_robl_x = steer_l*np.cos(np.radians(self.cita+90.0))
594                     r_robl_y = steer_l*np.sin(np.radians(self.cita+90.0))
595                     # center of the steering radius in the active window
596                     r_steer_x = X_r + r_robl_x
597                     r_steer_y = Y_r + r_robl_y
598
599                     c_dist2 = np.square(cij_x - r_steer_x) + np.square(cij_y - r_steer_y)
600                     #print "Left dist {:.3f}".format(c_dist2)
601                     if c_dist2 < np.square(R_RS + steer_l):
602                         phi_left = self.active_window[i,j,BETA] + 0.1
603                         print "Setting left limit angle to {:.1f} on cell ({:d},{:d}) =
                                {:.1f}".format(phi_left, i, j, self.active_window[i,j,MAG])
604
605                     #### Right steering radius
606                     r_robr_x = steer_r*np.cos(np.radians(self.cita-90.0))
607                     r_robr_y = steer_r*np.sin(np.radians(self.cita-90.0))
608                     # center of the steering radius in the active window
609                     r_steer_x = X_r + r_robr_x
610                     r_steer_y = Y_r + r_robr_y
611
612                     c_dist2 = np.square(cij_x - r_steer_x) + np.square(cij_y - r_steer_y)
613                     #print "Right dist {:.3f}, limit {:.3f}".format(c_dist2,np.square(
                            R_RS + steer_r))
614                     if c_dist2 < np.square(R_RS + steer_r):
615                         phi_right = self.active_window[i,j,BETA] - 0.1
616                         print "Setting right limit angle to {:.1f} on cell ({:d},{:d})".
                                format(phi_right, i, j)
617
618
619                 elif self._isInRange(self.cita, phi_left, self.active_window[i,j,BETA]):
620                     #left
621                     #print "Entering cell ({:d},{:d}) in left range".format(i,j)
622                     r_robl_x = steer_l*np.cos(np.radians(self.cita+90.0))
623                     r_robl_y = steer_l*np.sin(np.radians(self.cita+90.0))
624
625                     # center of the steering radius in the active window
626                     r_steer_x = X_r + r_robl_x
627                     r_steer_y = Y_r + r_robl_y
628
629                     # position of the cell
630                     cij_x = (i+0.5)*RESOLUTION
```

```
631                     cij_y = (j+0.5)*RESOLUTION
632
633                     # distance^2 from the cell to the steering center
634                     c_dist2 = np.square(cij_x - r_steer_x) + np.square(cij_y - r_steer_y)
635
636                     if c_dist2 < np.square(R_RS + steer_l):
637                         phi_left = self.active_window[i,j,BETA]
638                         print "Setting left limit angle to {:.1f} on cell ({:d},{:d}) =
                                {:.1f}".format(phi_left, i, j, self.active_window[i,j,MAG])
639
640                 elif self._isInRange(phi_right,self.cita,self.active_window[i,j,BETA]):
641                     #right
642                     #print "Entering cell ({:d},{:d}) in right range".format(i,j)
643                     r_robr_x = steer_r*np.cos(np.radians(self.cita-90.0))
644                     r_robr_y = steer_r*np.sin(np.radians(self.cita-90.0))
645
646                     # center of the steering radius in the active window
647                     r_steer_x = X_r + r_robr_x
648                     r_steer_y = Y_r + r_robr_y
649
650                     # position of the cell
651                     cij_x = (i+0.5)*RESOLUTION
652                     cij_y = (j+0.5)*RESOLUTION
653
654                     # distance^2 from the cell to the steering center
655                     c_dist2 = np.square(cij_x - r_steer_x) + np.square(cij_y - r_steer_y)
656
657                     if c_dist2 < np.square(R_RS + steer_r):
658                         phi_right = self.active_window[i,j,BETA]
659                         print "Setting right limit angle to {:.1f} on cell ({:d},{:d})".
                                format(phi_right, i, j)
660
661
662         print "Limit angles:"
663         print "Left:", phi_left
664         print "Right:", phi_right
665         print
666         for k in xrange(HIST_SIZE):
667             if self.bin_polar_hist[k] == False and self._isInRange(phi_right,phi_left,k*
                    ALPHA):
668                 self.masked_polar_hist[k] = False
669             else:
670                 self.masked_polar_hist[k] = True
671
672     # This function determines if an angle in the range
673     # [0, 360[ is inside the sector given, from start to
674     # end (counter-clockwise), also angles in the range [0, 360[
675     @staticmethod
676     def _isInRange(start,end,angle):
677         if start < end:
678             return (start <= angle and angle <= end)
679         else:
680             return (start <= angle and angle <= 360) or (0 <= angle and angle <= end)
681
682
683     def find_valleys(self):
684         r"""Analiza el histograma polar mascarado y determina los
685         valles candidatos.
686
```

```
687         Los valles se obtienen a partir del
688         :attr:`histograma polar mascarado<masked_polar_hist>`.
689
690         Returns
691         -------
692         int
693              * ``-1`` si no se encuentra ningún sector
694                bloqueado en el hisotgrama polar mascarado.
695              * De lo contrario, el número de valles encontrados.
696         """
697
698         start = None
699         for x in xrange(HIST_SIZE):
700             if self.masked_polar_hist[x]:
701                 start = x
702                 break
703
704         # If no value was found over the threshold no action
705         # needs to be taken since there are no nearby obstacles
706         if start == None:
707             return -1
708
709         # Else, look for valleys after 'start'
710         # True means blocked in the masked histogram
711         # False means free
712         self.valleys = []
713         valley_found = False
714         for i in xrange(HIST_SIZE+1):
715             index = (start + i)%HIST_SIZE
716             if not valley_found:
717                 if not self.masked_polar_hist[index]:
718                     v_start = index
719                     valley_found = True
720
721             else:
722                 if self.masked_polar_hist[index]:
723                     self.valleys.append(tuple([v_start, index-1]))
724                     valley_found = False
725         return len(self.valleys)
726
727     def calculate_steering_dir(self, valley_count):
728         r"""Calcula la dirección de movimiento para la evasión y
729         la rapidez del robot.
730
731         El cálculo de la dirección para el robot se hace a
732         partir de los valles candidato, la orientación actual
733         y el objetivo. Si no hay objetivo, se toma la dirección
734         actual de movimiento como objetivo.
735         Se usa una función de costo para seleccionar la dirección
736         de entre todas las candidatas. La rapidez se calcula del
737         costo de la dirección seleccionada, a mayor costo más
738         lento se moverá el robot.
739
740         Parameters
741         ----------
742         valley_count : int
743             El resultado del llamado a :func:`find_valleys`.
744
745         Returns
```

```
746         -------
747         new_dir  : float
748             La dirección del movimiento, dada en grados en el
749             rango  :math:`[0°,\:360°[` .
750         V : float
751             La velocidad del robot.
752 
753         Raises
754         ------
755         Exception
756             Si ``valley_count`` es cero (todas las direcciones
757             están bloqueadas).
758 
759         Notes
760         -----
761 
762         Los valles angostos definen una única dirección
763         candidata, el medio del valle. Los anchos pueden definir
764         hasta tres: los bordes del valle y la dirección del
765         objetivo si esta se encuentra dentro del valle.
766 
767         """
768 
769         if valley_count == 0:
770             raise Exception("No candidate valleys found to calculate a new direction")
771 
772         t_dir = None
773         if self.target != None:
774             # If there is a target for the robot we calculate the direction
775             t_dir = np.degrees(np.arctan2(self.target[1]-self.y_0, self.target[0]-self.
                    x_0))
776             t_dir = t_dir if t_dir >= 0 else t_dir + 360
777         else:
778             # Else set the current direction as the target
779             t_dir = self.cita
780 
781 
782         candidate_dirs = []
783 
784         print self.valleys
785         if valley_count == -1:
786             print "No obstacles nearby, setting route to target at {:.1f}".format(t_dir)
787             candidate_dirs.append(t_dir)
788         else:
789             for v in self.valleys:
790                 s1, s2 = v
791                 v_size = (s2 - s1) if s2 >= s1 else HIST_SIZE - (s1-s2)
792 
793                 if v_size < WIDE_V:
794                     # Narrow valley
795                     # The only target dir is the middle of
796                     # the oppening
797                     print "narrow valley"
798                     c_center = ALPHA*(s1 + v_size/2.0)
799                     c_center = c_center - 360.0 if c_center >= 360.0 else c_center
800                     candidate_dirs.append(c_center)
801 
802                 else:
803                     # Wide valley
```

```
804                     # Target dirs are the left and right
805                     # borders,
806                     print "wide valley"
807                     c_right = ALPHA*(s1 + WIDE_V/2.0)
808                     c_right = c_right - 360.0 if c_right >= 360.0 else c_right
809
810                     c_left = ALPHA*(s2 - WIDE_V/2.0)
811                     c_left = c_left + 360.0 if c_left < 0.0 else c_left
812
813                     candidate_dirs.append(c_left)
814                     candidate_dirs.append(c_right)
815
816                     if c_right != c_left and self._isInRange(c_right,c_left,t_dir):
817                         candidate_dirs.append(t_dir)
818
819         print candidate_dirs
820         # Once all we know all possible candidate dirs
821         # choose the one with the lowest cost
822         new_dir = None
823         best_cost = None
824         for c in candidate_dirs:
825             cost = mu1*self._abs_angle_diff(c,t_dir) + \
826                     mu2*self._abs_angle_diff(c,self.cita) + \
827                     mu3*self._abs_angle_diff(c,self.prev_dir)
828             print "For candidate dir {:.1f}: {:.3f} cost".format(c,cost)
829
830             if best_cost == None:
831                 new_dir = c
832                 best_cost = cost
833             elif cost < best_cost:
834                 new_dir = c
835                 best_cost = cost
836
837         self.prev_dir = new_dir
838         self.prev_cost = best_cost
839
840         V = V_MAX*(1 - best_cost/MAX_COST) + V_MIN
841
842         print "Setting dir to {:.1f}".format(new_dir)
843         return new_dir, V
844
845     @staticmethod
846     def _abs_angle_diff(a1, a2):
847         return min(360.0 - abs(a1 - a2), abs(a1 - a2))
848
849     def calculate_speed(self):
850
851         V = V_MAX*(1 - self.prev_cost/MAX_COST) + V_MIN
852         return V
853
854     @staticmethod
855     def _dist(array,i,j):
856         n_dist = abs(i-j)
857         return min(n_dist, len(array) - n_dist)
858
859
860 def main():
861     np.set_printoptions(precision=2)
862     robot = VFHPModel()
```

```
863     print "Obstacle grid"
864     print robot.obstacle_grid, "\n"
865     print "Active Window angles"
866     print robot.active_window[:,:,BETA], "\n"
867     print "Active Window squared distances"
868     print robot.active_window[:,:,DIST2], "\n"
869     print "Active Window a-bd^2 constants"
870     print robot.active_window[:,:,ABDIST], "\n"
871     print "Max distance squared: %f" % D_max2
872 
873 
874     print("Updating the obstacle grid and robot position")
875 
876     robot.update_position(1.5,1.5,90.0)
877 
878     #robot.obstacle_grid[1,6] = 1
879     #robot.obstacle_grid[1,5] = 2
880     #robot.obstacle_grid[1,4] = 2
881     #robot.obstacle_grid[1,3] = 5
882     #robot.obstacle_grid[2,2] = 3
883     #robot.obstacle_grid[3,2] = 3
884     #robot.obstacle_grid[4,2] = 3
885 #
886 #     robot.obstacle_grid[9,2] = 4
887 #     robot.obstacle_grid[9,3] = 5
888 #     robot.obstacle_grid[9,4] = 6
889 #     robot.obstacle_grid[9,5] = 5
890 #     robot.obstacle_grid[9,6] = 4
891 
892     robot.obstacle_grid[27,30] = 20
893     robot.obstacle_grid[27,31] = 20
894     robot.obstacle_grid[27,29] = 20
895     robot.obstacle_grid[28,30] = 20
896     robot.obstacle_grid[26,30] = 20
897 
898     robot.obstacle_grid[30,37] = 20
899     robot.obstacle_grid[31,37] = 20
900     robot.obstacle_grid[29,37] = 20
901     robot.obstacle_grid[30,38] = 20
902     robot.obstacle_grid[30,36] = 20
903 
904     robot.obstacle_grid[41,30] = 20
905     robot.obstacle_grid[40,30] = 20
906     robot.obstacle_grid[42,30] = 20
907     robot.obstacle_grid[41,31] = 20
908     robot.obstacle_grid[41,29] = 20
909 
910     print "i , j , k "
911     print robot.i_0, robot.j_0, robot.k_0
912     print robot._active_grid(), "\n"
913 
914     print "Simulating a set of sensor readings"
915     pseudo_readings = np.float_([[0.2, np.radians(x)] for x in range(0,90,2)])
916     #pseudo_readings = np.float_([[0.3,
917     robot.update_obstacle_density(pseudo_readings)
918 
919 
920     print "Updating the active window"
921     robot.update_active_window()
```

```
922     print robot.active_window[:, :, MAG], "\n"
923 
924     print "Updating polar histogram"
925     robot.update_polar_histogram()
926     print robot.polar_hist, "\n"
927 
928     print "Updating binary histogram"
929     robot.update_bin_polar_histogram()
930     print robot.bin_polar_hist, "\n"
931 
932     print "Updating masked polar histogram"
933     robot.update_masked_polar_hist(R_ROB*1,R_ROB*1)
934     print robot.masked_polar_hist, "\n"
935 
936     print "Looking for valleys"
937     robot.find_valleys()
938     print robot.valleys, "\n"
939 
940     print "Select new direction"
941     robot.prev_dir = 90.0
942     print robot.calculate_steering_dir(), "\n"
943 
944     print "Setting speed to (MAX = {:.2f})".format(V_MAX)
945     print robot.calculate_speed(), "\n"
946 
947 
948 #     print "Updating filtered histogram"
949 #     robot.update_filtered_polar_histogram()
950 #     print robot.filt_polar_hist, "\n"
951 #
952 #     print "Looking for valleys"
953 #     robot.find_valleys()
954 #     print robot.valleys, "\n"
955 #
956 #     try:
957 #         print "Setting steer direction"
958 #         cita = robot.calculate_steering_dir()
959 #         print cita, "\n"
960 #     except:
961 #         pass
962 
963     ### Figuras y graficos ###
964 #
965 #     plt.figure(2)
966 #     plt.plot(i, robot.filt_polar_hist ) #, 0.1, 0, color='b')
967 #     plt.title("Histograma polar filtrado")
968 
969     plt.figure(1)
970     plt.pcolor(robot._active_grid().T, alpha=0.75, edgecolors='k',vmin=0,vmax=20)
971     plt.xlabel("X")
972     plt.ylabel("Y", rotation='horizontal')
973 
974     plt.figure(2)
975     x = [ALPHA*x for x in range(len(robot.polar_hist))]
976     i = [a for a in range(len(robot.polar_hist))]
977     plt.bar(x, robot.polar_hist, 3.0, 0, color='r')
978     plt.title("Histograma polar")
979 
980     plt.figure(3)
```

```
981     x = [ALPHA*x for x in range(len(robot.polar_hist))]
982     i = [a for a in range(len(robot.polar_hist))]
983     plt.bar(x, robot.bin_polar_hist, 3.0, 0, color='b')
984     plt.title("Histograma polar binario")
985 
986     plt.figure(4)
987     x = [ALPHA*x for x in range(len(robot.polar_hist))]
988     i = [a for a in range(len(robot.polar_hist))]
989     plt.bar(x, robot.masked_polar_hist, 3.0, 0, color='g')
990     plt.title("Histograma polar mascarado")
991 
992     plt.show()
993 
994 if __name__ == "__main__":
995     main()
```

A.4. Modelo Robot Diferencial

```
1 
2  # coding=utf8
3 
4  import sys
5  import numpy as np
6  import PID
7  import VFH
8  import VFHP
9  import Braitenberg as bra
10 
11 r"""
12 .. ::module DiffRobot
13 
14 Este módulo define un modelo para el robot diferencial, que sirve de
15 interfaz entre los controladores de evasión y los motores.
16 
17 """
18 
19 TARGET_X = 2.5
20 TARGET_Y = 3.5
21 
22 M_VFH = 0
23 """int: Modo de operación con VFH.
24 """
25 
26 M_BRAIT = 1
27 """int: Modo de operación con vehículo de Braitenberg.
28 """
29 
30 M_VFHP = 2
31 """int: Modo de operación con VFH+.
32 """
33 
34 # Robot differencial
35 class DiffModel(object):
36     r"""Clase para el modelo del robot diferencial.
37 
38     Define un objeto que representa el robot diferencial, y permite
39     abstraer la cinemática (comandos para los motores) a comandos
40     de alto nivel.
```

```
41
42      Parameters
43      ----------
44      r : float, opcional
45          Radio de las ruedas (m).
46      b : float, opcional
47          Distancia entre las ruedas (m).
48      wm_max : float, opcional
49          Máxima velocidad angular de los motores (rad/s).
50      c_type : {:const:‘M_VFH‘, :const:‘M_BRAIT‘, :const:‘M_VFHP‘}, opcional
51          Tipo de controlador para la evasión de obstáculos.
52
53      Attributes
54      ----------
55      r : float
56          Radio de las ruedas (m).
57      b : float
58          Distancia entre las ruedas (m).
59      wm_max : float
60          Máxima velocidad angular del motor (rad/s).
61      omega_max : float
62          Máxima velocidad angular del robot (velocidad de giro, en rad/s).
63      v_max : float
64          Máxima velocidad lineal del robot (m/s).
65      model : Controlador
66          Una instacia de un controlador para evasión, se escoge a partir de ‘‘c_type‘‘.
67
68      v_ref : float
69          Valor de velocidad lineal de referencia para el movimiento del robot, definido
                por
70          el controlador :attr:‘model‘.
71      w_ref : float
72          Valor de velocidad angular de referencia para el movimiento del robot (rad/s).
                Definido por
73          el controlador :attr:‘model‘.
74      w_prev : float
75          Velocidad angular anterior (rad/s). Usado por el controlador VFH para calcular
76          la velocidad angular actual.
77
78      cita : float
79          Dirección actual del robot (rad).
80      cita_prev : float
81          Dirección anterior del robot (rad), usado por el controlador VFH para calcular la
82          velocidad angular actual.
83      cita_ref : float
84          Dirección de referencia para el movimiento del robot (rad). Se usa en el caso de
                los
85          controladores VFH y VFH+, luego un PID determina la velocidad angular requerida
86          :attr:‘w_ref‘ para alcanzar esta dirección.
87
88      left_motor : float
89          Velocidad angular del motor izquierdo (rad/s).
90      right_motor : float
91          Velocidad angular del motor derecho (rad/s).
92
93      angle_controller : :class:‘PID.PID‘
94          Un controlador PID que se usa con los algoritmos VFH y VFH+ para determinar la
95          velocidad angular de referencia :attr:‘w_ref‘ a partir de la dirección de
                referencia
```

```
96          :attr:`cita_ref`.
97      """
98
99      def __init__(self, r=0.02, b=0.05, wm_max=6.28, c_type = 0):
100         # r = radio de las ruedas [m]
101         self.r = r
102
103         # b = distancia entre las ruedas [m]
104         self.b = b
105
106         # wm_max = maxima velocidad angular de rotacion de las ruedas
107         # omega max = maxima velocidad angular de rotacion del robot
108         self.wm_max = wm_max
109         self.omega_max = 0.9*r*wm_max/b
110         self.v_max = 0.9*r*wm_max
111
112         self.model = None
113         self.c_type = c_type
114         if c_type == M_VFH:
115             print "Using VFH algorithm"
116             VFH.OMEGA_MAX = self.omega_max
117             VFH.V_MAX = self.v_max
118             VFH.V_MIN = self.v_max*0.2
119             self.model = VFH.VFHModel()
120         elif c_type == M_BRAIT:
121             print "Using Braitenberg algorithm"
122             self.model = bra.BraitModel(bra.SMODE_FULL, 0.05, 0.3, -self.v_max*0.1, self.
                    v_max, self.omega_max)
123         elif c_type == M_VFHP:
124             print "Using VFH+ algorithm"
125             self.model = VFHP.VFHPModel()
126             VFHP.OMEGA_MAX = self.omega_max
127             VFHP.V_MAX = self.v_max
128             VFHP.V_MIN = self.v_max*0.2
129             self.model = VFHP.VFHPModel()
130
131
132         # Valores deseados de velocidad, velocidad angular y
133         # orientacion
134         self.v_ref = 0.0
135         self.w_ref = 0.0
136         self.w_prev = 0.0
137
138         self.cita = 0.0
139         self.cita_prev = 0.0
140         self.cita_ref = 0.0
141
142         # Velocidades de los motores
143         self.left_motor = 0.0
144         self.right_motor = 0.0
145
146         # Controlador PI para la orientacion del robot
147         # Kp = 0.808, Ti = 3.5 (sintonizacion kogestad)
148         self.angle_controller = PID.PID(1.2, 10.0, 0.0, self.omega_max, -self.omega_max)
149         self.angle_controller.begin(0.0)
150
151     # Todas las mediciones se dan en el sistema metrico [m, rad]
152     def set_initial_pos(self, x, y, cita):
153         r"""Define la postura inicial del robot.
```

```
154
155         Indica las condiciones iniciales :math:`(x,\; y, \; \theta)` del robot.
156
157         Parameters
158         ----------
159         x : float
160             Posición absoluta del robot sobre el eje :math:`x`
161             en metros.
162         y : float
163             Posición absoluta del robot sobre el eje :math:`y`
164             en metros.
165         cita : float
166             Orientación del robot resepecto el eje :math:`z`
167             en radianes.
168
169         Notes
170         -----
171         Modifica el estado interno del controlador :attr:`model`.
172
173         """
174
175         if self.c_type == M_VFH:
176             self._set_initial_pos_VFH(x,y,cita)
177         elif self.c_type == M_BRAIT:
178             self._set_initial_pos_Brait(x,y,cita)
179         elif self.c_type == M_VFHP:
180             self._set_initial_pos_VFHP(x,y,cita)
181         else:
182             print "ERROR: no control type defined!"
183
184
185     def _set_initial_pos_VFH(self, x, y, cita):
186         self.cita = cita
187         self.cita_prev = cita
188         self.w_prev = 0.0
189         self.model.update_position(x, y, np.degrees(cita))
190         self.angle_controller.setInput(cita)
191         self.angle_controller.setRef(cita)
192
193     def _set_initial_pos_Brait(self, x, y, cita):
194         self.model.UpdatePos(x, y, cita)
195
196     def _set_initial_pos_VFHP(self, x, y, cita):
197         self.cita = cita
198         #self.cita_prev = cita
199         #self.w_prev = 0.0
200         self.model.update_position(x, y, np.degrees(cita))
201         self.angle_controller.setInput(cita)
202         self.angle_controller.setRef(cita)
203
204     def set_target(self, x, y):
205         r"""Indica un punto objetivo.
206
207         Habilita el seguimiento de trayectorias e indica
208         al robot el punto al cual debe dirigirse.
209
210         Parameters
211         ----------
212         x : float
```

```
213             Posición absoluta del objetivo sobre el eje :math:‘x‘.
214         y : float
215             Posición absoluta del objetivo sobre el eje :math:‘y‘.
216 
217         Notes
218         -----
219         Modifica el estado interno del controlador :attr:‘model‘.
220         """
221 
222         if self.c_type == M_VFH:
223             self.model.set_target(x,y)
224         elif self.c_type == M_BRAIT:
225             self.model.SetTarget(x,y)
226         elif self.c_type == M_VFHP:
227             self.model.set_target(x,y)
228         else:
229             print "ERROR: no control type defined!"
230 
231     def unset_target(self):
232         r"""Dehabilita el seguimiento de trayectorias.
233 
234 
235         Notes
236         -----
237         Modifica el estado interno del controlador :attr:‘model‘.
238         """
239         if self.c_type == M_VFH:
240             self.model.set_target()
241         elif self.c_type == M_BRAIT:
242             self.model.SetTarget(0,0,True)
243         elif self.c_type == M_VFHP:
244             self.model.set_target()
245         else:
246             print "ERROR: no control type defined!"
247 
248     def update_pos(self, x, y, cita, delta_t):
249         r"""Actualiza la posición del robot.
250 
251         Parameters
252         ----------
253         x : float
254             Posición absoluta del robot sobre el eje :math:‘x‘.
255         y : float
256             Posición absoluta del robot sobre el eje :math:‘y‘.
257         cita : float
258             Orientación del robot resepecto el eje :math:‘z‘
259             en radianes.
260         delta_t : float
261             El tiempo transcurrido desde la última actualización.
262 
263         Notes
264         -----
265         Modifica el estado interno del controlador :attr:‘model‘,
266         además del atributo :attr:‘cita‘. En el caso de VFH y VFH+
267         realiza un paso de integración del PID.
268         """
269 
270         if self.c_type == M_VFH:
271             self._update_pos_VFH(x,y,cita,delta_t)
```

```
272         elif self.c_type == M_BRAIT:
273             self._update_pos_Brait(x,y,cita,delta_t)
274         elif self.c_type == M_VFHP:
275             self._update_pos_VFHP(x,y,cita,delta_t)
276         else:
277             print "ERROR: no control type defined!"
278 
279     def _update_pos_VFH(self, x, y, cita, delta_t):
280 
281         self.cita_prev = self.cita
282         self.cita = cita
283         self.w_prev = 0.7*PID.angleDiff(self.cita, self.cita_prev)/delta_t + 0.3*self.
                w_prev
284 
285         # El controlador VFH requiere angulos en grados
286         self.model.update_position(x, y, np.degrees(cita))
287         self.angle_controller.setInput(cita)
288         self.w_ref = self.angle_controller.timestep(delta_t)
289         print "PID readings: effort %f, error %f, acumulated_error %f " % (self.w_ref ,
                self.angle_controller.error, self.angle_controller._integral)
290 
291     def _update_pos_Brait(self, x, y, cita, delta_t):
292         self.cita = cita
293         self.model.UpdatePos(x, y, cita)
294 
295     def _update_pos_VFHP(self, x, y, cita, delta_t):
296 
297         self.cita = cita
298         self.model.update_position(x, y, np.degrees(cita))
299 
300         self.angle_controller.setInput(cita)
301         self.w_ref = self.angle_controller.timestep(delta_t)
302         print "PID readings: effort %f, error %f, acumulated_error %f " % (self.w_ref ,
                self.angle_controller.error, self.angle_controller._integral)
303 
304     def update_readings(self, data):
305         r"""Procesa las lecturas de un sensor.
306 
307         Actualiza el estado del controlador :attr:`model`
308         con las lecturas de un sensor de distancia.
309 
310         Parameters
311         ----------
312         data : ndarray
313             Una estructura de datos tipo ``numpy.ndarray`` que
314             contiene las lecturas de un sensor de distancias.
315             Debe ser un arreglo de dimensiones
316             :math:`(n \times 2)` para :math:`n` puntos o
317             lecturas, donde cada par representa una coordenada
318             polar :math:`(r,\theta)` respecto al marco de
319             referencia del robot, dada en metros y radianes.
320 
321         """
322         if self.c_type == M_VFH:
323             self._update_readings_VFH(data)
324         elif self.c_type == M_BRAIT:
325             self._update_readings_Brait(data)
326         elif self.c_type == M_VFHP:
327             self._update_readings_VFHP(data)
```

```
328         else:
329             print "ERROR: no control type defined!"
330 
331     def _update_readings_VFH(self, data):
332         try:
333             self.model.update_obstacle_density(data)
334         except Exception as e:
335             print "Exception caught during sensor update:"
336             print e
337 
338     def _update_readings_Brait(self, data):
339         self.model.UpdateSensors(data)
340 
341     def _update_readings_VFHP(self, data):
342         try:
343             self.model.update_obstacle_density(data)
344         except Exception as e:
345             print "Exception caught during sensor update:"
346             print e
347         pass
348 
349     def update_target(self):
350         r"""Aplica la acción de control para evasión de obstáculos.
351 
352         Actualiza el estado interno de :attr:`model` para obtener
353         la acción de control :math:`(v_{ref}, w_{ref})`. Luego,
354         la convierte en acciones individuales sobre cada motor
355         mediante el método :meth:`setMotorSpeed`.
356 
357         Notes
358         -----
359         Modifica los siguientes atributos de clase:
360 
361         .. hlist::
362 
363             * :attr:`v_ref`
364             * :attr:`cita_ref`
365             * :attr:`w_ref`
366             * :attr:`left_motor`
367             * :attr:`right_motor`
368         """
369         if self.c_type == M_VFH:
370             self._update_target_VFH()
371         elif self.c_type == M_BRAIT:
372             self._update_target_Brait()
373         elif self.c_type == M_VFHP:
374             self._update_target_VFHP()
375         else:
376             print "ERROR: no control type defined!"
377 
378     def _update_target_VFH(self):
379         self.model.update_active_window()
380         self.model.update_polar_histogram()
381         self.model.update_filtered_polar_histogram()
382 
383         if self.model.find_valleys() != -1:
384             try:
385                 self.cita_ref = np.radians(self.model.calculate_steering_dir())
386             except Exception as e:
```

```
387                 print "No valleys found, keeping current dir"
388                 self.cita_ref = np.radians(self.model.cita)
389 
390             self.v_ref = self.model.calculate_speed(self.w_prev)
391             print "Obstacle detected! target dir is %f" % self.cita_ref
392         else:
393             print "No nearby obstacles keeping current dir"
394             self.cita_ref = np.radians(self.model.cita)
395             print "Setting target dir at %f" % self.cita_ref
396             #self.w_ref = 0.0
397             self.v_ref = self.v_max
398 
399         # Se actualiza el valor deseado de orientacion
400         # en el controlador
401         self.angle_controller.setRef(self.cita_ref)
402         self.setMotorSpeed()
403 
404     def _update_target_Brait(self):
405         if self.model.target == None:
406             v, w = self.model.Evade2b()
407         else:
408             v, w = self.model.Mixed2b3a()
409         self.v_ref = v
410         self.w_ref = w
411         self.setMotorSpeed()
412 
413     def _update_target_VFHP(self):
414         self.model.update_active_window()
415         self.model.update_polar_histogram()
416         self.model.update_bin_polar_histogram()
417         self.model.update_masked_polar_hist(self.b,self.b)
418 
419         try:
420             result = self.model.find_valleys()
421             cita, v = self.model.calculate_steering_dir(result)
422         except Exception as e:
423             print "Exception caught on VHF+ control loop"
424             print e
425             cita = self.cita_ref
426             v = self.v_ref
427 
428         self.v_ref = v
429         self.cita_ref = np.radians(cita)
430         self.angle_controller.setRef(self.cita_ref)
431         self.setMotorSpeed()
432 
433     def setMotorSpeed(self):
434         r"""Determina la velocidad de los motores.
435 
436         Determina la velocidad individual que debe tener cada
437         motor (en rad/s), según el modelo cinemático inverso, para
438         que el robot alcance la velocidad lineal y angular de
439         referencia :attr:`v_ref` y :attr:`w_ref`.
440 
441         Notes
442         -----
443         Modifica los siguientes atributos de clase:
444 
445         .. hlist::
```

```
446
447             * :attr:`left_motor`
448             * :attr:`right_motor`
449         """
450         self.left_motor = (self.v_ref - self.b*self.w_ref)/self.r
451         self.right_motor = (self.v_ref + self.b*self.w_ref)/self.r
452
453     def __str__(self):
454         info = "(x=%f, y=%f, g=%f)\n" % (self.model.x_0, self.model.y_0, self.cita)
455         ref = "(g_ref=%f, w_ref=%f, v_ref=%f)" % (self.cita_ref, self.w_ref, self.v_ref)
456         struct = "Radio de las ruedas: %f\nDistancia entre las ruedas: %f\n" %(self.r,
                self.b)
457         actu = "Vel Motores: [%f , %f]\n" % (self.left_motor, self.right_motor)
458         return "%s%s%s%s" % (struct, info, ref, actu)
459
460     def __print__(self):
461         print str(self)
462
463
464
465 def main():
466     # simulamos 50 ms
467     dt = 0.5
468
469     robot = DiffModel(c_type=M_VFHP)
470     robot.set_initial_pos(2.5, 2.5, 0)
471     robot.update_pos(2.5,2.5,0,dt)
472
473     pseudo_readings = np.float_([[0.25, np.radians(x)] for x in range(0,220,1)])
474     robot.update_readings(pseudo_readings)
475     robot.update_target()
476
477     robot.update_pos(2.5,2.5,0,dt)
478     robot.update_target()
479
480     print robot.model._active_grid()
481
482     print robot
483
484 if __name__ == "__main__":
485     main()
```

A.5. Controlador PID sencillo

```
1  import numpy as np
2  import time
3
4  class PID:
5
6      def __init__(self, Kp, Ti, Td, MAX, MIN):
7          self.Kp = np.float_(Kp)
8          self.Ti = np.float_(Ti)
9          self.Td = np.float_(Td)
10         self.max = np.float_(MAX)
11         self.min = np.float_(MIN)
12
13         self.ref = np.float_(0)
14
```

```
15          self.error = np.float_(0)
16          self._integral = np.float_(0)
17          self._derivada = np.float_(0)
18          self.effort = np.float(0)
19
20          self.past_error = np.float_(0)
21          self.past_input = np.dtype(np.float_)
22
23      def setRef(self, ref):
24          self.ref = np.float_(ref)
25
26      def setInput(self, inp):
27          self.input = np.float_(inp)
28
29      def begin(self,entrada):
30          self.past_input = np.float_(entrada)
31
32      def timestep(self, delta_t):
33
34          # Luego se determina el error, el termino integral y el derivativo
35          self.error = angleDiff(self.ref, self.input)
36          self._derivada = -angleDiff(self.input, self.past_input)/delta_t
37          self._integral = self._integral + (self.error + self.past_error)*delta_t/2
38
39
40          # Se calcula el esfuerzo del controlador
41          self.effort = self.Kp*(self.error + self._integral/self.Ti + self._derivada*self.
                Td)
42          if self.effort > self.max:
43              self.effort = self.max
44          elif self.effort < self.min:
45              self.effort = self.min
46
47          # Se actualizan los valores pasados
48          self.past_error = self.error
49          self.past_input = self.input
50          return self.effort
51
52  # Calcula a1 - a2 [radianes]
53  def angleDiff(a1, a2):
54      return np.arctan2(np.sin(a1-a2), np.cos(a1-a2))
```

A.6. Programa de enlace con V-REP

```
1  #!/usr/bin/env python
2  # -*- coding: utf-8 -*-
3
4  import sys
5  import platform
6  import os
7  import time
8  import math
9  import numpy as np
10 import matplotlib.pyplot as plt
11
12 logfile_name = 'HokuyoRob.log'
13 print "Redirecting stdout and stderr to logfile %s" % logfile_name
14 sys.stdout.flush()
```

```
15 logfile = open(logfile_name, 'w')
16 sys.stdout = logfile
17 print "LOG START"
18
19 windows_path = 'C:\\Program Files (x86)\\V-REP3\\V-REP_PRO_EDU\\Obstacle-Avoidance\\Py'
20 linux_path = '/home/daniel/Documents/UCR/XI Semestre/Proyecto/Codigo/Obstacle-Avoidance/
       Py'
21
22 pwd = os.path.dirname(os.path.realpath(__file__))
23 agnostic_path = os.path.abspath(os.path.join(pwd, '..', 'Py'))
24
25 if "Linux" in platform.system():
26     sys.path.insert(0, linux_path)
27 elif "Windows" in platform.system():
28     sys.path.insert(0, windows_path)
29 else:
30     print "Unrecognizable OS"
31     sys.path.insert(0, agnostic_path)
32
33 import Braitenberg as brait
34 import DiffRobot as DR
35 import VFH as vfh
36 import VFHP as vfhp
37
38
39
40 try:
41     import vrep
42 except:
43     print ('--------------------------------------------------------------')
44     print ('"vrep.py" could not be imported. This means very probably that')
45     print ('either "vrep.py" or the remoteApi library could not be found.')
46     print ('Make sure both are in the same folder as this file,')
47     print ('or appropriately adjust the file "vrep.py"')
48
49     print ('--------------------------------------------------------------')
50     print ('')
51
52
53 try:
54     import msvcrt as m
55     def wait():
56         sys.stdout.flush()
57         m.getch()
58
59 except Exception as e:
60     print e
61     print "Defaulting wait function"
62     def wait():
63         sys.stdout.flush()
64         raw_input("")
65
66
67 def runSim(argc, argv):
68
69     ####################################
70     ### First start up the conection  ###
71     ### and parse arguments           ###
72
```

```
73      portNb = 0
74      leftMotorHandle = None
75      rightMotorHandle = None
76      leftSensorHandle = None
77      rightSensorHandle = None
78
79      print "Initializing values"
80      if argc == 9:
81          portNb = int(argv[1])
82          leftMotorHandle = int(argv[2])
83          rightMotorHandle = int(argv[3])
84          leftSensorHandle = int(argv[4])
85          rightSensorHandle = int(argv[5])
86          laserSignalName = str(argv[6])
87          posSignalName = str(argv[7])
88          oriSignalName = str(argv[8])
89      else:
90          print "Arguments error"
91          time.sleep(5)
92          return 0
93
94      clientID = vrep.simxStart("127.0.0.1",portNb,True,True,2000,5)
95
96      ###                                  ###
97      ########################################
98
99      if clientID != -1:
100
101         ### Succesfull Conection!
102         print "%s Conected to V-Rep" % argv[0]
103
104         ##############################
105         ### Global var definition ###
106         ### and initialization    ###
107
108         #robot = vfh.VFHModel()
109         print "Iniciando el robot!"
110
111         modo = DR.M_VFH
112         #modo = DR.M_BRAIT
113         #modo = DR.M_VFHP
114
115         data_filename = "sim_data_{:d}.csv".format(modo)
116         data_dump = open(data_filename, 'w')
117
118
119         robot = DR.DiffModel(c_type=modo)
120
121         # Pista 1
122         robot.set_target(2.5,3.0)
123
124         # Pista 2
125         #robot.set_target(2.6,3.1)
126
127         # Pista 3
128         #robot.set_target(2.75,3.5)
129
130
131         X_TRAS = 2.5
```

```
132         Y_TRAS = 2.5
133         G_TRAS = 0.0
134         robot.set_initial_pos(X_TRAS, Y_TRAS,np.pi/2)
135 
136         # Sin objetivo
137         #target = None
138 
139         # Pista 1
140         target = np.array([2.5-X_TRAS,3.0-Y_TRAS])
141 
142         # Pista 2
143         #target = np.array([2.6-X_TRAS,3.1-Y_TRAS])
144 
145         # Pista 3
146         #target = np.array([2.75-X_TRAS,3.5-Y_TRAS])
147 
148         simTime = 0.0
149 
150         motorSpeeds = [0., 0.]
151         sensorReadings = [0., 0.]
152 
153         # Laser Signal
154         laserReturnCode, laserSignal = vrep.simxGetStringSignal(clientID, laserSignalName
                , vrep.simx_opmode_streaming)
155         if laserReturnCode == vrep.simx_return_ok:
156             print "Laser Signal returned ok on first call, this is unexpeted"
157         elif laserReturnCode == vrep.simx_return_novalue_flag:
158             print "Laser Signal stream opened succesfully!"
159         else:
160             print "ERROR: failed to open the Laser Signal stream"
161             raise Exception("Error while opening %s StringSignal, call returned %d" % (
                    laserSignal, laserReturnCode))
162 
163         # Position Signal
164         posReturnCode, posSignal = vrep.simxGetStringSignal(clientID, posSignalName, vrep
                .simx_opmode_streaming)
165         if posReturnCode == vrep.simx_return_ok:
166             print "Pos Signal returned ok on first call, this is unexpeted"
167         elif posReturnCode == vrep.simx_return_novalue_flag:
168             print "Pos Signal stream opened succesfully!"
169         else:
170             print "ERROR: failed to open the Pos Signal stream"
171             raise Exception("Error while opening %s StringSignal, call returned %d" % (
                    posSignal, posReturnCode))
172 
173         # Orientation Signal
174         oriReturnCode, oriSignal = vrep.simxGetStringSignal(clientID, oriSignalName, vrep
                .simx_opmode_streaming)
175         if oriReturnCode == vrep.simx_return_ok:
176             print "Ori Signal returned ok on first call, this is unexpeted"
177         elif oriReturnCode == vrep.simx_return_novalue_flag:
178             print "Ori Signal stream opened succesfully!"
179         else:
180             print "ERROR: failed to open the Ori Signal stream"
181             raise Exception("Error while opening %s StringSignal, call returned %d" % (
                    oriSignal, oriReturnCode))
182 
183         prevLaserSignal = ""
184         ### End of initialization ###
```

```
185         ##############################
186
187
188         ############################
189         ### Main Simulation Loop ###
190         while (vrep.simxGetConnectionId(clientID) != -1):
191
192
193             ### Read Sensors ###
194
195             laserReturnCode, laserSignal = vrep.simxGetStringSignal(clientID,
                    laserSignalName, vrep.simx_opmode_buffer)
196             posReturnCode, posSignal = vrep.simxGetStringSignal(clientID, posSignalName,
                    vrep.simx_opmode_buffer)
197             oriReturnCode, oriSignal = vrep.simxGetStringSignal(clientID, oriSignalName,
                    vrep.simx_opmode_buffer)
198
199             now = vrep.simxGetLastCmdTime(clientID)
200             if (now == simTime):
201                 #print "Simulation has not advanced, skipping..."
202                 continue
203             print "\nSIMTIME ", now
204             delta_t = (now - simTime)/100.0
205             simTime = now
206
207             ######################
208             ## Process position ##
209             if posReturnCode == vrep.simx_return_ok and oriReturnCode == vrep.
                    simx_return_ok:
210                 print "Pos Signal read"
211                 print "Ori Signal read"
212                 x, y, z = vrep.simxUnpackFloats(posSignal)
213                 alpha, beta, gamma = vrep.simxUnpackFloats(oriSignal)
214                 robot.update_pos(x + X_TRAS, y + Y_TRAS, gamma, delta_t)
215                 #print posData
216                 #print oriData
217                 print "current pos: %f, %f, %f" % (x, y, gamma)
218
219                 #print "Acording to robot: %f, %f, %f" % (robot.model.x_0, robot.model.
                        y_0, robot.model.cita)
220                 #print "Acording to robot: %f, %f, %f" % (robot.model.x, robot.model.y,
                        robot.model.gamma)
221             elif posReturnCode == vrep.simx_return_novalue_flag or oriReturnCode == vrep.
                    simx_return_novalue_flag:
222                 print "Pos Signal didn't have a value ready"
223             else:
224                 print "ERROR: failed to read Pos Signal"
225             ##                    ##
226             ######################
227
228
229             ##########################
230             ## Process Laser Sensor ##
231             if laserReturnCode == vrep.simx_return_ok:
232                 print "Laser Signal read"
233
234                 if laserSignal == prevLaserSignal:
235                     print "No new data in Laser Signal, skipping"
236                 else:
```

```
237                     prevLaserSignal = laserSignal
238                     laserData = vrep.simxUnpackFloats(laserSignal)
239                     if len(laserData)%3 != 0:
240                         print "ERROR: unexpected number of laser data floats"
241 
242                     total_points = len(laserData)/3
243                     laserPoints = np.float_(laserData).reshape((total_points,3))
244 
245                     radians = np.arctan2(laserPoints[:,1],laserPoints[:,0])
246                     dist = np.sqrt(np.square(laserPoints[:,0]) + np.square(laserPoints
                            [:,1]))
247                     new_data = np.vstack((dist,radians)).T
248                     robot.update_readings(new_data)
249 
250             elif laserReturnCode == vrep.simx_return_novalue_flag:
251                 print "Laser Signal didn't have a value ready"
252             else:
253                 print "ERROR: failed to read Laser Signal"
254             ##                        ##
255             ############################
256 
257 
258             ################################
259             ## Main control logic for VFH ##
260             if posReturnCode == vrep.simx_return_ok and \
261                     oriReturnCode == vrep.simx_return_ok and \
262                     laserReturnCode == vrep.simx_return_ok:
263                 robot.update_target()
264 
265             ## Set motor speeds
266             vrep.simxSetJointTargetVelocity(clientID, leftMotorHandle, robot.left_motor,
                    vrep.simx_opmode_oneshot)
267             vrep.simxSetJointTargetVelocity(clientID, rightMotorHandle, robot.right_motor
                    , vrep.simx_opmode_oneshot)
268             ##                               ##
269             ################################
270 
271             ########################
272             ## Dump data to file ##
273             if posReturnCode == vrep.simx_return_ok and oriReturnCode == vrep.
                    simx_return_ok:
274 
275                 if target is None:
276                     pass
277                     csv_line = "{:.5f},{:.5f},{:.5f},{:.5f}\n".format(now,x,y,gamma)
278                     data_dump.write(csv_line)
279                     if now >= 5000:
280                         print "Robot has run for 10 seconds!"
281                         vrep.simxSetJointTargetVelocity(clientID, leftMotorHandle, 0,
                                vrep.simx_opmode_oneshot)
282                         vrep.simxSetJointTargetVelocity(clientID, rightMotorHandle, 0,
                                vrep.simx_opmode_oneshot)
283                         break
284                 else:
285                     d = np.sqrt(np.square(target[0] - x) + np.square(target[1] - y))
286                     csv_line = "{:.5f},{:.5f},{:.5f},{:.5f},{:.5f}\n".format(now,x,y,
                            gamma,d)
287                     data_dump.write(csv_line)
288                     if d < 0.030:
```

```
289                         print "Robot has reached its target!"
290                         vrep.simxSetJointTargetVelocity(clientID, leftMotorHandle, 0,
                                vrep.simx_opmode_oneshot)
291                         vrep.simxSetJointTargetVelocity(clientID, rightMotorHandle, 0,
                                vrep.simx_opmode_oneshot)
292                         break
293             ##                      ##
294             ########################
295
296
297             time.sleep(0.05)
298
299         ### End of Simulation loop ###
300         ##############################
301         vrep.simxSetJointTargetVelocity(clientID, leftMotorHandle, 0.0, vrep.
                simx_opmode_oneshot)
302         vrep.simxSetJointTargetVelocity(clientID, rightMotorHandle, 0.0, vrep.
                simx_opmode_oneshot)
303         time.sleep(0.5)
304
305         vrep.simxFinish(clientID)
306
307         print "\nEnd of Simulation"
308         data_dump.close()
309
310         if robot.c_type == DR.M_VFH:
311             print "\nRobot position and orientation"
312             print robot.model.x_0, robot.model.y_0, robot.model.cita
313
314             print "\nRobot i, j, k"
315             print robot.model.i_0, robot.model.j_0, robot.model.k_0
316
317             print "\nRobot active grid"
318             print robot.model._active_grid()
319
320             print "\nRobot polar histogram"
321             print robot.model.polar_hist
322
323             print "\nRobot filtered histogram"
324             print robot.model.filt_polar_hist
325
326             print "\nValleys"
327             print robot.model.valleys
328             sys.stdout.flush()
329
330             # Figuras y graficos
331             plt.figure(1)
332             x = [vfh.ALPHA*x for x in range(len(robot.model.filt_polar_hist))]
333             i = [a for a in range(len(robot.model.filt_polar_hist))]
334             plt.bar(x, robot.model.polar_hist, 4.0, 0, color='r')
335             plt.xlabel(u"Ángulo [grados]")
336             plt.title("Histograma polar")
337
338             plt.figure(2)
339             plt.bar(x, robot.model.filt_polar_hist, 4.0, 0, color='b')
340             plt.xlabel(u"Ángulo [grados]")
341             plt.title("Histograma polar filtrado")
342
343             plt.figure(3)
```

```
344             plt.pcolor(robot.model._active_grid().T, alpha=0.75, edgecolors='k',vmin=0,
                    vmax=20)
345             plt.xlabel("X")
346             plt.ylabel("Y", rotation='horizontal')
347             plt.title("Ventana activa")
348             plt.show()
349
350         if robot.c_type == DR.M_BRAIT:
351             print "BraitRob end!"
352
353         if robot.c_type == DR.M_VFHP:
354             print "\nRobot position and orientation"
355             print robot.model.x_0, robot.model.y_0, robot.model.cita
356
357             print "\nRobot i, j, k"
358             print robot.model.i_0, robot.model.j_0, robot.model.k_0
359
360             print "\nRobot active grid"
361             print robot.model._active_grid()
362
363             print "\nRobot polar histogram"
364             print robot.model.polar_hist
365
366             print "Ventana activa"
367             plt.figure(1)
368             plt.pcolor(robot.model._active_grid().T, alpha=0.75, edgecolors='k',vmin=0,
                    vmax=20)
369             plt.xlabel("X")
370             plt.ylabel("Y", rotation='horizontal')
371             plt.title("Ventana activa")
372
373             x = [vfhp.ALPHA*x for x in range(vfhp.HIST_SIZE)]
374             print "histograma polar"
375             plt.figure(2)
376             plt.bar(x, robot.model.polar_hist, 4.0, 0, color='b')
377             plt.xlabel(u"Ángulo [grados]")
378             plt.title("Histograma polar")
379
380             print "histograma polar binario"
381             plt.figure(3)
382             plt.bar(x, robot.model.bin_polar_hist, 4.0, 0, color='b')
383             plt.xlabel(u"Ángulo [grados]")
384             plt.title("Histograma polar binario")
385
386             print "histograma polar mascarado"
387             plt.figure(4)
388             plt.bar(x, robot.model.masked_polar_hist, 4.0, 0, color='b')
389             plt.xlabel(u"Ángulo [grados]")
390             plt.title("Histograma polar mascarado")
391
392             plt.show()
393
394     return 0
395
396 if __name__ == "__main__":
397     np.set_printoptions(threshold=np.inf)
398     try:
399         runSim(len(sys.argv), sys.argv)
400     except Exception as inst:
```

```
401             print type(inst)
402             print inst.args
403             print inst
404             #print "press a key"
405             #wait()
406
407         print "End of exectution"
408         #wait()
409
410 logfile.close()
```

Apéndice B

Parámetros usados

Tabla B.1: Parámetros usados para el algoritmo de Braitenberg.

Nombre	Valor
radius	0.035 m
D_MIN	0.05 m
D_MAX	0.3 m
V_MIN	-0.011304 m/s
V_MAX	0.11304 m/s
W_MAX	2.2608 rad/s
alpha	0.5236 rad
smode	SMODE_FULL
MIN_READ	40

Tabla B.2: Parámetros usados para el algoritmo VFH.

Nombre	Valor
GRID_SIZE	125
RESOLUTION	0.04 m
WINDOW_SIZE	15
WINDOW_CENTER	7
ALPHA	5°
HIST_SIZE	72
THRESH	20000.0
WIDE_V	18
V_MAX	0.11304 m/s
V_MIN	0.022608 m/s
OMEGA_MAX	2.2608 rad/s
D_max2	98
B	1
A	99

Tabla B.3: Parámetros usados para el algoritmo VFH+.

Nombre	Valor
GRID_SIZE	125
C_MAX	20
RESOLUTION	0.04 m
WINDOW_SIZE	25
WINDOW_CENTER	12
ALPHA	5°
HIST_SIZE	72
D_max2	0.4608 m^2
B	10
A	5.608
R_ROB	0.02 m
D_S	0.04 m
T_LO	3000
T_HI	3500
WIDE_V	9
V_MAX	0.11304 m/s
V_MIN	0.022608 m/s
mu1	6
mu2	2
mu3	2
MAX_COST	1800

Apéndice C

Documentación

Obstacle Avoidance Documentation

Versión 1.0

Daniel Diaz Molina

17 de noviembre de 2017

Contenidos:

Esta es la documentación oficial del proyecto eléctrico *Implementación y simulación de algoritmos de evasión de obstáculos y seguimiento de trayectorias para un robot autónomo terrestre*.

Se recomienda consultar las páginas de los módulos que implementan los 3 algoritmos desarrollados, ahí se incluye la información necesaria para hacer uso del código desarrollado.

CAPÍTULO 1

Módulos

El proyecto está organizado en dos partes: el código relevante a la implementación de los algoritmos de evasión y el código relevante a la simulación con V-REP.

1.1 Py

1.1.1 Braitenberg module

Este modulo define el controlador para evasión de obstáculos por medio de vehículos de Braitenberg.

Nota: Como parte del módulo se definen las siguientes constantes relevantes al modo de funcionamiento del controlador:

- *SMODE_MIN*
- *SMODE_AVG*
- *SMODE_FULL*

Funciones de módulo

Braitenberg.**MapStimulus**(*s*, *s_min*, *s_max*, *r_min*, *r_max*)

Mapeo lineal de estímulos y respuesta. Aplica una transformación lineal del rango de entrada al rango de salida.

Parámetros

- **s** (*float*) – Valor del estímulo de entrada.
- **s_min** (*float*) – Cota inferior del valor de *s*.
- **s_max** (*float*) – Cota superior del valor de *s*.
- **r_min** (*float*) – Cota de la respuesta para un valor de entrada mínimo.

- **r_max** (*float*) – Cota de la respuesta para un valor de entrada máximo.

Devuelve Una respuesta en el rango dado por *r_min* y *r_max*.

Tipo del valor devuelto float

Nota: La función de mapeo no es mas que la ecuación de una recta que pasa por los puntos (s_min, r_min) y (s_max, r_max), por lo que es perfectamente aceptable que $r\_min > r\_max$. Esto corresponde a una respuesta con acción inversa, es decir una recta con pendiente negativa.

Clase BraitModel

class Braitenberg.**BraitModel**(*s_mode=0*, *min_read=40*, *r=0.035*, *d_min=0.01*, *d_max=0.25*, *v_min=-0.3*, *v_max=0.3*, *w_max=0.628*, *alpha=0.5235987755982988*)

Clase para el modelo de vehículo de Braitenberg

Parámetros

- **s_mode** (*{SMODE_MIN, SMODE_AVG, SMODE_FULL}, opcional*) – Modo de sensado, define la forma de obtener los estímulos a partir de las lecturas de los sensores.
- **min_read** (*int, opcional*) – Mínimo número de mediciones para el modo SMODE_FULL.
- **r** (*float*) – Radio del robot (en m).
- **d_min** (*float, opcional*) – Distancia mínima de detección de obstáculos (en m).
- **d_max** (*float, opcional*) – Distancia máxima de detección de obstáculos (en m).
- **v_min** (*float, opcional*) – Velocidad lineal mínima del robot (en m/s).
- **v_max** (*float, opcional*) – Velocidad lineal máxima del robot (en m/s).
- **w_max** (*float, opcional*) – Velocidad angular máxima del robot (se asume que es la misma en ambas direcciones, dada en rad/s).
- **alpha** (*float, opcional*) – Rango angular de visión del robot, dado en radianes.

x
float – Posición del robot sobre el eje x.

y
float – Posición del robot sobre el eje y.

gamma
float – Orientación del robot resepecto el eje z.

radius
float – Radio del robot.

sensor_l
float – Estímulo de evasión del lado izquierdo.

sensor_r
float – Estímulo de evasión del lado derecho.

target
tuple – Punto $(x,\ y)$ objetivo o None si no se está siguiendo una trayectoria.

s_mode
{SMODE_MIN, SMODE_AVG, SMODE_FULL}, – Modo de sensado, define la forma de obtener los estímulos a partir de las lecturas de los sensores

MIN_READ
int – Mínimo número de mediciones para el modo `SMODE_FULL`.

D_MIN
float, – Distancia mínima de detección de obstáculos.

D_MAX
float, – Distancia máxima de detección de obstáculos.

V_MIN
float, – Velocidad lineal mínima del robot.

V_MAX
float, – Velocidad lineal máxima del robot.

W_MAX
float, – Velocidad angular máxima del robot (se asume que es la misma en ambas direcciones).

ALPHA
float, – Rango angular de visión del robot, dado en radianes.

Examples

```
>>> import Braitenberg as Brait
>>> robot = Brait.BraitModel(s_mode=SMODE_FULL)
>>> pseudo_readings = np.float_([[0.41, np.radians(x)] for x in range(-5,90,1)])
>>> robot.UpdateSensors(pseudo_readings)
>>> robot.UpdatePos(0.0, 0.0, np.pi/2)
>>> robot.SetTarget(0.0, 0.20)
>>> v, w = robot.Mixed2b3a()
```

Métodos

BraitModel.UpdateSensors(sensor_readings)	Procesa las lecturas de los sensores para la detección de obstáculos.
BraitModel.Evade2b([d_left, d_right])	Obtiene la respuesta de evasión usando el algoritmo del vehículo 2b.
BraitModel.Evade3a([d_left, d_right])	Obtiene la respuesta de evasión usando el algoritmo del vehículo 3a.
BraitModel.UpdatePos(x, y, gamma)	Actualiza la postura actual del robot.
BraitModel.Mixed2b3a([d_left, d_right])	Obtiene la respuesta conjunta de evasión y seguimiento, usando el algoritmo 2b para evasión y el 3a para seguimiento.
BraitModel.Mixed3a2b([d_left, d_right])	Obtiene la respuesta conjunta de evasión y seguimiento, usando el algoritmo 3a para evasión y el 2b para seguimiento.

Braitenberg.BraitModel.UpdateSensors

BraitModel.**UpdateSensors**(*sensor_readings*)

Procesa las lecturas de los sensores para la detección de obstáculos.

Actualiza los valores de *sensor_r* y *sensor_l* a partir de las lecturas de los sensores. El comportamiento dependerá del atributo de clase *s_mode* y el rango de visión *ALPHA*.

Parámetros **sensor_readings** (*ndarray of float*) – Lecturas del sensor, un arreglo tipo numpy.ndarray que representa una nube de puntos (r, θ) donde r está dado en metros y θ en radianes. Solo se toman en cuenta puntos tal que $\theta \in [-\alpha, \alpha]$.

Raises

- TypeError – Si sensor_readings no es de tipo numpy.ndarray.
- ValueError – Si sensor_readings es de dimensión (2,N).

Nota: La forma en que se procesan los datos para definir el estímulo de cada lado del robot cambia dependiendo del valor de *s_mode*, de la siguiente manera:

SMODE_MIN El estímulo se define como el mínimo valor de distancia en el rango.

SMODE_AVG El estímulo se define como el promedio de los valores de distancia dentro del rango.

SMODE_FULL El estímulo se define como el promedio de los valores de distancia dentro del rango. Además se define un número mínimo de puntos por rango, dado por *MIN_READ*. Si hay menos de esta cantidad, se agregan puntos con el máximo valor de distancia *D_MAX* antes de calcular el promedio.

Braitenberg.BraitModel.Evade2b

BraitModel.**Evade2b**(*d_left=None*, *d_right=None*)

Obtiene la respuesta de evasión usando el algoritmo del vehículo 2b.

La respuesta se da en forma de velocidad lineal y angular en los rangos definidos por *V_MIN*, *V_MAX* y *W_MAX*.

Parámetros

- **d_left** (*float, opcional*) – Estímulo de evasión del lado izquierdo, por defecto toma el valor de *sensor_l*.
- **d_right** (*float, opcional*) – Estímulo de evasión del lado derecho, por defecto toma el valor de attr:*sensor_r*.

Devuelve

- **v_rob** (*float*) – Velocidad lineal de respuesta.
- **w_rob** (*float*) – Velocidad angular de respuesta (rad/s).

Braitenberg.BraitModel.Evade3a

BraitModel.**Evade3a**(*d_left=None*, *d_right=None*)

Obtiene la respuesta de evasión usando el algoritmo del vehículo 3a.

La respuesta se da en forma de velocidad lineal y angular en los rangos definidos por *V_MIN*, *V_MAX* y *W_MAX*.

Parámetros

- **d_left** (*float, opcional*) – Estímulo de evasión del lado izquierdo, por defecto toma el valor de *sensor_l*.
- **d_right** (*float, opcional*) – Estímulo de evasión del lado derecho, por defecto toma el valor de *sensor_r*.

Devuelve

- **v_rob** (*float*) – Velocidad lineal de respuesta.
- **w_rob** (*float*) – Velocidad angular de respuesta (rad/s).

Braitenberg.BraitModel.UpdatePos

BraitModel.**UpdatePos**(*x*, *y*, *gamma*)

Actualiza la postura actual del robot.

Parámetros

- **x** (*float*) – Posición del robot sobre el eje x.
- **y** (*float*) – Posición del robot sobre el eje y.
- **gamma** (*float*) – Orientación del robot resepecto el eje z en radianes.

Nota: Solo es necesario usar esta función si se desea hacer seguimiento de trayectorias.

Braitenberg.BraitModel.Mixed2b3a

BraitModel.**Mixed2b3a**(*d_left=None*, *d_right=None*)

Obtiene la respuesta conjunta de evasión y seguimiento, usando el algoritmo 2b para evasión y el 3a para seguimiento.

La respuesta se da en forma de velocidad lineal y angular en los rangos definidos por `vmin`, *V_MAX* y *W_MAX*.

Parámetros

- **d_left** (*float, opcional*) – Estímulo de evasión del lado izquierdo, por defecto toma el valor de *sensor_l*.
- **d_right** (*float, opcional*) – Estímulo de evasión del lado derecho, por defecto toma el valor de *sensor_r*.

Devuelve

- **v_rob** (*float*) – Velocidad lineal de respuesta.
- **w_rob** (*float*) – Velocidad angular de respuesta (rad/s).

Advertencia: Debe haberse definido un punto objetivo antes de llamar a esta función.

Braitenberg.BraitModel.Mixed3a2b

BraitModel.**Mixed3a2b**(*d_left=None*, *d_right=None*)

Obtiene la respuesta conjunta de evasión y seguimiento, usando el algoritmo 3a para evasión y el 2b para seguimiento.

La respuesta se da en forma de velocidad lineal y angular en los rangos definidos por *V_MIN*, *V_MAX* y *W_MAX*.

Parámetros

- **d_left** (*float, opcional*) – Estímulo de evasión del lado izquierdo, por defecto toma el valor de *sensor_l*.
- **d_right** (*float, opcional*) – Estímulo de evasión del lado derecho, por defecto toma el valor de *sensor_r*.

Devuelve

- **v_rob** (*float*) – Velocidad lineal de respuesta.
- **w_rob** (*float*) – Velocidad angular de respuesta (rad/s).

Advertencia: Debe haberse definido un punto objetivo antes de llamar a esta función.

1.1.2 DiffRobot module

DiffRobot.**M_VFH = 0**
int – Modo de operación con VFH.

DiffRobot.**M_BRAIT = 1**
int – Modo de operación con vehículo de Braitenberg.

DiffRobot.**M_VFHP = 2**
int – Modo de operación con VFH+.

Clase DiffModel

class DiffRobot.**DiffModel**(*r=0.02*, *b=0.05*, *wm_max=6.28*, *c_type=0*)
Clase para el modelo del robot diferencial.

Define un objeto que representa el robot diferencial, y permite abstraer la cinemática (comandos para los motores) a comandos de alto nivel.

Parámetros

- **r** (*float, opcional*) – Radio de las ruedas (m).
- **b** (*float, opcional*) – Distancia entre las ruedas (m).
- **wm_max** (*float, opcional*) – Máxima velocidad angular de los motores (rad/s).
- **c_type** ({*M_VFH*, *M_BRAIT*, *M_VFHP*}, opcional) – Tipo de controlador para la evasión de obstáculos.

r
float – Radio de las ruedas (m).

b
float – Distancia entre las ruedas (m).

wm_max
float – Máxima velocidad angular del motor (rad/s).

omega_max
float – Máxima velocidad angular del robot (velocidad de giro, en rad/s).

v_max
float – Máxima velocidad lineal del robot (m/s).

model
Controlador – Una instacia de un controlador para evasión, se escoge a partir de `c_type`.

v_ref
float – Valor de velocidad lineal de referencia para el movimiento del robot, definido por el controlador *model*.

w_ref
float – Valor de velocidad angular de referencia para el movimiento del robot (rad/s). Definido por el controlador *model*.

w_prev
float – Velocidad angular anterior (rad/s). Usado por el controlador VFH para calcular la velocidad angular actual.

cita
float – Dirección actual del robot (rad).

cita_prev
float – Dirección anterior del robot (rad), usado por el controlador VFH para calcular la velocidad angular actual.

cita_ref
float – Dirección de referencia para el movimiento del robot (rad). Se usa en el caso de los controladores VFH y VFH+, luego un PID determina la velocidad angular requerida *w_ref* para alcanzar esta dirección.

left_motor
float – Velocidad angular del motor izquierdo (rad/s).

right_motor
float – Velocidad angular del motor derecho (rad/s).

angle_controller
PID.PID – Un controlador PID que se usa con los algoritmos VFH y VFH+ para determinar la velocidad angular de referencia *w_ref* a partir de la dirección de referencia *cita_ref*.

Métodos

DiffModel.set_initial_pos(x, y, cita)	Define la postura inicial del robot.
DiffModel.set_target(x, y)	Indica un punto objetivo.
DiffModel.unset_target()	Dehabilita el seguimiento de trayectorias.
DiffModel.update_pos(x, y, cita, delta_t)	Actualiza la posición del robot.
DiffModel.update_readings(data)	Procesa las lecturas de un sensor.
DiffModel.update_target()	Aplica la acción de control para evasión de obstáculos.
DiffModel.setMotorSpeed()	Determina la velocidad de los motores.

DiffRobot.DiffModel.set_initial_pos

DiffModel.**set_initial_pos**(*x*, *y*, *cita*)
Define la postura inicial del robot.

Indica las condiciones iniciales $(x,\ y,\ \theta)$ del robot.

Parámetros

- **x** (*float*) – Posición absoluta del robot sobre el eje x en metros.
- **y** (*float*) – Posición absoluta del robot sobre el eje y en metros.
- **cita** (*float*) – Orientación del robot resepecto el eje z en radianes.

Notes

Modifica el estado interno del controlador *model*.

DiffRobot.DiffModel.set_target

DiffModel.**set_target**(*x*, *y*)
Indica un punto objetivo.

Habilita el seguimiento de trayectorias e indica al robot el punto al cual debe dirigirse.

Parámetros

- **x** (*float*) – Posición absoluta del objetivo sobre el eje x.
- **y** (*float*) – Posición absoluta del objetivo sobre el eje y.

Notes

Modifica el estado interno del controlador *model*.

DiffRobot.DiffModel.unset_target

DiffModel.**unset_target**()
Dehabilita el seguimiento de trayectorias.

Notes

Modifica el estado interno del controlador *model*.

DiffRobot.DiffModel.update_pos

DiffModel.**update_pos**(*x*, *y*, *cita*, *delta_t*)
Actualiza la posición del robot.

Parámetros

- **x** (*float*) – Posición absoluta del robot sobre el eje x.
- **y** (*float*) – Posición absoluta del robot sobre el eje y.
- **cita** (*float*) – Orientación del robot resepecto el eje z en radianes.
- **delta_t** (*float*) – El tiempo transcurrido desde la última actualización.

Notes

Modifica el estado interno del controlador *model*, además del atributo *cita*. En el caso de VFH y VFH+ realiza un paso de integración del PID.

DiffRobot.DiffModel.update_readings

DiffModel.**update_readings**(*data*)

Procesa las lecturas de un sensor.

Actualiza el estado del controlador *model* con las lecturas de un sensor de distancia.

Parámetros **data** (*ndarray*) – Una estructura de datos tipo `numpy.ndarray` que contiene las lecturas de un sensor de distancias. Debe ser un arreglo de dimensiones $(n \times 2)$ para n puntos o lecturas, donde cada par representa una coordenada polar (r, θ) respecto al marco de referencia del robot, dada en metros y radianes.

DiffRobot.DiffModel.update_target

DiffModel.**update_target**()

Aplica la acción de control para evasión de obstáculos.

Actualiza el estado interno de *model* para obtener la acción de control (v_{ref}, w_{ref}). Luego, la convierte en acciones individuales sobre cada motor mediante el método *setMotorSpeed()*.

Notes

Modifica los siguientes atributos de clase:

- *v_ref*
- *cita_ref*
- *w_ref*
- *left_motor*
- *right_motor*

DiffRobot.DiffModel.setMotorSpeed

DiffModel.**setMotorSpeed**()

Determina la velocidad de los motores.

Determina la velocidad individual que debe tener cada motor (en rad/s), según el modelo cinemático inverso, para que el robot alcance la velocidad lineal y angular de referencia *v_ref* y *w_ref*.

Notes

Modifica los siguientes atributos de clase:

- *left_motor*
- *right_motor*

1.1.3 PID module

class PID.**PID**(*Kp*, *Ti*, *Td*, *MAX*, *MIN*)

begin(*entrada*)

setInput(*inp*)

setRef(*ref*)

timestep(*delta_t*)

PID.**angleDiff**(*a1*, *a2*)

1.1.4 VFH module

Este módulo define el controlador para evasión mediante el algoritmo VFH, y constantes asociadas.

VFH.**GRID_SIZE = 125**
int – Tamaño de la cuadrícula de certeza.

VFH.**RESOLUTION = 0.040000000000000001**
float – Resolución de cada celda (en m).

VFH.**WINDOW_SIZE = 15**
int – Tamaño de la ventana activa.

> **Advertencia:** La ventana activa debe tener un número impar de celdas.

VFH.**WINDOW_CENTER = 7**
int – Índice de la celda central en la ventana activa.

Se define automáticamente a partir de WINDOW_SIZE.

VFH.**ALPHA = 5**
int – Tamaño de cada sector del histograma polar (en grados).

> **Advertencia:** ALPHA debe ser un divisor de 360.

VFH.**HIST_SIZE = 72**
int – Cantidad de sectores en el histograma polar. Se define automáticamente a partir de ALPHA.

VFH.**THRESH = 20000.0**
float – Valor de umbral para valles y picos.

VFH.**WIDE_V = 18**
int – Tamaño límite de un valle ancho y angosto.

VFH.**V_MAX = 0.0628**
float – Velocidad máxima del robot (m/s).

VFH.**V_MIN = 0.00628**
float – Velocidad mínima del robot (m/s).

VFH.**OMEGA_MAX = 1.256**
float – Velocidad angular máxima del robot (rad/s)

VFH.**B = 1.0**
float – Constante b de la ecuación de la magnitud del vector de obstáculos.

VFH.**A = 99.0**
float – Constante a de la ecuación de la magnitud del vector de obstáculos. Se define a partir de B de modo que se cumpla $a - b \cdot d_{max}^2 = 1$.

Clase VFHModel

class VFH.**VFHModel**
Clase que define el controlador para evasión mediante el algoritmo VFH.

obstacle_grid
ndarray of short – La cuadrícula de certeza. Cada celda tiene un valor entre 0 y 20 que indica que tan probable es que haya un obstáculo ocupando la celda.

active_window
ndarray of float – La ventana activa que se mueve con el robot. Cada celda contiene tres valores: la magnitud del vector de obstaculos $m_{i,j}$, la dirección del vector $\beta_{i,j}$, y la distancia al robot $d_{i,j}$.

polar_hist
ndarray of float – El histograma polar. Indica la densidad de obstáculos en cada sector alrededor de la vecindad del robot.

filt_polar_hist
ndarray of float – El histograma polar filtrado. Se construye a aplicando un filtro al histograma polar.

valleys
list of tuples – Contiene una lista de los valles en el histograma polar. Cada valle es un par (s_1, s_2), donde s_1 es el sector donde inicia y s_2 donde termina, en sentido antihorario.

x_0
float – Posición del robot sobre el eje x.

y_0
float – Posición del robot sobre el eje y.

cita
float – Orientación del robot resepecto el eje z.

i_0
float – Índice de la celda que ocupa el robot sobre la cuadrícula de certeza en el eje x.

j_0
float – Índice de la celda que ocupa el robot sobre la cuadrícula de certeza en el eje y.

k_0
float – Sector en el histrograma polar de la dirección del robot resepecto el eje z.

target
tuple – Punto $(x,\ y)$ objetivo o None si no se está siguiendo una trayectoria.

Examples

```
>>> import VFH
>>> robot = VFH.VFHModel()
>>> robot.update_position(1.5,1.5,270.0)
>>> pseudo_readings = np.float_([[0.3, np.radians(x)] for x in range(0,90,1)])
>>> robot.update_obstacle_density(pseudo_readings)
```

```
>>> robot.update_active_window()
>>> robot.update_polar_histogram()
>>> robot.update_filtered_polar_histogram()
>>> robot.find_valleys()
>>> try:
>>>     cita = robot.calculate_steering_dir()
>>> except:
>>>     pass
>>> v = robot.calculate_speed(3.14/180)
```

Métodos

VFHModel.update_position(x, y, cita)	Actualiza la posición del robot.
VFHModel.set_target([x, y])	Define el punto objetivo o deshabilita el seguimiento de trayectorias.
VFHModel.update_obstacle_density(sensor_readings)	Actualiza la cuadrícula de certeza a partir de las lecturas de un sensor.
VFHModel.update_active_window()	Calcula la magnitud de los vectores de obstáculo para las celdas de la ventana activa.
VFHModel.update_polar_histogram()	Actualiza el histograma polar de obstáculos.
VFHModel.update_filtered_polar_histogram()	Calcula el histograma polar filtrado a partir de el `histograma polar`.
VFHModel.find_valleys()	Analiza el histograma polar filtrado y determina los valles candidatos.
VFHModel.calculate_steering_dir()	Calcula la dirección de movimiento para la evasión.
VFHModel.calculate_speed([omega])	Calcula la velocidad del robot.

VFH.VFHModel.update_position

VFHModel.**update_position**(*x*, *y*, *cita*)

Actualiza la posición del robot.

Parámetros

- **x** (*float*) – Posición absoluta del robot sobre el eje x.
- **y** (*float*) – Posición absoluta del robot sobre el eje y.
- **cita** (*float*) – Orientación del robot resepecto el eje z en grados.

Notes

Modifica los siguientes atributos de clase:

- *x_0*
- *y_0*
- *cita*
- *i_0*
- *j_0*
- *k_0*

VFH.VFHModel.set_target

VFHModel.**set_target**(*x=None*, *y=None*)

Define el punto objetivo o deshabilita el seguimiento de trayectorias.

Define un punto objetivo (x, y) para la evasión con seguimiento de trayectorias, o deshabilita el seguimiento si el método es invocado sin parámetros.

Parámetros

- **x** (*float, opcional*) – Posición absoluta del robot sobre el eje x.
- **y** (*float, opcional*) – Posición absoluta del robot sobre el eje y.

Devuelve Retorna 1 si se estableció un punto objetivo o 0 si inhabilitó el seguimiento.

Tipo del valor devuelto int

Raises ValueError – Si el objetivo dado se sale de la cuadrícula de certeza.

Notes

Modifica los siguientes atributos de clase:

- *target*

VFH.VFHModel.update_obstacle_density

VFHModel.**update_obstacle_density**(*sensor_readings*)

Actualiza la cuadrícula de certeza a partir de las lecturas de un sensor.

Para cada lectura aumenta el valor de una única celda en 1, hasta un máximo de 20.

Parámetros **sensor_readings** (*ndarray*) – Una estructura de datos tipo numpy.ndarray que contiene las lecturas de un sensor de distancias. Debe ser un arreglo de dimensiones $(n \times 2)$ para n puntos o lecturas, donde cada par representa una coordenada polar (r, θ) respecto al marco de referencia del robot.

Raises

- TypeError – Si sensor_readings no es de tipo numpy.ndarray.
- ValueError – Si sensor_readings no tiene las dimensiones correctas.

Notes

Las coordenadas deben ser dadas en metros y radianes.

Modifica los siguientes atributos de clase:

- *obstacle_grid*

VFH.VFHModel.update_active_window

VFHModel.**update_active_window**()

Calcula la magnitud de los vectores de obstáculo para las celdas de la ventana activa.

El cálculo se hace a partir de los datos guardados en la *cuadrícula de certeza*.

Notes

Modifica los siguientes atributos de clase:

- *active_window*

VFH.VFHModel.update_polar_histogram

VFHModel.**update_polar_histogram**()
Actualiza el histograma polar de obstáculos.

El cálculo se hace a partir de los vectores de obstáculo guardados en la *ventana activa*.

Notes

Modifica los siguientes atributos de clase:

- *polar_hist*

VFH.VFHModel.update_filtered_polar_histogram

VFHModel.**update_filtered_polar_histogram**()
Calcula el histograma polar filtrado a partir de el *histograma polar*.

Notes

Modifica los siguientes atributos de clase:

- *filt_polar_hist*

VFH.VFHModel.find_valleys

VFHModel.**find_valleys**()
Analiza el histograma polar filtrado y determina los valles candidatos.

Los valles se obtienen a partir del *histograma polar filtrado* y el *valor de umbral*.

Devuelve

- `-1` si no se encuentra ningún sector con un valor mayor a *THRESH* en el hisotgrama filtrado.
- `0` de lo contrario.

Tipo del valor devuelto int

VFH.VFHModel.calculate_steering_dir

VFHModel.**calculate_steering_dir**()
Calcula la dirección de movimiento para la evasión.

El cálculo de la dirección para el robot se hace a partir de los valles candidato, la orientación actual y el objetivo. Si no hay objetivo, se toma la dirección actual de movimiento como objetivo.

Devuelve **new_dir** – La dirección del movimiento, dada en grados en el rango $[0,\ 360[$.

Tipo del valor devuelto float

Raises `Exception` – Si *`valleys`* está vacío.

VFH.VFHModel.calculate_speed

VFHModel.**calculate_speed**(*omega=0*)

Calcula la velocidad del robot.

Calcula la velocidad a partir de la densidad de obstáculos en la dirección actual del movimiento. El robot se mueve más despacio entre mayor sea la densidad.

Opcionalmente recibe la velocidad angular del robot como parámetro. En este caso, se hace una reducción adicional de la velocidad dependiendo del valor de `omega` en comparación a *`OMEGA_MAX`*.

Parámetros **omega** (*`float, opcional`*) – Velocidad angular del robot (rad/s).

Devuelve **V** – Velocidad lineal deseada del robot (m/s).

Tipo del valor devuelto float

1.1.5 VFHP module

Este módulo define el controlador para evasión mediante el algoritmo VFH+, y constantes asociadas.

VFHP.**GRID_SIZE = 125**

int – Tamaño de la cuadrícula de certeza.

VFHP.**C_MAX = 20**

int – Valor máximo de certeza.

VFHP.**RESOLUTION = 0.040000000000000001**

float – Resolución de cada celda (en m).

VFHP.**WINDOW_SIZE = 25**

int – Tamaño de la ventana activa.

> **Advertencia:** La ventana activa debe tener un número impar de celdas.

VFHP.**WINDOW_CENTER = 12**

int – Índice de la celda central en la ventana activa.

Se define automáticamente a partir de `WINDOW_SIZE`.

VFHP.**ALPHA = 5**

int – Tamaño de cada sector del histograma polar (en grados).

> **Advertencia:** `ALPHA` debe ser un divisor de 360.

VFHP.**HIST_SIZE = 72**

int – Cantidad de sectores en el histograma polar. Se define automáticamente a partir de `ALPHA`.

VFHP.**B = 10.0**

float – Constante b de la ecuación de la magnitud del vector de obstáculos.

VFHP.**A = 5.6079999999999997**
float – Constante a de la ecuación de la magnitud del vector de obstáculos. Se define a partir de B de modo que se cumpla $a - b \cdot d_{max}^2 = 1$.

VFHP.**R_ROB = 0.02**
float – Radio del robot (en m).

VFHP.**T_LO = 3000.0**
float – Valor de umbral inferior para valles en el histograma polar.

VFHP.**T_HI = 3500.0**
float – Valor de umbral superior para valles en el histograma polar.

VFHP.**WIDE_V = 9**
int – Tamaño mínimo de valle ancho.

VFHP.**V_MAX = 0.0628**
float – Velocidad máxima del robot.

VFHP.**V_MIN = 0.0**
float – Velocidad mínima del robot.

VFHP.**mu1 = 6.0**
float – Peso del costo de seguimiento.

VFHP.**mu2 = 2.0**
float – Peso del costo de cambios abruptos de dirección.

VFHP.**mu3 = 2.0**
float – Peso del costo de compromiso a una dirección.

VFHP.**MAX_COST = 1800.0**
float – Máximo valor de la función de costo.

Clase VFHPModel

class VFHP.**VFHPModel**
Clase que define el controlador para evasión mediante el algoritmo VFH+.

obstacle_grid
ndarray of short – La cuadrícula de certeza. Cada celda tiene un valor entre 0 y *C_MAX* que indica que tan probable es que haya un obstáculo ocupando la celda.

active_window
ndarray of float – La ventana activa que se mueve con el robot. Cada celda contiene cuatro valores: la magnitud del vector de obstaculos $m_{i,j}$, la dirección del vector $\beta_{i,j}$, la distancia al robot $d_{i,j}$ y el ángulo de ensanchamiento $\gamma_{i,j}$.

polar_hist
ndarray of float – El histograma polar. Indica la densidad de obstáculos en cada sector alrededor de la vecindad del robot.

bin_polar_hist
ndarray of bool – El histograma polar binario. Se construye a partir del histograma polar. Un valor True indica un sector bloqueado.

masked_polar_hist
ndarray of bool – El histograma polar mascarado. Se contruye a partir del histograma polar binario y los radios de giro instantáneos. Un valor True indica un sector bloqueado.

valleys
list of 2-tuples – Contiene una lista de los valles en el histograma polar. Cada es un par (s_1, s_2), donde s_1 es el sector donde inicia y s_2 donde termina, en sentido antihorario.

x_0
float – Posición del robot sobre el eje x.

y_0
float – Posición del robot sobre el eje y.

cita
float – Orientación del robot resepecto el eje z.

i_0
float – Índice de la celda que ocupa el robot sobre la cuadrícula de certeza en el eje x.

j_0
float – Índice de la celda que ocupa el robot sobre la cuadrícula de certeza en el eje y.

k_0
float – Sector en el histrograma polar de la dirección del robot resepecto el eje z.

target
tuple – Punto $(x,\ y)$ objetivo o `None` si no se está siguiendo una trayectoria.

prev_dir
float – Última dirección de control.

prev_cost
cost – Costo de la última dirección de control.

Examples

```
>>> import VFHP
>>> pseudo_readings = np.float_([[0.3, np.radians(x)] for x in range(0,90,1)])
>>> R_ROB = 0.01
>>> robot = VFHP.VFHModel()
>>> robot.update_position(1.5,1.5,270.0)
>>> robot.update_obstacle_density(pseudo_readings)
>>> robot.update_active_window()
>>> robot.update_polar_histogram()
>>> robot.update_bin_polar_histogram()
>>> robot.update_masked_polar_hist(R_ROB,R_ROB)
>>> valles = robot.find_valleys()
>>> cita, v = robot.calculate_steering_dir(valles)
```

Métodos

VFHPModel.update_position(x, y, cita)	Actualiza la posición del robot.
VFHPModel.set_target([x, y])	Define el punto objetivo o deshabilita el seguimiento de trayectorias.
VFHPModel.update_obstacle_density(...)	Actualiza la cuadrícula de certeza a partir de las lecturas de un sensor.
VFHPModel.update_active_window()	Calcula la magnitud de los vectores de obstáculo para las celdas de la ventana activa.

Continúa en la página siguiente

Tabla 1.4 – proviene de la página anterior

VFHPModel.update_polar_histogram()	Actualiza el histograma polar de obstáculos.
VFHPModel.update_bin_polar_histogram()	Calcula el histograma polar binario.
VFHPModel.update_masked_polar_hist(steer_l, ...)	Calcula el histograma polar mascarado.
VFHPModel.find_valleys()	Analiza el histograma polar mascarado y determina los valles candidatos.
VFHPModel.calculate_steering_dir(valley_count)	Calcula la dirección de movimiento para la evasión y la rapidez del robot.

VFHP.VFHPModel.update_position

VFHPModel.**update_position**(*x*, *y*, *cita*)

Actualiza la posición del robot.

Parámetros

- **x** (*float*) – Posición absoluta del robot sobre el eje x.
- **y** (*float*) – Posición absoluta del robot sobre el eje y.
- **cita** (*float*) – Orientación del robot resepecto el eje z en grados.

Notes

Modifica los siguientes atributos de clase:

- *x_0*
- *y_0*
- *cita*
- *i_0*
- *j_0*
- *k_0*

VFHP.VFHPModel.set_target

VFHPModel.**set_target**(*x=None*, *y=None*)

Define el punto objetivo o deshabilita el seguimiento de trayectorias.

Define un punto objetivo (x, y) para la evasión con seguimiento de trayectorias, o deshabilita el seguimiento si el método es invocado sin parámetros.

Parámetros

- **x** (*float, opcional*) – Posición absoluta del objetivo sobre el eje x.
- **y** (*float, opcional*) – Posición absoluta del objetivo sobre el eje y.

Devuelve Retorna 1 si se estableció un punto objetivo o 0 si inhabilitó el seguimiento.

Tipo del valor devuelto int

Raises ValueError – Si el objetivo dado se sale de la cuadrícula de certeza.

Notes

Modifica los siguientes atributos de clase:

- *target*

VFHP.VFHPModel.update_obstacle_density

VFHPModel.**update_obstacle_density**(*sensor_readings*)

Actualiza la cuadrícula de certeza a partir de las lecturas de un sensor.

Para cada lectura aumenta el valor de una única celda en 1, hasta un máximo de *C_MAX*.

Parámetros **sensor_readings** (*ndarray*) – Una estructura de datos tipo numpy.ndarray que contiene las lecturas de un sensor de distancias. Debe ser un arreglo de dimensiones $(n \times 2)$ para n puntos o lecturas, donde cada par representa una coordenada polar (r, θ) respecto al marco de referencia del robot.

Raises

- TypeError – Si sensor_readings no es de tipo numpy.ndarray.
- ValueError – Si sensor_readings no tiene las dimensiones correctas.

Notes

Las coordenadas deben ser dadas en metros y radianes.

Modifica los siguientes atributos de clase:

- *obstacle_grid*

VFHP.VFHPModel.update_active_window

VFHPModel.**update_active_window**()

Calcula la magnitud de los vectores de obstáculo para las celdas de la ventana activa.

El cálculo se hace a partir de los datos guardados en la *cuadrícula de certeza*.

Notes

Modifica los siguientes atributos de clase:

- *active_window*

VFHP.VFHPModel.update_polar_histogram

VFHPModel.**update_polar_histogram**()

Actualiza el histograma polar de obstáculos.

El cálculo se hace a partir de los vectores de obstáculo guardados en la *ventana activa*. Cada celda puede afectar más de un sector angular, esto se determina mediante el radio de compensación R_RS.

Notes

Modifica los siguientes atributos de clase:

- *polar_hist*

VFHP.VFHPModel.update_bin_polar_histogram

VFHPModel.**update_bin_polar_histogram**()

Calcula el histograma polar binario.

El cálculo se hace a partir del *histograma polar* y los valores de umbral *T_HI* y *T_LO*. Cada sector se indica como bloqueado (`True`) o libre (`False`).

Modifica los siguientes atributos de clase:

- *bin_polar_hist*

VFHP.VFHPModel.update_masked_polar_hist

VFHPModel.**update_masked_polar_hist**(*steer_l*, *steer_r*)

Calcula el histograma polar mascarado.

El cálculo se hace a partir del *histograma polar binario* y el radio de giro mínimo en el momento del cálculo. Las celdas ocupadas de la ventana activa bloquean sectores adicionales dependiendo de la dirección actual del robot y su capacidad de giro.

Parámetros

- **steer_l** (*float*) – Radio de giro mínimo del robot hacia la izquierda.
- **steer_r** (*float*) – Radio de giro mínimo del robot hacia la derecha.

Notes

Modifica los siguientes atributos de clase:

- *masked_polar_hist*

VFHP.VFHPModel.find_valleys

VFHPModel.**find_valleys**()

Analiza el histograma polar mascarado y determina los valles candidatos.

Los valles se obtienen a partir del *histograma polar mascarado*.

Devuelve

- `-1` si no se encuentra ningún sector bloqueado en el hisotgrama polar mascarado.
- De lo contrario, el número de valles encontrados.

Tipo del valor devuelto int

VFHP.VFHPModel.calculate_steering_dir

VFHPModel.**calculate_steering_dir**(*valley_count*)

Calcula la dirección de movimiento para la evasión y la rapidez del robot.

El cálculo de la dirección para el robot se hace a partir de los valles candidato, la orientación actual y el objetivo. Si no hay objetivo, se toma la dirección actual de movimiento como objetivo. Se usa una función de costo para seleccionar la dirección de entre todas las candidatas. La rapidez se calcula del costo de la dirección seleccionada, a mayor costo más lento se moverá el robot.

Parámetros **valley_count** (*int*) – El resultado del llamado a *find_valleys()*.

Devuelve

- **new_dir** (*float*) – La dirección del movimiento, dada en grados en el rango $[0,\ 360[$.
- **V** (*float*) – La velocidad del robot.

Raises Exception – Si valley_count es cero (todas las direcciones están bloqueadas).

Notes

Los valles angostos definen una única dirección candidata, el medio del valle. Los anchos pueden definir hasta tres: los bordes del valle y la dirección del objetivo si esta se encuentra dentro del valle.

1.2 Vrep

1.2.1 HokuyoRob script

Este script permite el enlace entre V-REP y los modelos desarrollados en el proyecto. Hace uso de la API de V-REP para Python, y crea un instacia de *DiffRobot.DiffModel* para controlar el robot diferencial, a partir de los datos suministrados por el simulador.