UNIVERSIDAD DE LAS PALMAS DE GRAN CANARIA
ESCUELA DE INGENIERÍA INFORMÁTICA

PROYECTO FINAL DE CARRERA

Modelado de sensores sónar en robótica móvil

JAVIER HERNÁNDEZ TRUJILLO

Julio de 2012

Datos del proyecto

Título: Modelado de datos obtenidos a partir de sensores sónar utilizando el método de la Transformada de Hough, desarrollado sobre framework CoolBot
Apellidos y nombre del alumno: Hernández Trujillo, Javier
Fecha : Julio de 2012

Tutor: D. Domínguez Brito, Antonio
Tutor: D. Hernández Sosa, José Daniel

Agradecimientos

Desde mi punto de vista, llevar a cabo un PFC implica organización, motivación y constancia por parte del alumno. Siempre he pensado que la motivación se consigue alcanzando pequeños objetivos mediante la constancia, y para dividir un proceso tan grande en esas pequeñas partes, se exige organización.

He de agradecer, tanto a mi familia como a aquellos que siempre se han preocupado de cómo se desarrolla este PFC, el haberme motivado y escuchado en esos momentos en los que me faltaba ese empuje tan necesario; el estar ahi siempre, en pocas palabras.

Cabe, además, una mención a ciertas personas en especial. Por un lado a José Manuel, por haber estado cerca en los comienzos del proyecto y mantenerse al tanto hasta su consecución. Por otro lado, a Antonio Carlos y a José Daniel, mi tutores y responsables de que todos los pasos se hicieran con la garantía de un trabajo firme y bien estructurado, siempre pensando a largo plazo.

Más allá de haber satisfecho los objetivos formales de este proyecto, tengo la impresión de haberme acercado más a las exigencias y condiciones de un futuro trabajo, alejándome del carácter introductorio y concreto de muchas asignaturas propias de la carrera.

Índice general

Capítulo 1

Introducción y objetivos

1.1. Introducción

Actualmente, las nuevas tecnologías acogen un abanico de utilidades inconcebible hace décadas. La portabilidad y usabilidad han permitido que los sistemas robóticos se encuentren cada vez más inmersos en el uso particular, lejos de su orígen enfocado a la industria. Este gran paso no sólo ha llevado a nuevas líneas de investigación en cuanto a las posibles utilidades, sino a pretender mejorar la autonomía de estos dispositivos con tal de asegurar la independencia y consecuente comodidad de los usuarios. Por otra parte, dicha autonomía está sujeta a cierta responsabilidad o garantía, que derivará directamente de optimizar los mecanismos mediante los cuales estos sistemas serán capaces de interpretar su entorno de trabajo y cumplir satisfactoriamente su cometido.

Por su parte, la utilidad de estos mecanismos o dispositivos de medición no sólo depende de la calidad de éstos, sino de la manera en la que se interpretan los datos que son capaces de captar. Con tal fin se utilizan las diferentes técnicas de modelado, algoritmos con un fin común, pero cuya elección dependerá de un estudio previo de ciertas particularidades, como el tipo de entorno o el ruido y precisión de los sensores empleados. Ya en concreto, los dispositivos de medición que ocupan a este proyecto son los sensores láser y los sensores ultrasónicos o sonars.

El funcionamiento del sensor láser se basa normalmente en el principio de tiempo de vuelo o *time of flight*, que consiste en enviar un haz de luz y medir el tiempo de retorno para calcular la distancia de los objetos percibidos con respecto al sensor. Por otro lado, existen sensores de este tipo, normalmente a nivel industrial, basados en lo que se denomina interferometría láser. El principio de funcionamiento se basa en la superposición de dos ondas de igual frecuencia, una directa y la otra reflejada. Se genera un haz de luz que se divide en dos partes ortogonales mediante un separador. Un haz se aplica sobre un espejo plano fijo, mientras el otro refleja sobre el objeto cuya distancia se quiere determinar; los dos haces se superponen de nuevo en el separador, de forma que al separarse el objeto se generan máximos y mínimos a cada múltiplo de la longitud de onda del haz. Finalmente, la distancia se mide contando dichas oscilaciones o franjas, obteniéndose una salida digital de elevada precisión. En la figura 1.1 se muestra un ejemplo de modelo habitual utilizado en plataformas robóticas.

Figura 1.1: Ejemplo de modelo de sensor láser utilizado en robótica móvil

Los sensores de ultrasonidos o sensores sónar son sensores de rango de tiempo de vuelo, bastante comunes en la mayoría de las plataformas de robótica móvil más utilizadas en la actualidad. Su funcionamiento se basa en la emisión de un arco de barrido que devolverá cierta distancia al rebotar con el obstáculo, pero con una información ángular muy imprecisa, a diferencia de los sensores láser, que se fundamentan en un haz rectilíneo de luz. En la figura 1.2 se muestra un robot con una distribución de sónars dispuestos horizontalmente.

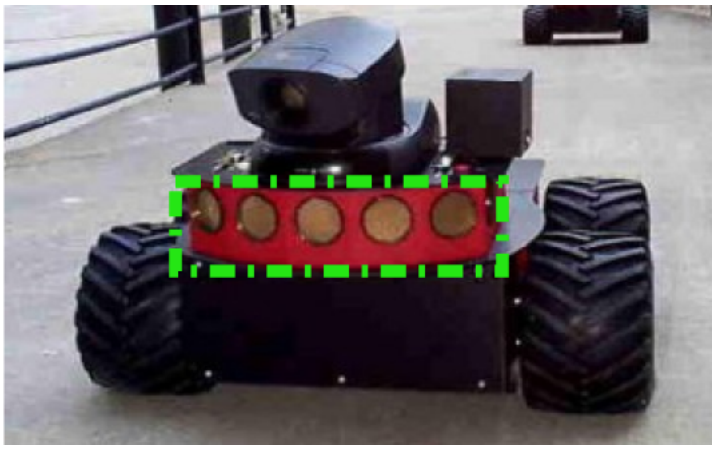


Figura 1.2: Distribución de sónars (delimitada en color verde) en robot móvil

Una vez introducidos ambos tipos de sensores, cabe analizar sus principales diferencias:

- Costo: El sónar es de muy bajo costo en comparación con el sensor láser, y de ahí su gran disponibilidad en un gran número de plataformas. Sin embargo, se debe tener presente que los sónars normalmente son funcionales al estar integrados en grupos, formando “anillos” o “líneas” de 8, 16 o 24 unidades; a favor de los sónars, este detalle no afecta a su costo en proporción al del sensor láser.
- Precisión: la imprecisión sobre la información angular de los objetos detectados con los sónars supone una gran desventaja con respecto a los sensores láser, pese a que pro-

porcionan una buena estimación de rango. Esa imprecisión, además, crece a medida que el eco se recibe desde puntos cada vez más alejados. No obstante, el sensor láser proporciona una gran precisión siempre y cuando la superficie de los objetos se lo permiten, dado que presenta problemas ante superficies transparentes o muy refractantes, lo que supone una gran ventaja para los sónars.

- Información sensorial: por cada sónar se obtiene una sola lectura de rango a frecuencias que oscilan entre 2-3 Hz; esto justifica la necesidad de distribuir estos sensores en grupos, como se mencionaba anteriormente.

Aparte de las desventajas de los sónars, normalmente es necesario solventar en su uso el problema de *cross-talk* o lecturas cruzadas. El *cross-talk* consiste en que la recepción del eco emitido por un sónar es recibido por otro sónar cercano, derivando en una detección errónea. Para subsanar este problema inherente a estos dispositivos se suelen utilizar, por ejemplo, técnicas para que todos los sónars no emitan el arco de barrido en el mismo instante.

La falta de uso de los sensores ultrasónicos en las plataformas robóticas de las que se dispone en el laboratorio, así como el bajo costo de éstos, justifica la necesidad de aplicar una técnica de modelado adecuada para utilizar la información proporcionada por este tipo de dispositivos de medición.

Por tanto y ya en mayor detalle, en este Proyecto de Final de Carrera se pretende aprovechar la información proporcionada por un tipo de sensores concreto, los sensores ultrasónicos o sónars, con tal de utilizarla en algoritmos de navegación segura y construcción de mapas. Esta información habrá de ser filtrada e interpretada convenientemente mediante cierta técnica de modelado de sensores, la cual deberá ajustarse a las circunstancias particulares del tipo de entorno, sensores y plataforma robótica a emplear.

1.2. Objetivos

El fin último de cualquier Proyecto Final de Carrera es que el alumno se enfrente a un problema a resolver de mayor calibre y más próximo a lo que se encontrará potencialmente en el ámbito laboral, en comparación a las prácticas realizadas a lo largo de la formación académica en las distintas materias. Asimismo, habrá de hacer uso de los conocimientos adquiridos en dichas materias para acometer satisfactoriamente los objetivos previstos en el PFC.

Este proyecto en cuestión es exploratorio, de investigación, dado que su objetivo final es dar uso a los sensores ultrasónicos disponibles basándose en una técnica de modelado concreta que se pretende implementar, pero de la cual se desconocen sus resultados en relación a las condiciones particulares del propio proyecto.

Por tanto, los objetivos vinculados a este PFC se pueden dividir en específicos y particulares. Los objetivos específicos son aquellos relacionados con el ámbito técnico del PFC en cuestión, esto es, qué se pretende realizar y cómo:

- Modelado matemático de sensores sónar.

- Estudiar la necesidad de técnicas de modelado para sensores ultrasónicos.
- Estudiar idoneidad de técnica de Transformada de Hough como técnica de modelado para esta clase de sensores. Estudiar la técnica en cuestión.
- Aplicar la metodología de desarrollo seleccionada para un prototipo inicial en Matlab, que permita tener una primera aproximación práctica de la técnica de la

▪ Prototipo operativo integrado en CoolBOT

- Transformada de Hough y ofrezca un marco de desarrollo previo y directo para solventar posibles dificultades antes de proceder con el siguiente objetivo.
- Estudio de framework CoolBOT.
- Aplicar la metodología de desarrollo seleccionada para un prototipo final en framework CoolBOT, que implemente la Transformada de Hough con tal de poder demostrar la factibilidad de su uso en el contexto de detección de entornos.

Por su parte, los objetivos académicos se refieren tanto a competencias a adquirir por el alumno como a conocimientos a aplicar a lo largo del desarrollo. Sin embargo, estos objetivos no parten realmente de requisitos exigidos, pero no es menos cierto que la dificultad del PFC irá en proporción a los conocimientos previos de los que disponga y aplique el alumno.

Capítulo 2

Metodología y Plan de trabajo

En este capítulo se abordará tanto la metodología utilizada como el plan de trabajo establecido en el desarrollo del software. Por otra parte, también se centrará en los recursos, tanto hardware como software, requeridos durante el transcurso de este proyecto.

2.1. Metodología

Este concepto se refiere al marco de trabajo usado para estructurar, planificar y controlar el proceso de desarrollo. La metodología a utilizar es de suma importancia, dado que una buena organización del método de desarrollo permite evitar riesgos potenciales y conlleva a que el proyecto resulte más predecible y de mayor calidad.

Existen diversas metodologías, y cada una sigue su propio enfoque de cara a las técnicas utilizadas. En el caso de este proyecto, la metodología más recomendable a seguir ha sido la del denominado Proceso Unificado o *UP* [Jacobson, Booch and Rumbaugh, 2000]. Éste consiste en un marco de desarrollo que se caracteriza por estar dirigido por casos de uso, estar centrado en la arquitectura, enfocado en los riesgos y ser iterativo e incremental; posteriormente se explicarán en detalle tales características. Por otro lado, se encuentra el *RUP* o Proceso Racional Unificado, que, junto con el Lenguaje Unificado de Modelado o *UML*[1], constituye la metodología estándar más utilizada para el análisis, implementación y documentación de sistemas orientados a objetos. No obstante, no hay que confundir este acrónimo con el software homónimo creado por la empresa Rational, hoy propiedad de IBM, ya que en este caso las siglas *RUP* aluden a *Proceso Unificado de Rational*.

Retomando la metodología que compete a este proyecto, se analizarán en detalle las diferentes carácterísticas ya citadas:

- Dirigido por casos de uso: éstos se utilizan para capturar los requisitos funcionales y para definir los contenidos de las iteraciones. El fundamento es que cada iteración tome un conjunto de casos de uso y desarrolle todo el proceso a través de las distintas etapas

[1]Se trata del lenguaje de modelado de sistemas más estandarizado y que, en términos generales, permite describir gráficamente los procesos involucrados en dichos sistemas.

de desarrollo.

- Centrado en arquitectura: a grandes rasgos, la arquitectura consiste en el conjunto de todas las vistas del sistema, que a su vez representan diferentes perspectivas del mismo. La arquitectura, por tanto, indica la estructura, funcionamiento e interacción entre las partes del software.

  Philippe Kruchten[2] la definiría del siguiente modo:

  > "*La arquitectura de software, tiene que ver con el diseño y la implementación de estructuras de software de alto nivel. Es el resultado de ensamblar un cierto número de elementos arquitectónicos de forma adecuada para satisfacer la mayor funcionalidad y requerimientos de desempeño de un sistema, así como requerimientos no funcionales, como la confiabilidad, escalabilidad, portabilidad, y disponibilidad.*"

- Enfocado en los riesgos: esta metodología propone identificar los riesgos críticos a priori, en una etapa temprana del ciclo de vida. De esta forma, los resultados de cada iteración deben ser contemplados en un orden que garantice que los riesgos principales sean considerados primero.

- Iterativo e incremental: se trata de un proceso de desarrollo de software, creado en respuesta a las debilidades del modelo tradicional de cascada. En la figura 2.1 se presenta gráficamente su forma de proceder.

  En un proceso de desarrollo iterativo, el trabajo es organizado en iteraciones -pequeños "proyectos" de duración fija- con ciertas características, cuya aplicación es posible por la orientación a objetos del sistema:

  - Cada una incluye sus propias actividades: análisis de los requisitos, diseño, implementación y prueba.
  - El resultado de cada iteración debería ser un sistema ejecutable pero incompleto, no listo para la producción. Sin embargo, dicho resultado tampoco será descartable, dado que se reutilizará, mejorará o ampliará en posteriores iteraciones.
  - El sistema final requerirá normalmente de cierto número de iteraciones antes de estar listo para la producción y ser considerado un sistema probado, integrado y ejecutable.
  - La duración de cada iteración exige un buen control, dado que poco tiempo implicaría producir sistemas carentes de suficiente información para obtener una realimentación de valor y, en el lado opuesto, demasiado tiempo conllevaría aumentar los riesgos del proyecto, al no recibir realimentación a tiempo. En el caso de prever que no se alcanzarán los objetivos establecidos para la iteración en curso por cuestión de tiempo, convendría tratar sus requerimientos en futuras iteraciones, con tal de no posponer su terminación.

[2]Ingeniero de software canadiense, director de *RUP* en Rational Software desde 1996 a 2003, cuando IBM adquirió la compañia.

- Normalmente una iteración aborda nuevos requerimientos y desencadena un incremento en el sistema, pero puntualmente puede encargarse de revisar el software existente con tal de mejorarlo.

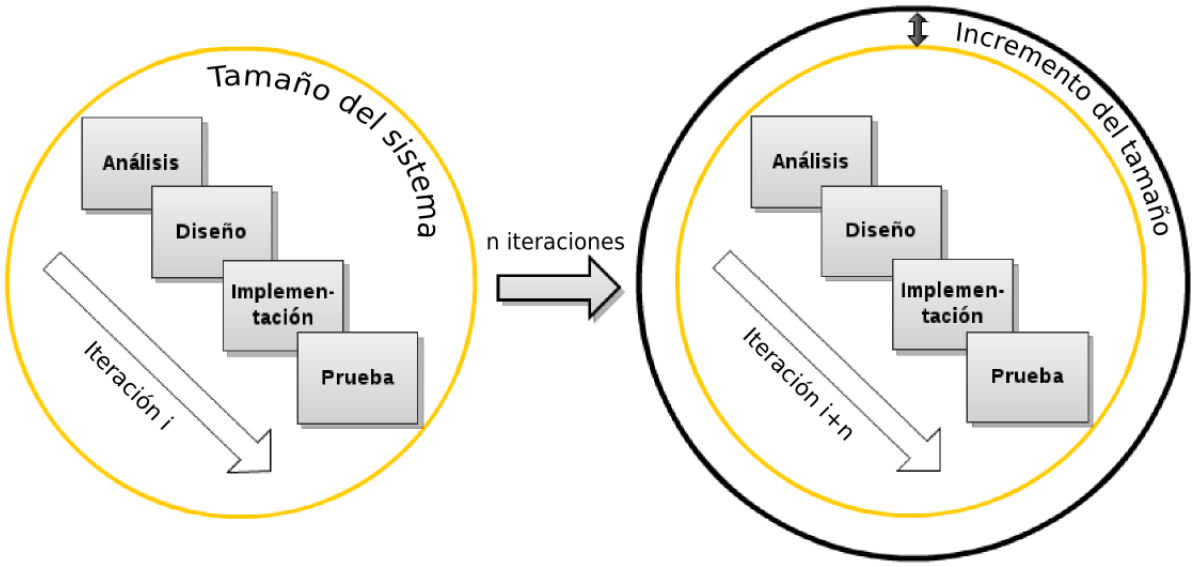


Figura 2.1: Proceso de desarrollo iterativo e incremental

Este mecanismo garantiza ciertos beneficios de cara al proceso de desarrollo, a saber:

- Mitigación de riesgos altos tan pronto como sea posible.
- Progreso visible en las primeras etapas.
- Una temprana retroalimentación mediante el control de la duración de las iteraciones, compromiso de los usuarios del sistema y adaptación.
- Al ir desarrollando parte de las funcionalidades, es más fácil determinar si los requerimientos planeados para los niveles subsiguientes son correctos.
- Si se produce un error importante, sólo la última iteración necesita ser descartada y puede utilizarse un incremento previo.
- Gestión de la complejidad; el equipo de desarrollo no se ve abrumado por "parálisis" en el análisis o pasos muy largos y complejos.
- El conocimiento adquirido en una iteración puede ser utilizado para mejorar el propio proceso de desarrollo.

Una vez introducida la metodología de desarrollo a seguir, procede darle un enfoque más concreto, orientado a los objetivos planteados en este proyecto.

La técnica de *Proceso Unificado* se aplicará, **de forma independiente**, tanto al desarrollo del prototipo en Matlab como del prototipo final en CoolBOT:

1. Prototipo Matlab: en primera instancia, se plantearán los requisitos previos y el objetivo final de estos prototipos. Seguidamente, comenzarán las iteraciones del proceso

iterativo e incremental con tal de alcanzar una versión final que satisfaga los requisitos establecidos, asumiendo una necesidad de mejora del producto final, o una revisión de alguna iteración anterior -puntualmente-, en cada una de éstas.

2. Prototipo final en CoolBot: del mismo modo que en el caso del prototipo en Matlab, en esta implementación se definirán a priori los requisitos a satisfacer por el sistema final, ejecutando posteriormente iteraciones necesarias para alcanzar el objetivo.

2.2. Recursos

En este apartado se especificarán los diferentes recursos utilizados durante el desarrollo de este proyecto, tanto a nivel *hardware* como *software*.

2.2.1. Recursos hardware

- Portátil *LG* modelo *R500*, con las siguientes especificaciones:
  - Procesador Intel Core 2 Duo T7300, a 2 GHz. Arquitectura de 64 bits.
  - Tarjeta gráfica Nvidia Geforce 8600m GS con 256 Mb dedicados.
  - 2 Gb DDR2 de RAM.
  - 1 Gb Intel Turbo Memory.
  - 500 Gb de disco duro SATA a 5400 rpm.
- Robot Pioneer 3-DX con 16 sensores ultrasónicos y 1 sensor láser.
- Red inalámbrica.
- Servidor Git para control de versiones/copias de seguridad.

2.2.2. Recursos software

En este caso se detallarán las utilidades software más destacadas entre las asociadas al desarrollo de este proyecto.

- Sistema Operativo y distribución
  - **Ubuntu 11.10 -*Oneiric Ocelot*-**.
- IDE o Entornos de Desarrollo Integrado
  - **Matlab r2008a**: entorno de programación para el desarrollo de algoritmos, el análisis de datos, la visualización y el cálculo numérico. Ofrece un *IDE* con un lenguaje de programación propio, el lenguaje *M*.

  - **Kdevelop 4.2**: entorno de programación utilizado en este caso para implementar y compilar código escrito en el lenguaje *C++*.
  - **Kile 2.1.0**: entorno que permite editar y compilar ficheros generados en LaTeX, usando la versión 4.7.4 de la plataforma de desarrollo de *KDE*. Por su parte, LaTeX [Lamport, 1986] [Roland, 1983] es un sistema de composición de textos basado en comandos de TeX, un lenguaje tipográfico ampliamente utilizado.

- Librerías
  - **Gtk+ 2.0** [GTK+ 2.0 Web Page]: acrónimo de *Gimp ToolKit*, que consiste en un conjunto de utilidades o *widgets* multiplataforma para la creación de Interfaces Gráficas de Usuario (*GUI*).

- Compiladores
  - **g++**: compilador del lenguaje de programación *C++* [Stroustrup,1997] [Vandevoorde,2003].
  - **CMake**: conjunto de herramientas diseñadas para controlar el proceso de compilación del software y compilar archivos de configuración independientes.
  - **pkg-config**: software que provee una interfaz unificada para llamar bibliotecas instaladas cuando se está compilando un programa a partir del código fuente.

- Depuradores
  - **gdb**: depurador *GNU* de *C/C++*.

- Gestores de paquetes
  - **Apt**: acrónimo en inglés de *Herramienta Avanzada de Empaquetado*, que ofrece una serie de utilidades que facilitan el manejo de paquetes a través de comandos en el terminal. Existe una interfaz gráfica de este gestor mediante *GTK+* denominada **Synaptic**, también utilizada en ciertas ocasiones.

- Sistemas de Control de Versiones
  - **Git**: se trata de un Sistema de Control de Versiones cuyo uso se encuentra bastante extendido y que, entre otras prestaciones, permite mantener un registro de cambios o *versiones* del software en desarrollo.

- Herramientas de gráficos
  - **Inkscape 0.48.2 r9819**: editor de gráficos vectoriales de código abierto, que busca ser compatible con *SVG* o *Gráficos Vectoriales Escalables*.
  - **Gimp 2.6**: acrónimo de *GNU Image Manipulation Program* o *Programa de Manipulación de Imágenes de GNU*, que ofrece una gran variedad de herramientas con tal finalidad.
  - **Gliffy**: editor de diagramas online, que permite crear y compartir diagramas de flujo y diseños de interfaces de usuario entre otros.

2.2.3. Frameworks de programación

Aunque en realidad forman parte de los recursos software, los frameworks utilizados se explicarán seguidamente con un mayor nivel de detalle, dada su relevancia en el transcurso de este proyecto.

Player-Stage

Player [Player-Stage] es un software que ejerce de capa de abstracción de hardware para dispositivos robóticos (sensores y actuadores). Esto permite controlar los distintos dispositivos que pudiera incluir un robot y obtener información a partir de sus sensores sin que los desarrolladores tengan la necesidad de atender a detalles físicos o *hardware*. Siguiendo un modelo cliente-servidor, las funcionalidades del robot actuarían como clientes del servidor Player, haciendo uso de ciertas funciones aportadas por éste último para abstraerse de problemas de comunicación. En este PFC se ha hecho uso de la versión 2.1.2 de Player.

Stage, por su parte, es un simulador en dos dimensiones de entornos o mapas -contiene algunos por defecto y permite nuevas inclusiones generadas por el usuario- para robots, capaz de simular cierto número de unidades coexistentes al mismo tiempo. Es posible usarlo en conjunción con Player para probar las funcionalidades implementadas, y precisamente el motivo por el que se ha usado durante gran parte del desarrollo de este proyecto ha sido por su gran similitud con el mundo real de cara al comportamiento de los algoritmos y de los sensores del robot. En la figura 2.2 se ilustra un ejemplo de mapa generado mediante Stage, habiendo utilizado en este PFC la versión 2.1.1.

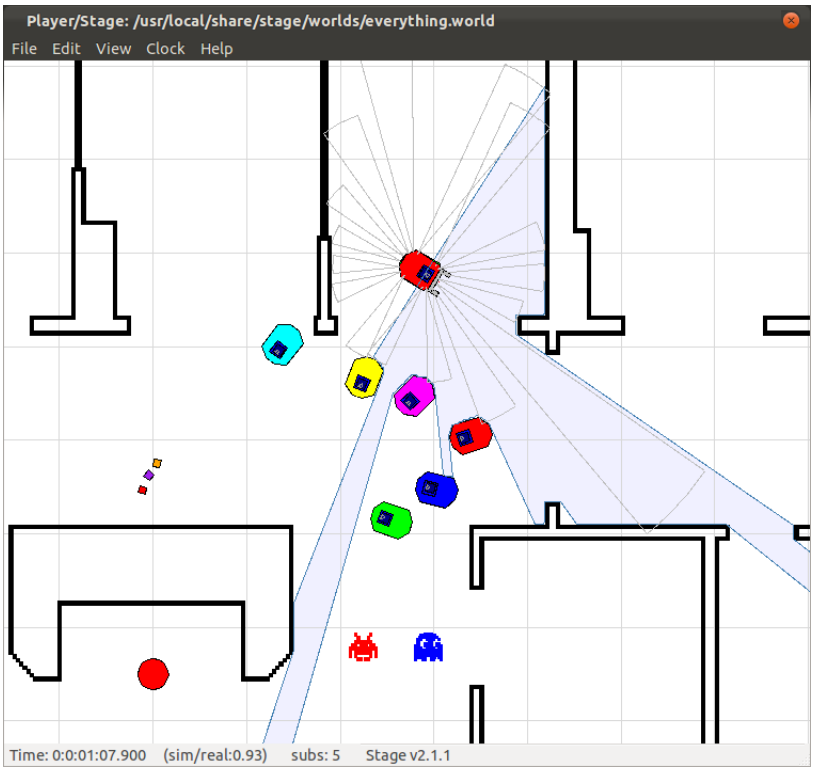


Figura 2.2: Captura de robot con sensores ultrasónicos y láser en Stage

CoolBOT

CoolBOT [Domínguez-Brito,2011b] [Domínguez-Brito,2012b] [Domínguez-Brito,2012c] es un framework o marco de programación distribuido en C++ que sigue el paradigma *CBSE* (Component Based Software Engineering o Ingeniería de Software Basado en Componentes) [George,2001] como principio de diseño de sistemas software. La clave de este paradigma es el concepto de ***componente software***, que es una unidad de integración, composición y reutilización de software. Los sistemas complejos podrían estar compuestos de varios componentes listos para usarse; interconectando componentes disponibles a partir de un repositorio de componentes previamente desarrollados, se podría programar un sistema completo. Así, debería bastar con una interfaz gráfica o similar para configurar un sistema.

La figura 2.3 aporta una visión global de un sistema típico desarrollado usando CoolBOT. Como se puede observar, hay cinco componentes y dos vistas, formando entre éstos cuatro integraciones que implican a tres máquinas diferentes compartiendo una red. Además, en una de las máquinas hay una aplicación externa a CoolBOT que utiliza una sonda para interactuar con uno de los componentes del sistema.

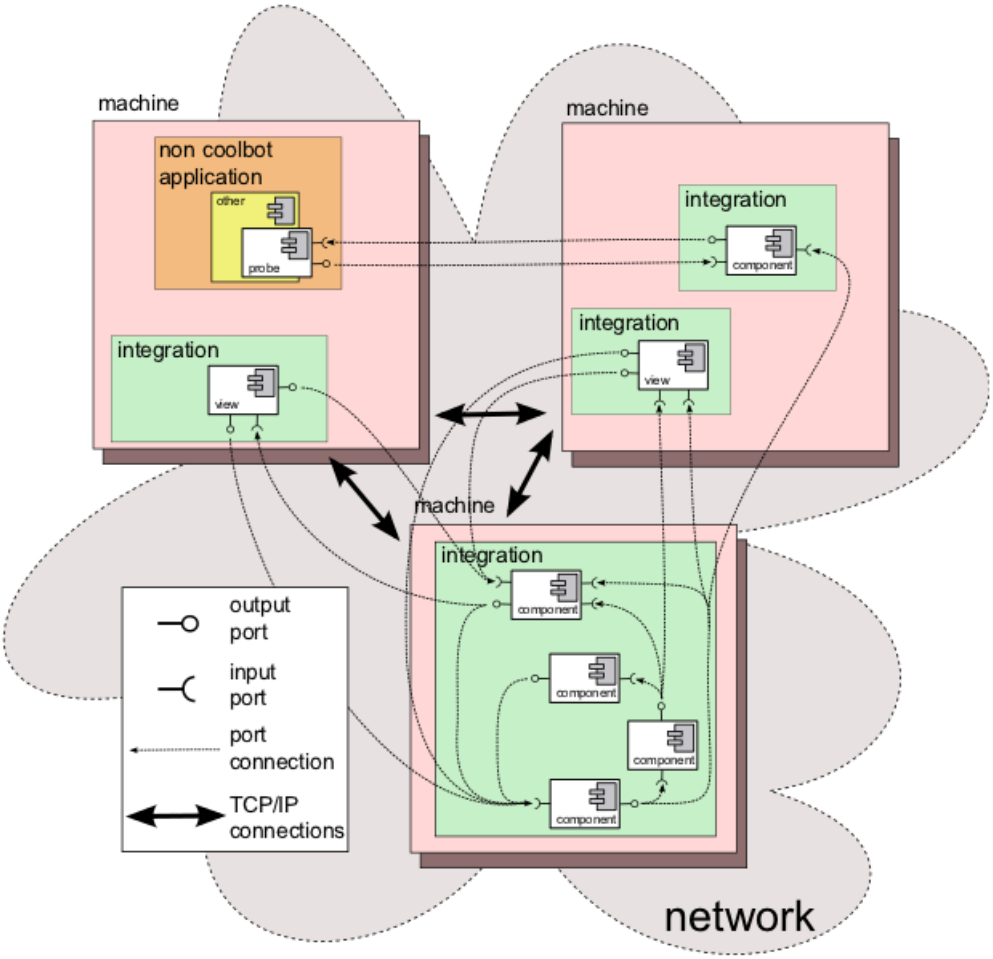


Figura 2.3: Ejemplo de sistema desarrollado en CoolBOT (extraído de [Domínguez-Brito, 2012a])

De esta forma, en CoolBOT se pueden encontrar 3 tipos de componentes software: componentes, vistas y sondas o *probes*. Todos estos tipos son componentes software en un sentido amplio, en cuanto a que se pueden combinar de cualquier modo sin cambiar su implementación para construir un sistema. La principal diferencia entre éstos es que las vistas y las sondas se pueden considerar software "ligero" en comparación con los componentes. Las vistas son componentes software que implementan interfaces gráficas de monitorización y control para sistemas CoolBOT, totalmente desacoplados de éstos. Por otra parte, las sondas principalmente permiten implementar interfaces desacopladas para interoperación de sistemas CoolBOT con aplicaciones software no CoolBOT.

Desde un punto de vista de programación, se pueden identificar diferentes perfiles para los usuarios de CoolBOT, relacionados con las diferentes capas de abstracción de la figura 2.4.

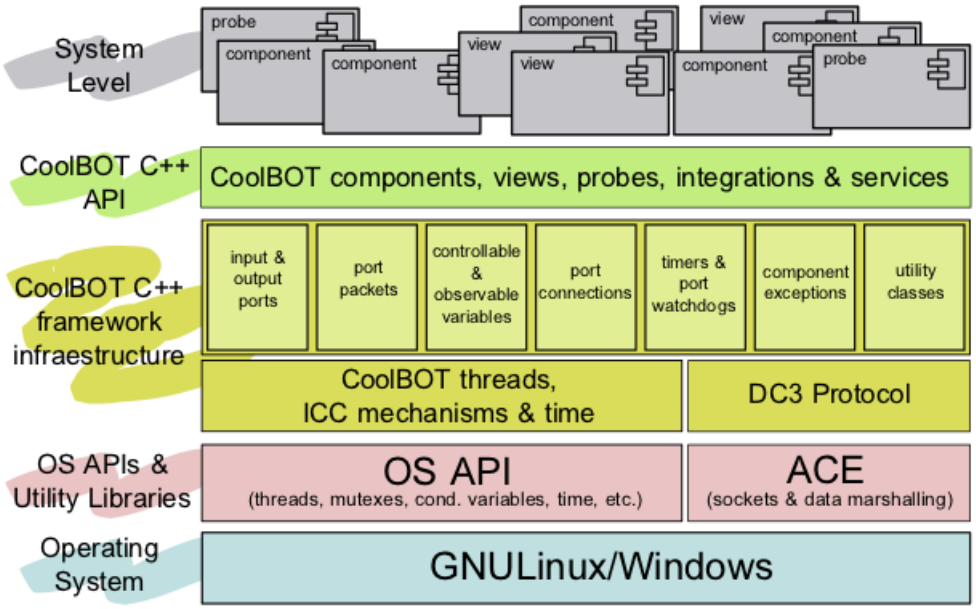


Figura 2.4: Diagrama de capas de abstracción de software en CoolBOT (extraído de [Domínguez-Brito, 2012a]

- *Desarrolladores del framework*: CoolBOT es principalmente un marco o framework de programación en C++ a la hora del diseño, desarrollo y ejecución, que provee facilidades para el soporte de aplicaciones basadas en abstracciones CoolBOT. Esta infraestructura es la denominada CoolBOT framework. Para diseño y desarrollo, CoolBOT también aporta infraestructura a modo de aplicaciones y herramientas de desarrollo. Los desarrolladores del framework son los desarrolladores de la infraestructura de programación de CoolBOT a cualquier nivel.

- *Desarrolladores de componentes*: programadores/desarrolladores de componentes software CoolBOT (componentes propiamente dichos, vistas y sondas). También se pueden incluir aqui desarrolladores de aplicaciones software no CoolBOT que operan con sistemas CoolBOT usando sondas.

- *Integradores del sistema*: son aquellos que generan integraciones CoolBOT a partir de componentes software ya desarrollados.
- *Usuarios genéricos*: se trata de usuarios de CoolBOT que recurren a sistemas ya desarrollados e integrados.

Entre los objetivos inherentes a este proyecto se halla la creación de un ***bundle*** con ciertas funcionalidades. Un *bundle* consiste en un paquete de software CoolBOT que encapsula todos los componentes software e integraciones, así como los paquetes de puertos necesarios, con los que se vaya a generar cierto sistema. Un *bundle* provee de una infraestructura para configurar y compilar todos los elementos que contenga, así como de una infraestructura CMake para su instalación. El concepto de *bundle*, por ende, permite una mayor organización y portabilidad de los sistemas generados en CoolBOT.

2.3. Plan de trabajo

Este último apartado del capítulo se centrará en estructurar las diferentes etapas de desarrollo del proyecto, añadiendo información relevante cuando se considere preciso. Además, se citarán las asignaturas del proyecto docente de la carrera cuyos conocimientos han sido necesarios para la consecución de los objetivos previstos.

- *Etapa 1*: Estudio del problema a resolver, planteamiento de la solución y aproximación de solución seleccionada.
- *Etapa 2*: Análisis, diseño y desarrollo de prototipo Matlab inicial.
- *Etapa 3*: Análisis, diseño y desarrollo de prototipo final en CoolBOT.
- *Etapa 4*: Confección del Documento de Proyecto Final de Carrera y preparación de la Defensa del Proyecto.

El estudio de la técnica de modelado *HT* o *Hough Transform* (Transformada de Hough) es fundamental, y por ello se hace imprescindible entender perfectamente su forma de proceder antes de plantear cómo integrarla en los prototipos y el *bundle* propiamente dicho.

Para el caso de los prototipos en Matlab, fueron válidos los conocimientos adquiridos con asignaturas como las siguientes:

- *FGC* (*Fundamentos Gráficos por Computador*) para la transformación entre los distintos sistemas de referencia y manipulación y cálculo de vectores.
- *AAM* (*Ampliación de Análisis Matemático*) para detalles como interpolación mediante splines cúbicos.
- *CPC* (*Control de Procesos por Computador*), *ITS* (*Introducción a la Teoría de Sistemas*), *TS* (*Teoría de Sistemas*) e *Instrumentación* para la programación en el entorno de Matlab en general.

Para el caso del desarrollo en CoolBOT se siguió un perfil de *desarrollador de componentes software*, citado en el apartado anterior. Asimismo, las asignaturas base serían las referidas a continuación:

- *BIO* (*Biocibernética Computacional* y *SRM* (*Sistemas Robóticos Móviles*) para la familiarización con sistemas robóticos, tanto desde el punto de vista *hardware* -diferentes dispositivos de movimiento y medición- como *software* -interactuación con éstos mediante comandos y creación de funcionalidades-.
- Asignaturas de programación en lenguaje C++, como *TP* (*Tecnología de la Programación* y *ED* (*Estructuras de Datos*), para ficheros distribuidos, estructuras de datos y orientación a objetos.
- *SO* (*Sistemas Operativos*) y *DSO* (*Diseño de Sistemas Operativos*), para manejo de tecnologías de concurrencia y multihilos, así como para controlar el terminal de Linux.

A nivel más general, y ya en cuanto a la organización del desarrollo, fueron necesarios conocimientos extraidos de las asignaturas *IS1* e *IS2* (*Ingeniería del Software I y II*).

Finalmente, para la confección de este documento mediante LaTeX, se atendió entre otras fuentes a la introducción ofrecida en la asignatura *PI* (*Proyectos Informáticos*).

Capítulo 3

Análisis del problema

3.1. Necesidad de técnicas de modelado: la Transformada de Hough

En el capítulo 1.1 se introducen las particularidades de los sensores ultrasónicos, y se sugiere la necesidad de utilizar una técnica de modelado apropiada para interpretar los datos obtenidos a partir de este tipo de sensores.

En este proyecto exploratorio se ha decidido, como prerequisito, utilizar la aproximación para la formalización y modelado de sensores de ultrasonidos propuesta en [Tardós,2002].

En [Tardós,2002, págs. 7–9] se demuestra que con cierto movimiento del robot y usando la odometría (valores como distancia recorrida, posición y velocidad del mismo), es posible agrupar sensores ultrasónicos y determinar los tipos de objetos detectados en el entorno [Burguera,2009]. En la figura 3.1 se muestra un ejemplo de datos obtenidos con un anillo de 24 sónars a partir de cierto movimiento simple del robot (primero una rotación y luego un movimiento rectilíneo). En ésta, cada punto representa la posición nominal de un retorno sónar, que coincide con el eje de abcisas del sensor. Las líneas de puntos, por su parte, representan el mapa real.

En la figura se puede apreciar como las paredes a ambos lados de la trayectoria del robot se muestran claramente como líneas de los retornos de los sónars. Sin embargo, delante del robot aparecen algunas líneas falsas; normalmente se deben a personas u objetos móviles que se mueven en el entorno del robot, pero también pueden deberse a problemas como el *cross-talk* (explicado en capítulo 1.1) o a reflexiones especulares de los ecos emitidos. Por tanto, las técnicas de segmentación simples utilizando sólo la posición nominal de los retornos sónar pueden encontrar satisfactoriamente grupos de retornos provenientes del mismo objeto (real o ficticio), pero seguramente fallarán a la hora de distinguir entre objetos reales y falsos o ficticios y en cuanto a la identificación del tipo de objeto (línea o esquina).

Sin embargo, si se utiliza una técnica de modelado más detallada, la ambigüedad puede ser reducida significativamente. En la figura 3.2 se muestra la detección del mismo entorno que en la figura 3.1, pero esta vez considerando los arcos de barrido con una incertidumbre en el ángulo (en la figura 3.1 coincidía con el eje de abcisas de cada sónar).

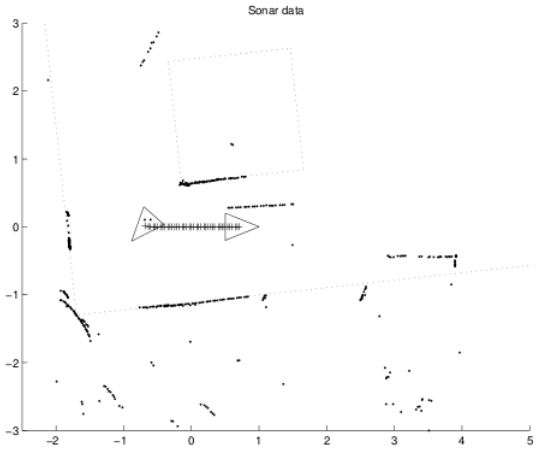


Figura 3.1: Mapa generado tras 40 posiciones del robot, atendiendo a posición nominal de retorno en cada sónar (extraído de [Tardós,2002])

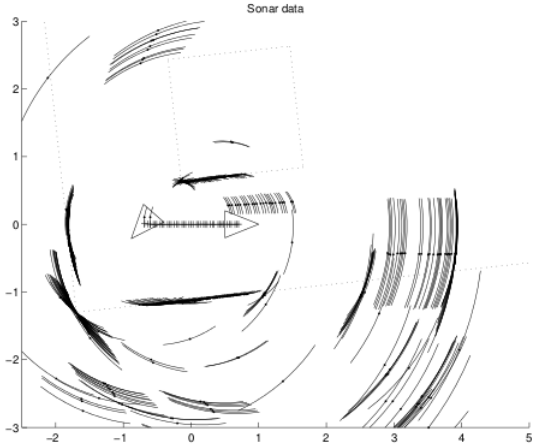


Figura 3.2: Mapa generado tras 40 posiciones del robot, atendiendo esta vez a los arcos de barrido sin especificación de ángulo (extraído de [Tardós,2002])

En esta representación las paredes -con características de líneas- aparecen como conjuntos de arcos sónar tangentes a la paredes potencialmente reales, mientras que las esquinas o filos -con características de puntos- aparecen como conjuntos de arcos que se cruzan entre sí. Por el contrario, los objetos potencialmente falsos producen conjuntos de arcos sónar incoherentes, de manera que pueden ser fácilmente detectados y descartados. La conclusión de la mejora en este ejemplo es clara: usar un modelo adecuado de sensor es crucial para interpretar adecuadamente los datos obtenidos con los sónars.

Con este último modelo de los sónars, el asociar retornos sónar con características de líneas y puntos puede traducirse en encontrar grupos de arcos de barrido tangentes a la misma línea o cruzándose en el mismo punto, respectivamente. Dada la gran cantidad de datos falsos -provenientes de objetos móviles, reflexiones especulares o a fallos inherentes a los sónars como el *cross-talk*-, parece muy adecuado utilizar técnicas robustas como RANSAC [Fischler and Bolles,1981] [Hartley and Zisserman,2000] o la **Transformada de Hough** [Ballard and Brown,1982] [Illinworth and Kittler,1988] [Censi,2005].

La Transformada de Hough es un sistema de votación donde cada información de sensor acumula evidencia sobre la presencia de ciertas características compatibles con la medida en cada instante. La votación en un espacio paramétrico discretizado, conocido como el espacio de Hough, que representa todas las posibles localizaciones de las características. Las celdas más votadas del espacio de Hough deberían corresponder a las características presentes en el entorno. Evaluando los votos es sencillo obtener los grupos de datos de los sensores provenientes de cada característica detectada.

Esta técnica parece conveniente para la resolución del problema de asociación de los datos de los sónars, dadas las siguientes razones:

- La localización de características de líneas y puntos puede ser fácilmente descrita con dos parámetros, teniendo un espacio de Hough en dos dimensiones donde el proceso de votación y la búsqueda de los valores máximos puede ser realizado de manera eficiente.
- Una vez que cada retorno de sónar emite un número constante de votos, el proceso de Hough es lineal con respecto al número de retornos evaluados,
- Tratándose de un sistema de votación, es robusto de forma inherente contra la presencia de retornos falsos de los sensores ultrasónicos.

3.2. Estudio de las etapas de desarrollo para los sónars

En este apartado se realizará un estudio detallado de las diferentes etapas implementadas para el caso de los sensores ultrasónicos durante el desarrollo del proyecto:

- Etapa 1: Discretización en el arco de barrido y cálculo de posición con respecto a sensor
- Etapa 2: Cálculo de posición con respecto a origen relativo
- Etapa 3: Discretización e inserción en matrices de la Transformada de Hough
- Etapa 4: Valoración de persistencia
- Etapa 5: Extracción de líneas y puntos en matrices de la Transformada de Hough
- Etapa 6: Fusión de las líneas y puntos extraídos
- Etapa 7: Hipótesis de líneas

En concreto, este nivel de detalle se centrará tanto en la descripción de los problemas a solventar como de los métodos matemáticos y de implementación de los que se ha hecho uso.

Las primeras tres etapas se ejecutarán secuencialmente para cada uno de los sónars cada vez que éstos reciban el retorno de los ecos ultrasónicos.

La etapa 4 se ejecutará cada cierto período de tiempo T, mientras que las tres ultimas etapas pasarán a realizarse de forma secuencial cada cierto período de tiempo T'.

3.2.1. Etapa 1: Discretización en el arco de barrido y cálculo de posición con respecto a sensor

La base de todas las etapas de desarrollo es el conjunto de puntos discretizados del arco de barrido de cada uno de los sensores ultrasónicos, tanto para hallar líneas como para hallar puntos.

Así pues, esta etapa se fundamenta en la distancia devuelta por el sensor correspondiente y en ciertos parámetros preestablecidos. En el algoritmo 1 se muestra la forma de proceder.

Algoritmo 1 Discretización de puntos en el arco de barrido

Entrada: Distancia devuelta ρ

Ángulo de apertura del arco de barrido β

Número de puntos discretizados a evaluar n

Salida: Lista de puntos en coordenadas cartesianas P$\{p_1,p_2,...p_n\}$

1: $\Delta\theta \leftarrow \beta/\mathrm{n}$

2: $\theta \leftarrow -\beta/2$

3: **mientras** $\theta \leq \beta/2$ **hacer**

4: $p_i \leftarrow \rho \cdot \begin{bmatrix} cos_\theta \\ sin_\theta \end{bmatrix}$

5: Añadir p_i a P

6: $\theta \leftarrow \theta + \Delta\theta$

7: **fin mientras**

8: **devolver** P

El cálculo expuesto en el algoritmo se justifica atendiendo al gráfico 3.3.

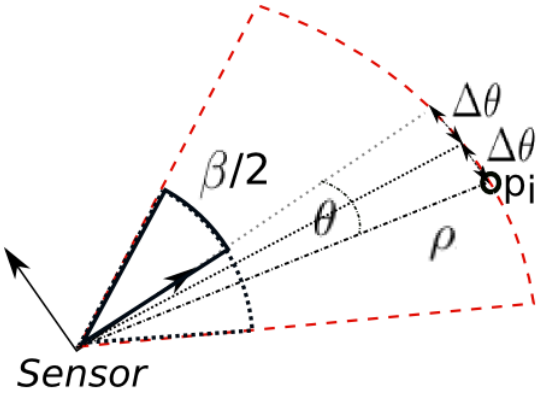


Figura 3.3: Ejemplo de cálculo de punto a partir de coordenadas polares

Por otra parte, si la velocidad del robot es nula (robot parado), no se recogerá ningún dato proveniente de estos sensores. Del mismo modo, si la distancia devuelta por algún sensor

supera cierto límite, no se procederá a la discretización de puntos pertinente en su arco de barrido. La justificación de esta estrategia parte de lo siguiente:

- Votación en movimiento: la lógica de la Transformada de Hough es reforzar aquellas características del entorno que el robot va detectando a medida que se mueve. Si el robot está parado, el algoritmo no debería recoger los datos de los sensores para evitar reforzar aquellas características que, de estar moviéndose el robot, pasarían desapercibidas. En resumen, la detección sin movimiento puede generar falsos positivos que podrían evitarse.

- Límite de distancia: a partir de cierta distancia, los sónars empiezan a perder precisión, y los datos obtenidos comienzan a perder calidad además de utilidad de cara al modelado del entorno, consecuentemente.

En la salida de esta primera etapa se dispondrá de la lista de puntos P discretizados, posicionados con respecto al sensor de ultrasonidos pertinente. Antes de pasar a La siguiente etapa, conviene introducir un concepto clave en la implementación de la Transformada de Hough: el **origen relativo**.

El espacio de Hough, mencionado en la sección 3.1, debe estar vinculado a un sistema de coordenadas **SC** en base al cual se parametrizan las diferentes características (líneas y puntos) que van siendo detectadas en el entorno del robot. De esta necesidad surge el origen relativo, que inicialmente coincide con el SC del origen absoluto, de donde partió el robot originariamente.

3.2.2. Etapa 2: Cálculo de posición con respecto a origen relativo

En esta etapa es necesario presentar los diferentes sistemas de coordenadas o de referencia (SC) con los que se trabajará, así como las letras o símbolos que los representarán de aqui en adelante:

- Sensor (S).

- Robot (R).

- Origen absoluto (a), de donde partió el robot inicialmente.

- Origen relativo (r).

Un esquema representativo podría ser el que se muestra en la figura 3.4. En esta figura se especifican los SC utilizados según la notación propuesta en [Asada,1986], donde A^i_j es la matriz en coordenadas homogéneas que transforma del SC i al j.

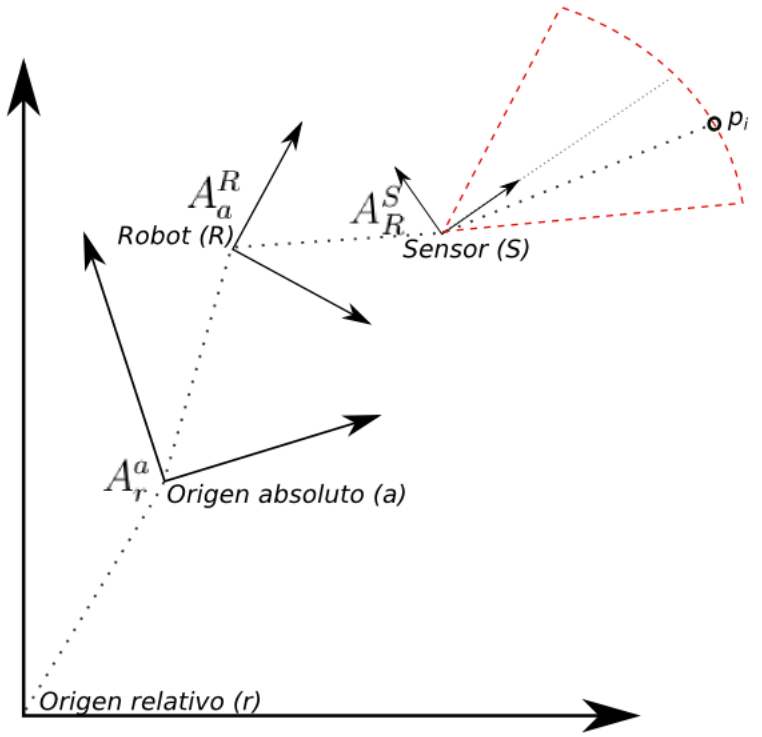


Figura 3.4: Representación de los distintos sistemas de coordenadas

En la etapa anterior, quedaba determinada la posición de los puntos con respecto al sensor correspondiente. En esta etapa se procederá a calcular lo que a partir de ahora se denominará **Punto q**. En el espacio de Hough, las características (líneas y puntos) parametrizadas equivaldrán a los parámetros polares (ρ, θ) de los Puntos q asociados a líneas y puntos. Para cada una de estas características, el cálculo del Punto q es diferente, tal y como se muestra en los siguientes apartados.

Caso de los puntos

En el caso de los puntos, el *Punto q* se corresponde directamente con la posición del punto, posicionado con respecto al origen relativo. En la figura 3.5 se expone lo dicho. Por su parte, en el algoritmo 2 se presenta la forma de proceder.

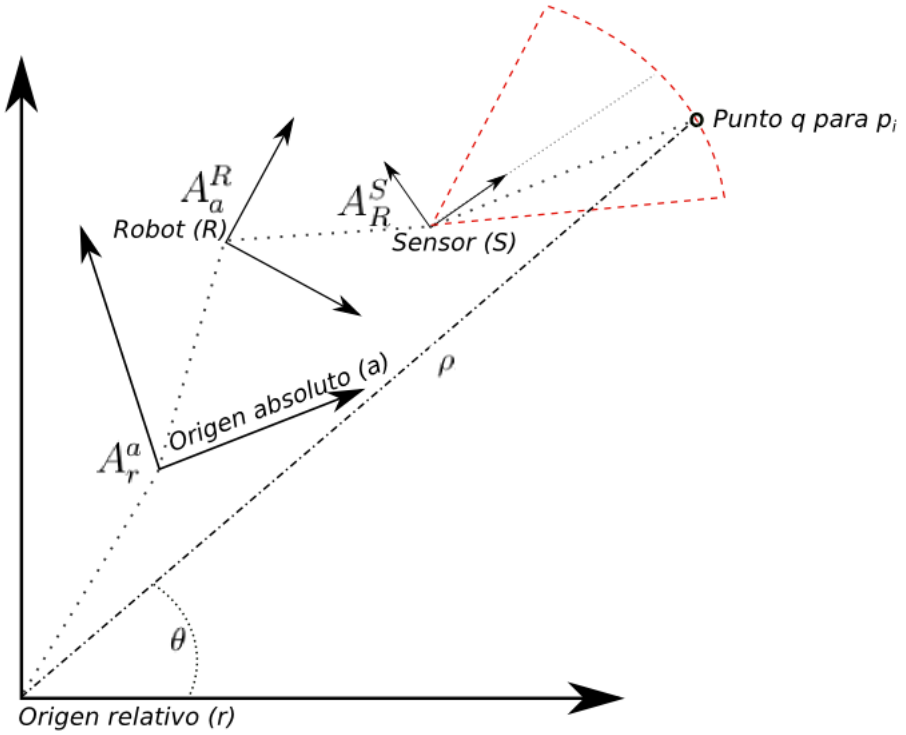


Figura 3.5: Cálculo de *Punto q* para caso de los puntos

Algoritmo 2 Cálculo de *Punto q* para caso de puntos

Entrada: Lista de puntos en coordenadas cartesianas P{p_1,p_2,...p_n}

Salida: Lista de Q{q_1,q_2,...q_m} para puntos

1: **para** cada p_i **hacer**

2: $q_i = p_i^r \leftarrow A_r^a \cdot A_a^R \cdot A_R^S \cdot p_i^S$

3: Añadir q_i a Q

4: **fin para**

5: **devolver** Q

La salida de esta subetapa proporcionará una lista de *Puntos q* en coordenadas cartesianas, cada uno asociado a un punto diferente.

Caso de las líneas

Las líneas son representadas como líneas infinitas y parametrizadas en el espacio de Hough a modo de coordenadas polares (ρ, θ) como en el caso de los puntos.

El cálculo del Punto q en este caso es algo más complejo, dado que no se trata de un cálculo tan directo. A grandes rasgos, el algoritmo 3, acompañado de la figura 3.6, muestra como se pretende hallar la tangente de cada punto con respecto al sensor. Una vez determinada, se

pasa a determinar el *Punto q*, que en este caso se corresponderá con aquella posición tal que la tangente anterior se hace normal al origen relativo.

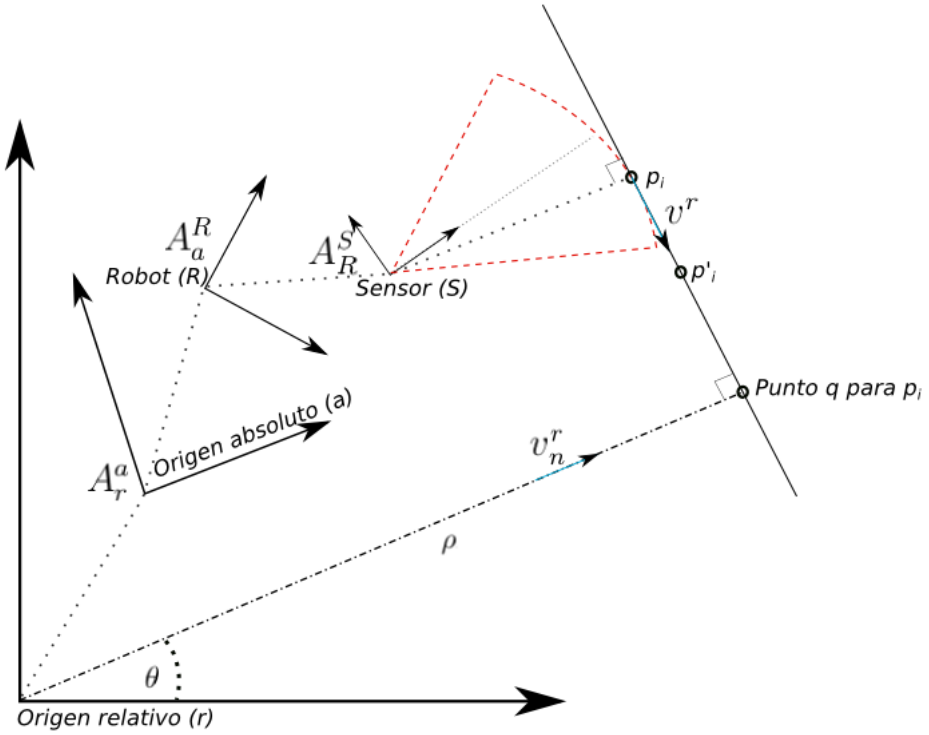


Figura 3.6: Cálculo de *Punto q* para caso de las líneas

De forma análoga a la subetapa anterior, en este caso la salida consistirá en una lista de *Puntos q* en coordenadas cartesianas, esta vez asociados a las diferentes líneas.

Tal y como se aprecia, cada p_i del conjunto de puntos discretizados se corresponderá con un *Punto q* como línea y un *Punto q* como punto.

3.2.3. Etapa 3: Discretización e inserción en matrices de la Transformada de Hough

Una vez determinados los Puntos q, ya es posible insertarlos a modo de coordenadas polares en el espacio de Hough. Para la implementación del espacio de Hough se crearán dos matrices: **la matriz de la Transformada de Hough para líneas** y **la matriz de la Transformada de Hough para puntos**.

Por tanto, Las matrices consisten en una representación discretizada de parámetros polares. Cada matriz dispondrá de valores de distancia como columnas (ρ, en metros) y de valores de ángulo como filas (θ, en grados). Asimismo, las dimensiones dependerán de los siguientes factores:

- Número de filas: (360/grados por celda) + 1. La representación en grados y su carácter cíclico evita problemas de cara a inserción para filas.

Algoritmo 3 Cálculo de *Punto q* para caso de líneas

Entrada: Lista de puntos en coordenadas cartesianas P$\{p_1,p_2,...p_n\}$

Salida: Lista de Q$\{q_1,q_2,...q_m\}$ para líneas

1: **para** cada p_i **hacer**

2: $p_i^r \leftarrow A_r^a \cdot A_a^R \cdot A_R^S \cdot p_i^S$

3: $v^S \leftarrow \begin{bmatrix} -p_{iy}^S \\ p_{ix}^S \end{bmatrix}$

4: $p_i^{'S} \leftarrow p_i^S + v^S$

5: $p_i^{'r} \leftarrow A_r^a \cdot A_a^R \cdot A_R^S \cdot p_i^{'S}$

6: $v^r \leftarrow p_i^{'r} - p_i^r$

7: $v_n^r \leftarrow \begin{bmatrix} -v_y^r \\ v_x^r \end{bmatrix}$

8: $u_n^r \leftarrow \frac{v_n^r}{||v_n^r||}$

9: $\rho \leftarrow p_i^r \cdot u_n^r$

10: $q_i \leftarrow \rho \cdot u_n^r$

11: Añadir q_i a Q

12: **fin para**

13: **devolver** Q

- Número de columnas: (d_{max}/metros por celda) + 1, donde d_{max} representa el límite de distancia tal que, si es rebasado, conducirá al proceso de transformación de la matriz que corresponda, que se explicará más adelante.

En el algoritmo 5 se muestra el procedimiento general de inserción. Cada conjunto de *Puntos q* se insertará en la matriz correspondiente, de las dos posibles, según estén vinculados a líneas o a puntos. De esta forma, en cada inserción se aumentará el número de votos de la celda vinculada al Punto q en cuestión.

Algoritmo 4 Función **PosicionEnMatriz()**

Entrada: Punto q_i

Número de filas y columnas de matriz *(n,m)*

Distancia máxima d_{max}

Salida: Fila y columna para indicar posición en matriz *(f,c)*

1: $\rho_i \leftarrow ||q_i||$

2: $\theta_i \leftarrow atan2(q_{iy}/q_{ix})$

3: $c \leftarrow \rho_i \cdot \frac{m}{d_{max}}$

4: $f \leftarrow \theta_i \cdot \frac{n}{360}$

5: **devolver** (f,c)

Transformación de las matrices de la Transformada de Hough

Este mecanismo se presenta imprescindible, y no sólo por la finitud de la matriz, sino por la acumulación de errores de odometría a medida que el robot se va alejando del sistema de referencia del que parte.

Cada una de las dos matrices de la Transformada de Hough tiene su propio sistema de referencia, consistente en el origen relativo ya mencionado. Asimismo, el origen relativo coincidirá con la posición del robot en el instante en el que se haya detectado un *Punto q* con una distancia que supere el límite anteriormente expuesto. Dependiendo de si el *Punto q* es propio de una línea o de un punto, habrá de realizarse el proceso de transformación de la matriz pertinente.

El procedimiento de la transformación pretende reubicar cada uno de los *Puntos q* cuyas celdas disponen de algún voto con respecto al nuevo origen relativo generado. La idea general de la transformación se muestra en la figura 3.7.

Algoritmo 5 Inserción de *Punto q* en Matriz de la Transformada de Hough

Entrada: Lista de puntos Q{q_1,q_2,...q_z}

Matriz de la Transformada de Hough M(n,m)

Distancia máxima d_{max}

Número máximo de votos por celda v_{max}

Salida: Matriz de la Transformada de Hough M(n,m)

Nuevo origen relativo A_a^r //si se provoca transformación de M

1: **para** cada q_i **hacer**
2: $(f, c) \leftarrow$ **PosicionEnMatriz(**$q_i, (n, m), d_{max}$**)** //llamada a algoritmo 4
3: **si** $(c \leq m) y (M[f, c] \leq v_{max})$ **entonces**
4: $M[f, c] \leftarrow M[f, c] + 1$
5: **si no**
6: **si** $c > m$ **entonces**
7: $A_a^r \leftarrow A_a^R$
8: M $\leftarrow$ **Transformar(**q_i, M**)** //Llamada a algoritmos 7 u 8 según caso
9: **fin si**
10: **fin si**
11: **fin para**
12: **devolver** (M, A_a^r)

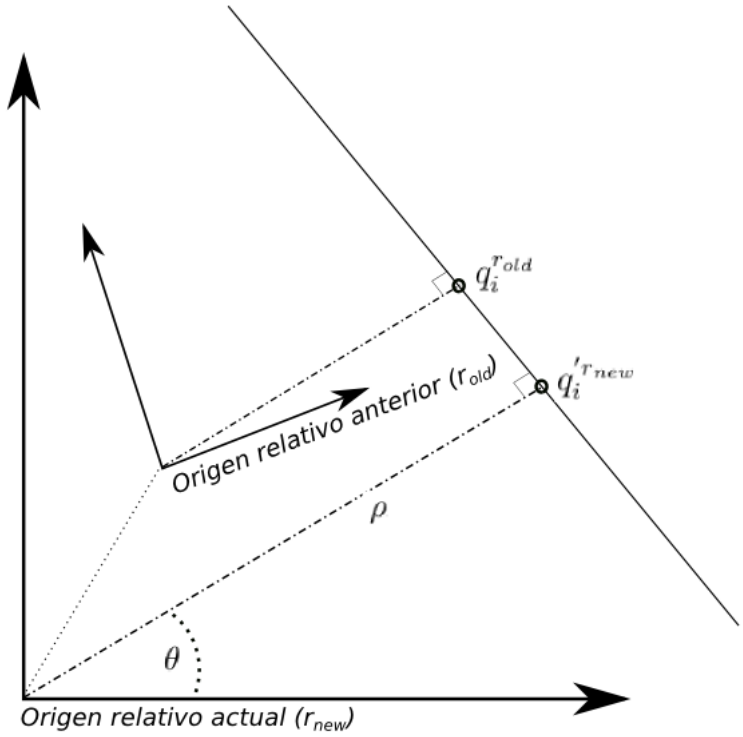


Figura 3.7: Fundamento de recálculo de posición de *Punto q*

Algoritmo 6 Función **PuntoqDeCelda()**

Entrada: Fila y columna para indicar posición en matriz *(f,c)*

Número de filas y columnas de matriz *(n,m)*

Distancia máxima d_{max}

Salida: Punto asociado a la celda q_i

1: $\rho \leftarrow c \cdot \frac{d_{max}}{m}$

2: $\theta \leftarrow f \cdot \frac{360}{n}$

3: $q_i \leftarrow \rho \cdot \begin{bmatrix} cos_\theta \\ sin_\theta \end{bmatrix}$

4: **devolver** q_i

Para ambos casos, el primer paso consiste en ubicar el *Punto q* que generó la transformación en la nueva matriz M_{new}, para seguidamente reubicar todos los Puntos q de las celdas de la matriz original M_{old} con algún voto en la nueva matriz M_{new}. Esta última, cuyos datos estarán referidos con respecto al nuevo origen relativo r_{new}, mantendrá todos los Puntos q que estaban parametrizados en la antigua matriz M_{old} con el mismo número de votos que entonces y en las nuevas posiciones calculadas.

Algoritmo 7 Función **Transformar()** para caso de puntos

Entrada: Punto que provocó la transformación $q_i^{r_{old}}$

Matriz de la Transformada de Hough $M_{old}(n,m)$

Distancia máxima d_{max}

Número máximo de votos por celda v_{max}

Salida: Matriz de la Transformada de Hough $M_{new}(n,m)$

1: $A_{r_{new}}^{r_{old}} \leftarrow A_R^a \cdot A_a^{r_{old}}$
2: $q_i^{'r_{new}} \leftarrow A_{r_{new}}^{r_{old}} \cdot q_i^{r_{old}}$
3: $(f,c) \leftarrow$ **PosicionEnMatriz**$(q_i^{'r_{new}}, (n,m), d_{max})$ //Llamada a algoritmo 4
4: $M_{new}[f,c] \leftarrow 1$
5: **para** cada $i \in [1,n]$ **hacer**
6: **para** cada $j \in [1,m]$ **hacer**
7: **si** $M_{old}[i,j] \neq 0$ **entonces**
8: $q_i^{r_{old}} \leftarrow$ **PuntoqDeCelda**((i,j),(n,m),d_{max}) //Llamada a algoritmo 6
9: $q_i^{'r_{new}} \leftarrow A_{r_{new}}^{r_{old}} \cdot q_i^{r_{old}}$
10: $(f,c) \leftarrow$ **PosicionEnMatriz**$(q_i^{'r_{new}}, (n,m), d_{max})$ //Llamada a algoritmo 4
11: $M_{new}[f,c] \leftarrow M_{old}[i,j]$
12: **fin si**
13: **fin para**
14: **fin para**
15: **devolver** M_{new}

Algoritmo 8 Función **Transformar()** para caso de líneas

Entrada: Punto que provocó la transformación $q_i^{r_{old}}$

Matriz de la Transformada de Hough $M_{old}(n,m)$

Distancia máxima d_{max}

Número máximo de votos por celda v_{max}

Salida: Matriz de la Transformada de Hough $M_{new}(n,m)$

1: $A_{r_{new}}^{r_{old}} \leftarrow A_R^a \cdot A_a^{r_{old}}$

2: $q_i^{r_{new}} \leftarrow A_{r_{new}}^{r_{old}} \cdot q_i^{r_{old}}$

3: $v_t^{r_{new}} \leftarrow R_{r_{new}}^a \cdot R_a^{r_{old}} \cdot \frac{q_i^{r_{old}}}{||q_i^{r_{old}}||}$

4: $\rho \leftarrow q_i^{r_{new}} \cdot v_t^{r_{new}}$

5: $q_i^{'r_{new}} \leftarrow \rho \cdot v_t^{r_{new}}$

6: $(f,c) \leftarrow$ **PosicionEnMatriz**$(q_i^{'r_{new}}, (n,m), d_{max})$ //Llamada a algoritmo 4

7: $M_{new}[f,c] \leftarrow 1$

8: **para** cada $i \in [1,n]$ **hacer**

9: **para** cada $j \in [1,m]$ **hacer**

10: **si** $M_{old}[i,j] \neq 0$ **entonces**

11: $q_i^{r_{old}} \leftarrow$ **PuntoqDeCelda**((i,j),(n,m),d_{max}) //Llamada a algoritmo 6

12: $q_i^{r_{new}} \leftarrow A_{r_{new}}^{r_{old}} \cdot q_i^{r_{old}}$

13: $v_t^{r_{new}} \leftarrow R_{r_{new}}^a \cdot R_a^{r_{old}} \cdot \frac{q_i^{r_{old}}}{||q_i^{r_{old}}||}$

14: $\rho \leftarrow q_i^{r_{new}} \cdot v_t^{r_{new}}$

15: $q_i^{'r_{new}} \leftarrow \rho \cdot v_t^{r_{new}}$

16: $(f,c) \leftarrow$ **PosicionEnMatriz**$(q_i^{'r_{new}}, (n,m), d_{max})$ //Llamada a algoritmo 4

17: $M_{new}[f,c] \leftarrow M_{old}[i,j]$

18: **fin si**

19: **fin para**

20: **fin para**

21: **devolver** M_{new}

3.2.4. Etapa 4: Valoración de persistencia

Cuando se trata de realizar un modelo del entorno, se hace indispensable que el robot "olvide". En términos más técnicos, olvidar equivaldría a penalizar aquellas características del entorno percibidas de manera puntual, dado que si no se vuelven a percibir seguramente es que no existen realmente.

Esta utilidad trata de hacer destacar aquellas líneas y puntos cuyas respectivas celdas son actualizadas con cierta frecuencia, con tal de aumentar las posibilidades de descartar casos falsos. Con tal objetivo, se centra en decrementar votos de aquellas celdas de sendas matrices que cumplan ciertas condiciones. Su forma de proceder sigue el algoritmo 9.

Algoritmo 9 Persistencia

Entrada: Matriz de la Transformada de Hough M(n,m)
Distancia máxima con respecto a robot d_{max}
Número de votos a decrementar d_{votos}
Tiempo máximo sin actualizar t_{max}

Salida: Matriz de la Transformada de Hough M(n,m)

para cada $i \in [1, n]$ **hacer**
  para cada $j \in [1, m]$ **hacer**
    si $M[i, j] \neq 0$ **entonces**
      $q_i^r \leftarrow$ **PuntoqDeCelda**((i,j),(n,m),d_{max}) //Llamada a algoritmo 6
      $A_R^r \leftarrow A_R^a \cdot A_a^r$
      $q_i^R \leftarrow A_R^r \cdot q_i^r$
      Primero se evalúa la distancia con el robot del *Punto q* correspondiente
      si $q_i^R > d_{max}$ **entonces**
        $M[i, j] \leftarrow max(0, M[i, j] - d_{votos})$
      fin si
      Luego se evalúa el tiempo que lleva sin actualizarse
      si $(t_{M[i,j]} - t_{actual}) \geq t_{max}$ **entonces**
        $M[i, j] \leftarrow max(0, M[i, j] - d_{votos})$
      fin si
    fin si
  fin para
fin para
devolver M

Es importante hacer notar que se debe mantener una marca de tiempo por cada celda de ambas matrices si se desea implementar la utilidad de la persistencia.

3.2.5. Etapa 5: Extracción de líneas y puntos en matrices de la Transformada de Hough

Esta etapa se puede considerar el primer filtro para el contenido de las matrices, y en su salida se dispondrá de las líneas y puntos con más votos y, por tanto, con más evidencia sensorial. En ésta se examina cada una de las celdas de ambas matrices, con el propósito de hallar líneas y puntos que superen cierto número de votos. El algoritmo 10 refleja la forma de proceder con tal cometido.

Algoritmo 10 Umbralizado de puntos y líneas a partir de la Transformada de Hough

Entrada: Matriz de la Transformada de Hough M(n,m)

Distancia máxima d_{max}

Mínimo número de votos v_{min}

Salida: Lista de puntos o líneas -según caso- en coordenadas polares S$\{s_1,s_2,...s_z\}$

1: **para** cada $i \in [1, n]$ **hacer**
2: **para** cada $j \in [1, m]$ **hacer**
3: **si** $M[i, j] \geq v_{min}$ **entonces**
4: $s_\rho \leftarrow j \cdot \frac{d_{max}}{m}$
5: $s_\theta \leftarrow i \cdot \frac{360}{n}$
6: Añadir s a S
7: **fin si**
8: **fin para**
9: **fin para**
10: **devolver** S

3.2.6. Etapa 6: Fusión de las líneas y puntos extraídos

La fusión o "merge" ejerce como un nuevo filtro para el objetivo final de obtener aquellas líneas y puntos del entorno que sean más "fidedignos". A grandes rasgos, pretende aunar líneas suficientemente cercanas entre sí, de igual forma que con los puntos, bajo ciertas condiciones. Este proceso permitirá disponer de las líneas y puntos más representativos de los diferentes conjuntos que hayan podido fusionarse. Cabe indicar, como sugiere el nombre de esta etapa, que la fusión sólo acogerá aquellas líneas y puntos extraídos de las matrices de la Transformada de Hough, en la etapa anterior (ver 3.2.5).

Existen diversas formas de implementar la fusión, y a continuación se muestra la utilizada para el caso de ambas características.

Caso de las líneas

En este caso, el algoritmo a seguir es el 11.

Caso de los puntos

En el caso de la fusión de puntos, se sigue el algoritmo 14.

En el caso de la fusión de líneas, finalmente resulta una lista con aquellas líneas derivadas de la fusión de subconjuntos de líneas, en parámetros polares. Por su parte, en el caso de los puntos la salida será una lista donde cada elemento será el punto derivado de la fusión junto al subconjunto de puntos que lo generaron, en coordenadas cartesianas.

3.2.7. Etapa 7: Hipótesis de líneas

Esta etapa supone el último filtro de líneas, y se basa en las líneas y puntos **que han resultado de la etapa de fusión**. Su objetivo es determinar qué líneas tienen mayor evidencia sensorial, evaluando para ello la distribución de puntos alrededor de éstas. Una acumulación de puntos con una distribución determinada podrá aumentar las posibilidades de que una línea cercana a éstos exista realmente en el entorno.

En la figura 3.8 se puede apreciar la idea en la que se fundamenta esta etapa, implementada en el algoritmo 17. Cada línea l se compara con cada uno de los puntos p^{si}, y si son suficientemente cercanos, pasará a evaluarse la distribución de los puntos $\{p_1^{si}, ...p_n^{si}\}$ cuya fusión generó dicho p^{si} (cabe hacer notar que el subíndice diferencia a cada p^{si} de cada uno de los puntos del subconjunto que lo generó). Finalmente, si la distribución satisface ciertas condiciones, la línea l superará el filtro de esta etapa, aumentando así sus probabilidades de ser real.

De cara a la nomenclatura del algoritmo 17, L será la lista de líneas l, mientras que L_{reg} contendrá cada uno de los puntos p^{si} junto a su lista de puntos $P^{si}\{p_1^{si}, ...p_n^{si}\}$, esto es, L_{reg}=$\{\{p^{s1}$,$\{p_1^{s1}, ..., p_n^{s1}\}\}$,$\{p^{s2}$,$\{p_1^{s2}, ...p_m^{s2}\}\}\}$.

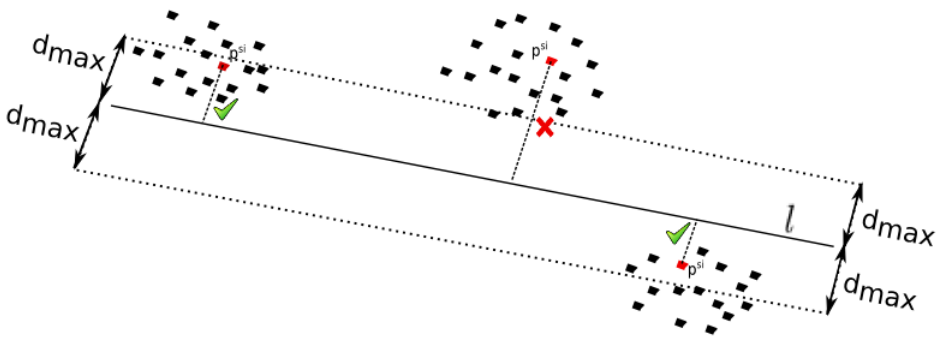


Figura 3.8: Consideración de nubes de puntos de zonas con p^{si} suficientemente cercano a línea l

Algoritmo 11 Fusión de líneas

Entrada: Lista de líneas umbralizadas L_e

Mínimo número de líneas para fusionar l_{min}

Salida: Lista de líneas obtenidas tras fusión L_s

1: **mientras** Tamaño(L_e)$\neq 0$ **hacer**
2: $L_{nv}(1) \leftarrow Extrae(L_e(1))$ //*Extrae* desplaza elementos tras extracción
3: **mientras** Tamaño(L_{nv}) $\neq 0$ **hacer**
4: $pos_{L_e} \leftarrow 1$
5: **mientras** $pos_{L_e} \leq$ Tamaño(L_e) **hacer**
6: // En la siguiente condición se realiza una llamada al algoritmo 12
7: **si** $DistanciaAceptable(L_e(pos_{L_e}), L_{nv}(1))$ **entonces**
8: $l \leftarrow Extrae(L_e(pos_{L_e}))$
9: Añadir l a L_{nv}
10: **si no**
11: $pos_{L_e} \leftarrow pos_{L_e} + 1$
12: **fin si**
13: **fin mientras**
14: $l \leftarrow Extrae(L_{nv}(1))$
15: Añadir l a L_v
16: **fin mientras**
17: **si** Tamaño(L_v) $\geq l_{min}$ **entonces**
18: $l \leftarrow Fusion(L_v)$ //llamada a algoritmo 13
19: Añadir l a L_s
20: **fin si**
21: Vacia(L_v) //se eliminan todos los elementos que contuviera la lista
22: **fin mientras**
23: **devolver** L_s

Algoritmo 12 DistanciaAceptable()

Entrada: Elemento en coordenadas polares e_1

Elemento en coordenadas polares e_2

Diferencia máxima permitida para ρ ρ_{max}

Diferencia máxima permitida para θ θ_{max}

Salida: Devuelve **verdadero** si se cumplen ambas condiciones

1: **si** $(e_{1\rho} - e_{2\rho} \leq \rho_{max})$ y $(e_{1\theta} - e_{2\theta} \leq \theta_{max})$ **entonces**

2: **devolver verdadero**

3: **fin si**

4: **devolver falso**

Algoritmo 13 Fusion()

Entrada: Lista de líneas o puntos en coordenadas polares L_e

Salida: Devuelve resultado de la fusión *ganador*

1: $L_{max} \leftarrow$ MáximoDeVotos(L_e) //devuelve lista de elementos que tengan número de votos máximo

2: $n \leftarrow$Tamaño(L_{max})

3: **devolver** $\frac{1}{n} \cdot \sum_{i=1}^{n} L_{max_i}$

Algoritmo 14 Fusión de puntos

Entrada: Lista de puntos umbralizados L_e

Mínimo número de puntos para fusionar p_{min}

Salida: Lista de puntos obtenidos tras fusión L_s

1: **mientras** Tamaño(L_e)$\neq 0$ **hacer**
2: $L_{nv}(1) \leftarrow Extrae(L_e(1))$ //*Extrae* desplaza elementos tras extracción
3: **mientras** Tamaño(L_{nv}) $\neq 0$ **hacer**
4: $pos_{L_e} \leftarrow 1$
5: **mientras** $pos_{L_e} \leq$ Tamaño(L_e) **hacer**
6: // En la siguiente condición se realiza una llamada a los algoritmos 12, 15 y 16
7: **si** $DistanciaAceptable(L_e(pos_{L_e}), L_{nv}(1))$ y **DistanciaEnCartesianasAceptable**($L_e(pos_{L_e})$,**PuntoMedio**(L_{nv}, L_v)) **entonces**
8: $p \leftarrow Extrae(L_e(pos_{L_e}))$
9: Añadir p a L_{nv}
10: **si no**
11: $pos_{L_e} \leftarrow pos_{L_e} + 1$
12: **fin si**
13: **fin mientras**
14: $p \leftarrow Extrae(L_{nv}(1))$
15: Añadir p a L_v
16: **fin mientras**
17: **si** Tamaño(L_v) $\geq l_{min}$ **entonces**
18: $p \leftarrow Fusion(L_v)$ //llamada a algoritmo 13
19: Añadir p a L_s
20: **fin si**
21: Vacia(L_v) //se eliminan todos los elementos que contuviera la lista
22: **fin mientras**
23: **devolver** L_s

Algoritmo 15 DistanciaEnCartesianasAceptable()

Entrada: Elemento en coordenadas cartesianas e_1

Elemento en coordenadas cartesianas e_2

Diferencia máxima permitida entre elementos d_{max}

Salida: Devuelve **verdadero** si se cumple la condición de distancia

1: $\Delta_e \leftarrow e_1 - e_2$

2: **si** $||\Delta_e|| \leq d_{max}$ **entonces**

3: **devolver verdadero**

4: **fin si**

5: **devolver falso**

Algoritmo 16 PuntoMedio()

Entrada: Lista en coordenadas cartesianas L_{e_1}

Lista en coordenadas cartesianas L_{e_2}

Salida: Devuelve punto medio p_m

1: $L \leftarrow L_{e_1} \cup L_{e_2}$

2: $n \leftarrow$Tamaño(L)

3: **devolver** $p_m \leftarrow \frac{1}{n} \cdot \sum_{i=1}^{n} L_i$

Algoritmo 17 Hipótesis de líneas

Entrada: Lista de líneas $L\{l_1, l_2, ...l_n\}$

Lista de regiones L_{reg}

Distancia máxima permitida entre punto (p^s) y línea(l) $dist_{max}$

Mínimo valor de distribución de puntos en eje horizontal x_{min}

Máximo valor de distribución de puntos en eje vertical y_{max}

Salida: Lista de líneas S

1: **para** cada l_i **hacer**
2: **para** cada p^{si} **hacer**
3: **si Distancia**$(l_i, p^{si}) \leq dist_{max}$ **entonces**
4: Añadir P^{si} a lista D //D será una lista de subconjuntos de puntos p_x^{si}, vacía en un principio.
5: **fin si**
6: **fin para**
7: $(distrib_x, distrib_y) \leftarrow Distribucion(D, l_i)$ //llamada a algoritmo 18
8: **si** $distrib_x \geq x_{min}$ y $distrib_y \leq y_{max}$ **entonces**
9: Añadir l_i a S
10: **fin si**
11: **fin para**
12: **devolver** S

Algoritmo 18 Distribucion()

Entrada: Linea en coordenadas cartesianas l

Lista de sublistas de puntos D

Salida: Distribución de puntos en eje horizontal $distrib_x$

Distribución de puntos en eje vertical $distrib_y$

1: $v_u \leftarrow VectorUnitario(L)$

2: $v_n \leftarrow \begin{bmatrix} -v_{u_y} \\ v_{u_x} \end{bmatrix}$

3: totalVotos $\leftarrow \sum p^{si}_{x_{votos}}$ //suma de votos de todos los puntos de los subconjuntos de D

4: puntoMedio $\leftarrow \frac{1}{totalVotos} \cdot \sum p^{si}_{x} \cdot p^{si}_{x_{votos}}$ //media aritmética ponderada

5: distancia $\leftarrow (puntoMedio - l_{p1}) \cdot v_n$

6: puntoMedioSobreLinea $\leftarrow puntoMedio - distancia \cdot v_n$

7: **para** cada p^{si}_j **hacer**

8: aux $\leftarrow p^{si}_j - puntoMedioSobreLinea$

9: Añadir (aux $\cdot\ v_u$) a $D_x\{d_{x1}, d_{x2}, ...d_{xn}\}$//lista de distancias en horizontal

10: Añadir (aux $\cdot\ v_n$) a $D_y\{d_{y1}, d_{y2}, ...d_{ym}\}$ //lista de distancias en vertical

11: **fin para**

12: $x \leftarrow$ Tamaño(D_x)

13: $\overline{x} \leftarrow \frac{1}{x} \cdot \sum_{i=1}^{x} d_{xi}$ //media aritmética

14: $distrib_x \leftarrow \frac{1}{x}\sqrt{\sum_{i=1}^{x} \cdot (d_{xi} - \overline{x})^2}$ //desviación típica

15: $y \leftarrow$ Tamaño(D_y)

16: $\overline{y} \leftarrow \frac{1}{y} \cdot \sum_{i=1}^{y} d_{yi}$

17: $distrib_y \leftarrow \frac{1}{y}\sqrt{\sum_{i=1}^{y} \cdot (d_{yi} - \overline{y})^2}$

18: **devolver** $(distrib_x, distrib_y)$

Aquí concluye el estudio de las etapas de desarrollo de los sensores ultrasónicos. De esta forma, al finalizar todas las etapas se dispondrá de aquellas líneas fusionadas que tengan alrededor una distribución de puntos adecuada. Asimismo, de la etapa anterior se extraerán los *superpuntos* fusionados a partir de puntos probables.

Finalmente se incluye el diagrama 3.9, donde se resumen a grandes rasgos las diferentes etapas de desarrollo implicadas y la relación entre éstas.

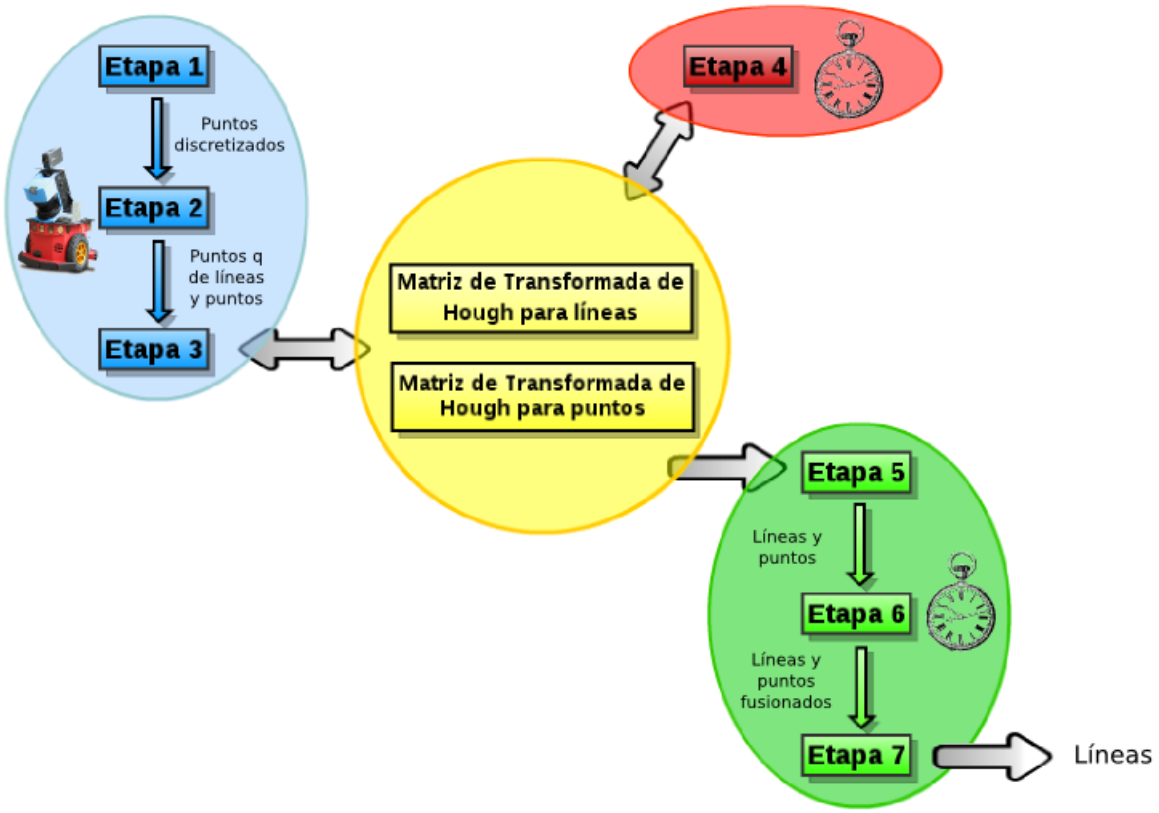


Figura 3.9: Diagrama de etapas implicadas para el caso de los sónars

3.3. Estudio de las etapas de desarrollo para el láser

Además del modelado de los sensores sónar, se procederá a implementar un algoritmo concreto para interpretar los datos obtenidos a partir del sensor láser. El algoritmo en particular es el *IEPF* o *Iterative End Point Fit* extendido, una variante del algoritmo conocido como *split & merge*.

Por tanto, este apartado se referirá a las diferentes etapas a desarrollar para alcanzar el objetivo expuesto:

- Etapa 1: Detección de puntos del sensor
- Etapa 2: División de conjuntos de puntos y generación de líneas
- Etapa 3: Fusión de líneas

3.3.1. Etapa 1: Detección de puntos del sensor

En esta primera etapa se determinará la distancia con respecto al sensor de cada punto captado por el láser. Si esta distancia es menor que cierto umbral dicho punto será aceptado como válido.

Cabe señalar que aunque el sensor láser trabaje con las tres dimensiones espaciales, sólo es necesario ubicar cada punto en el plano, dado el entorno de desarrollo de este proyecto.

3.3.2. Etapa 2: División de conjuntos de puntos y generación de líneas

Esta segunda etapa se presenta como la más relevante en el ámbito del láser, y es que permite hallar las diferentes líneas del mapa en base a la distribución de puntos captados. Para ello se implementará el algoritmo *IEPF* citado anteriormente, cuyo procedimiento es el referido en el algoritmo 19 para una implementación recursiva, relacionado asimismo con la 3.10.

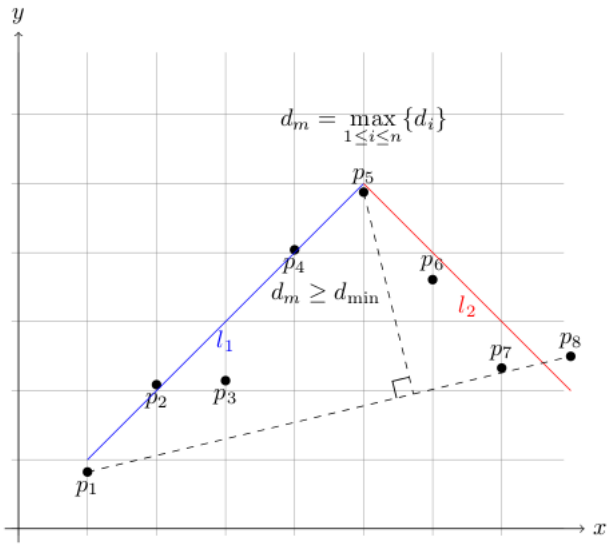


Figura 3.10: Algoritmo IEPF (extraída de [Fernández-Perdomo,2009]

3.3.3. Etapa 3: Fusión de líneas

En esta última etapa se procede a la fusión de las líneas de la etapa anterior tal y como en el caso de la correspondiente etapa para los sónars (sección 3.2.6), pudiendo disponer así tanto de las líneas originales a partir de los puntos como del resultado de la fusión de las mismas.

Algoritmo 19 Iterative End-Point Fit extendido (IEPFext())

Entrada: Lista de puntos en coordenadas cartesianas P$\{p_1, p_2, ...p_n\}$

Distancia mínima permitida para el punto P_{max} (punto más distante) con respecto a la línea d_{min}

Longitud mínima de línea permitida l_{min}

Salida: Lista de líneas L

1: $L \leftarrow \emptyset$
2: l $\leftarrow$ línea desde p_1 hasta p_n
3: $(P_{max}, d_{max}) \leftarrow$ punto con máxima distancia d_{max} a l
4: **si** $d_{max} \leq d_{min}$ **entonces**
5: **si** $||P_1 - P_n|| \leq l_{min}$ **entonces**
6: **devolver** 0
7: **si no**
8: Añadir $P_2...P_{n-1}$ a L
9: **devolver** L
10: **fin si**
11: **si no**
12: **devolver** **IEPFext**($\{P_1,...P_m\}$,d_{min},l_{min}) $\cup$ **IEPFext**($\{P_{m+1},...P_n\}$,d_{min},l_{min})
13: **fin si**

Capítulo 4

Diseño de prototipos, implementación y pruebas

En este nuevo capítulo se abordará el diseño y la implementación tanto del prototipo en Matlab como del prototipo final en CoolBot, desde una perspectiva enfocada tanto a la estructura general como a nivel de detalle, acompañando convenientemente mediante capturas de pantalla que aporten una mayor transparencia.

Éste comprenderá dos secciones bien delimitadas:

- Prototipo en Matlab: implementado en Matlab con tal de vislumbrar posibles problemas antes de la implementación final.
- Prototipo final en CoolBot: implementación en framework CoolBot de diversas funcionalidades que satisfarán la problemática expuesta en las etapas del capítulo anterior.

4.1. Prototipo en Matlab

Aunque se han realizado más prototipos, éstos se han generado como evolución del prototipo inicial hasta llegar al prototipo final, de manera que pasarán a estudiarse éstos dos únicamente.

4.1.1. Versión inicial

El fundamento de este primer diseño se basa en los siguientes objetivos:

1. Implementar los cambios entre los distintos sistemas de referencia.
2. Imprimir matriz de la Transformada de Hough tras insertar el *Punto q* hallado.

En base al índice de etapas de desarrollo para los sónars del capítulo de *Análisis del problema*, a grandes rasgos acogería las siguientes con ciertas restricciones, que se explicarán posteriormente:

- Etapa 1: Discretización en el arco de barrido y cálculo de posición con respecto a sensor (sección 3.2.1)
- Etapa 2: Cálculo de posición con respecto a origen relativo (sección 3.2.2)
- Etapa 3: Discretización e inserción en matrices de la Transformada de Hough (sección 3.2.3)

Asimismo, las citadas restricciones serán las siguientes:

- No se considerará el sistema de referencia del origen relativo.
- Sólo se atenderá a un único sensor ultrasónico para el robot, y a un único punto discretizado en el arco de barrido de éste.
- Se hallará el *Punto q* asumiendo únicamente el caso de que se tratara de una línea, dada la mayor complejidad de éste frente al caso de los puntos.

La justificación de los límites mencionados se encuentra en que la implementación del caso genérico, a partir de este caso particular, se hace prácticamente trivial.

El procedimiento de este prototipo sigue los siguientes pasos secuencialmente:

1. Petición de datos al usuario. En la figura 4.1 aparece el diálogo con el usuario tras haber introducido todos los datos, que son los que siguen:
   1.1. Posición del punto P con respecto al sensor.
   1.2. Posición del sensor con respecto al robot.
   1.3. Posición del robot con respecto al origen absoluto, así como su inclinación en grados.

```
>> HT
> Datos acerca del punto a evaluar:
Introduzca la posición en x del punto a evaluar
3
Introduzca la posición en y del punto a evaluar
4
> Datos acerca del sensor:
Introduzca la posición del sensor en x
2
Introduzca la posición del sensor en y
3
> Datos acerca del robot:
Introduzca la posición del robot en x
1
Introduzca la posición del robot en y
2
Introduzca el ángulo de inclinación -en grados- del robot
67
```

Figura 4.1: Ejemplo concreto de diálogo tras haber insertado datos

2. Cálculo del otro punto de la tangente *P2* y transformación de P y éste a sistema de referencia del origen absoluto.

3. Cálculo de *Punto q*.

4. Creación de matriz de la Transformada de Hough e inserción de *Punto q* determinado.

   - Las dimensiones de la matriz dependerán de lo siguiente:
     - Número de filas: el intervalo será [-160,160].
     - Número de columnas: el valor máximo dependerá de cierta proporción con respecto a la distancia del *Punto q* hallado.
   - La discretización, por su parte, se basará en las siguientes pautas:
     - Para columnas: 3 centésimas de unidad por cada celda.
     - Para filas: 3 unidades por cada celda.

5. Impresión de los distintos elementos en pantalla, como se muestra en la figura 4.2:

   - Sistema de referencia del origen absoluto.
   - Ejes del robot.
   - Ejes del sensor. Su inclinación se basará en que el punto P coincide en dirección con el eje de abcisas del mismo.
   - Punto *P*.
   - Punto *P2*.
   - Recta que forman los dos puntos anteriores.
   - *Punto q* y línea discontinua que lo une con origen absoluto.

6. Impresión de la matriz de la Transformada de Hough, con la posición de *Punto q* marcada con una cruz azul y el resto de posiciones con círculos amarillos.

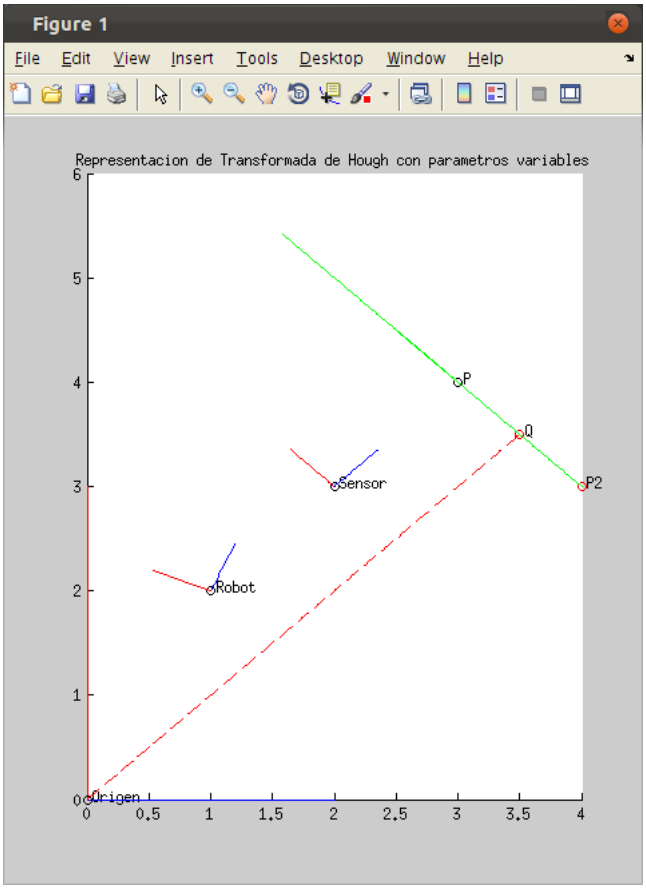


Figura 4.2: Captura de ventana generada por prototipo inicial para el caso concreto

Finalmente, se hace conveniente citar las funciones principales a las que se recurre en el código principal:

- *calcula_angulo*: permite cálcular el ángulo con respecto al eje horizontal entre 2 puntos dados.
- *genera_rectas*: en base a un punto de partida, un tamaño de eje y un ángulo determinado, dibuja las rectas resultantes y/o devuelve el extremo de éstas, dependiendo de lo que se solicite en sus parámetros de entrada. Más concretamente, hace uso de la siguiente ecuación de la recta, donde m es la pendiente y b el punto de corte de la recta con el eje y:

$$y = m \cdot x + b \tag{4.1}$$

4.1.2. Versión avanzada

Tal y como se comentaba anteriormente, se trata de la versión más avanzada de cuantas se han hecho a partir del prototipo inicial.

En este caso, los objetivos que han primado en su desarrollo son, incluyendo los propios del modelo base:

1. Incluir obstáculos y movimiento de robot.
2. Detección dinámica de los obstáculos a partir de los diferentes sensores en las distintas posiciones del robot.
3. Extracción de rectas con mayor número de votos sobre la matriz de la Transformada de Hough rellenada.

Así pues, y con respecto a las etapas del capítulo de *Análisis del problema*, se añadiría la Etapa 5: Extracción de líneas y puntos en matrices de la Transformada de Hough a las ya consideradas.

En cuanto a las restricciones, comparte las propias del prototipo inicial, pero exceptuando la de atender a un único punto para un único sensor. Para este modelo se partirá de 8 sensores posicionados con respecto al centro del robot, y para cada cual se evaluará cierto número de puntos discretizados cuando se detecte algún obstáculo. El robot viene representado como se muestra en la figura 4.3, y en ésta es posible percatarse de ciertos detalles:

- Triángulo: El vértice que une las rectas negras laterales y la azul indica la orientación del robot.
- Rectas de sensores: señalan la dirección de la bisectriz de cada arco de barrido.

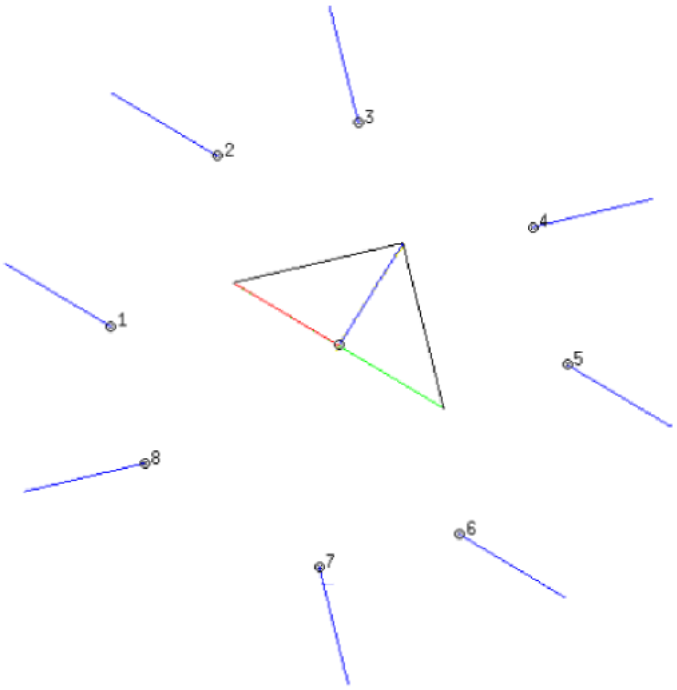


Figura 4.3: Representación de robot y sensores con cierto grado de inclinación

El algoritmo de este prototipo es el siguiente:

1. Petición de datos iniciales al usuario, como se muestra para casos diferentes en la figuras 4.4 y 4.5:

   1.1. Modo automático o manual: el primero ahorra al usuario la necesidad de pulsar una tecla en cada paso del proceso, algo que, sin embargo, se presenta útil para depuración.

   1.2. Número de obstáculos a dibujar: este parámetro da a entender a la aplicación cuándo habrá terminado el usuario de insertar obstáculos y se proponga establecer el recorrido del robot.

   1.3. Parámetros por defecto: se permite al usuario cambiar los parámetros genéricos de:

   - Distancia máxima de detección para los sensores.
   - Ángulo de apertura del arco de barrido.
   - Número de puntos a discretizar y evaluar para cada arco de barrido.

2. Impresión en pantalla de origen absoluto y robot posicionado sobre éste (4.6). Además, se calcula posición de cada sensor con respecto a dicho origen.

3. Impresión en pantalla de obstáculos a medida que el usuario vaya insertándolos: para generar un obstáculo, basta con pulsar la tecla *Enter* tras haber situado los diferentes nodos del mismo. En la figura 4.7 se muestra el ejemplo para 2 obstáculos, el superior con 2 nodos y el inferior, con 3.

4. Impresión en pantalla de interpolación de nodos para recorrido del robot, como en el ejemplo de la figura 4.8: antes el usuario marcará en pantalla los diferentes puntos por los que pasará el robot, y finalizará pulsando nuevamente la tecla *Enter*. La orientación del robot en cada una de las posiciones dependerá del trazado de la propia curva de interpolación.

5. Para cada una de las posiciones del robot:

   5.1. Se imprimen en pantalla los elementos mencionados y se añade el robot y los sensores inclinados convenientemente según la inclinación que corresponda.

   5.2. Para cada uno de los sensores, se comprueba si existe alguna colisión de su arco de barrido con alguno de los obstáculos, teniendo en cuenta la distancia máxima de detección. Si es así, se dibuja el arco de barrido con los puntos discretizados a cierta distancia con respecto al sensor (figura 4.9; ésta, a su vez, será aquella con el mínimo valor de cuantas se hayan evaluado con respecto a los diferentes obstáculos.

   5.3. Para cada punto de cada arco de barrido, se calcula la recta tangente con respecto al sensor pertinente, como aparece en la figura 4.10. Una vez hallada, se procede a obtener el *Punto q* y se incrementa el valor de su celda, asociada a sus parámetros polares con respecto al origen absoluto, de donde partió el robot inicialmente.

6. Una vez acumulada toda la información, pasa a calcularse el conjunto de rectas con más probabilidad de existir. Una vez determinadas, se muestran en pantalla sobre los elementos ya presentes, tal y como en la captura 4.11.

7. Se imprime la matriz de la Transformada de Hough en escala de grises, donde a mayor número de votos, mayor tendencia a color blanco, como en el ejemplo de la figura 4.12.

```
<<<PROTOTIPO SIMULACIÓN v.5>>>
Pulse 1 si desea ejecutar en modo automático, o 0 si no:
0
Indique el número de obstáculos que desea dibujar:
2
Introduzca 1 si desea realizar algún cambio en los valores de los parámetros predeterminados, o 0 si no.
Los valores son: distancia máxima: 3 // ángulo de apertura: 30 // nº puntos a discretizar: 35
...
0
```

Figura 4.4: Ejemplo de diálogo con el usuario tras la inserción de datos

```
<<<PROTOTIPO SIMULACIÓN v.5>>>
Pulse 1 si desea ejecutar en modo automático, o 0 si no:
0
Indique el número de obstáculos que desea dibujar:
2
Introduzca 1 si desea realizar algún cambio en los valores de los parámetros predeterminados, o 0 si no.
Los valores son: distancia máxima: 3 // ángulo de apertura: 30 // nº puntos a discretizar: 35
...
1
La distancia máxima de detección por defecto es de 3 unidades
Introduzca 1 si desea cambiarla, y 0 si no:
1
Introduzca el nuevo valor de distancia máxima común:
4
El arco de barrido tiene un ángulo por defecto de 30 grados
Introduzca 1 si desea cambiarlo, y 0 si no:
0
El número de puntos que se discretizará en su caso es de 35, por defecto
Introduzca 1 si desea cambiarlo, y 0 si no:
1
Introduzca el nuevo número de puntos a discretizar:
32
```

Figura 4.5: Otro ejemplo de diálogo con el usuario tras la inserción de datos

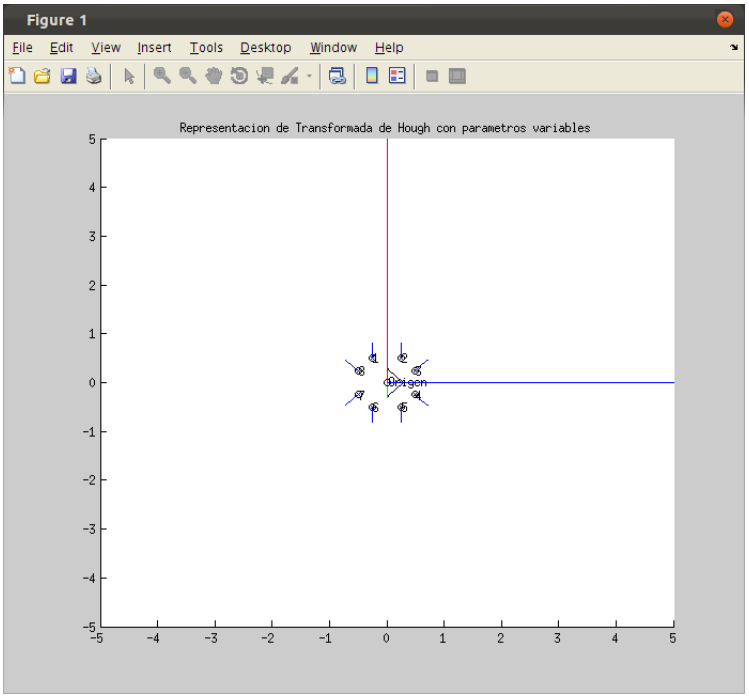


Figura 4.6: Robot posicionado inicialmente en origen absoluto

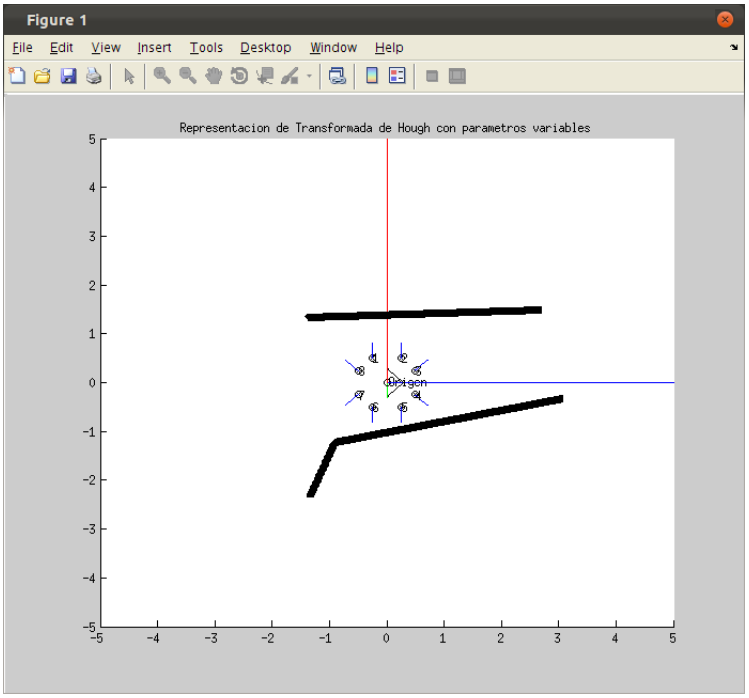


Figura 4.7: Ejemplo de 2 obstáculos insertados por el usuario

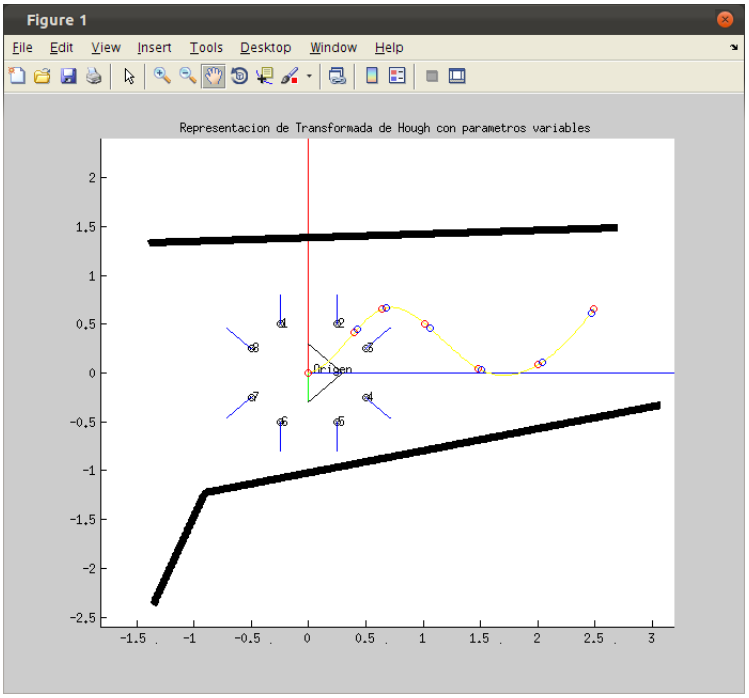


Figura 4.8: Ejemplo de interpolación de nodos de recorrido

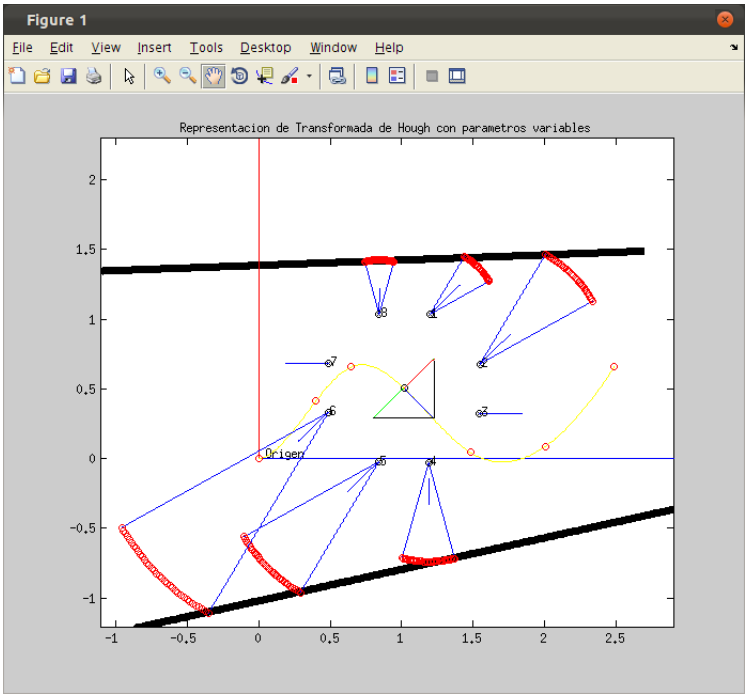


Figura 4.9: Captura de cierta posición del robot con ciertos sensores que han detectado obstáculos

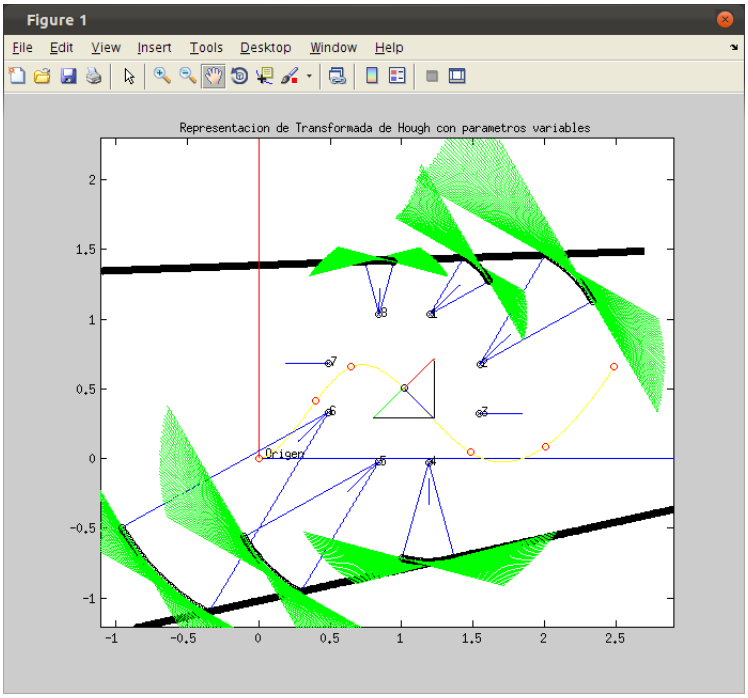


Figura 4.10: Captura de cálculo de tangentes para situación de figura 4.9

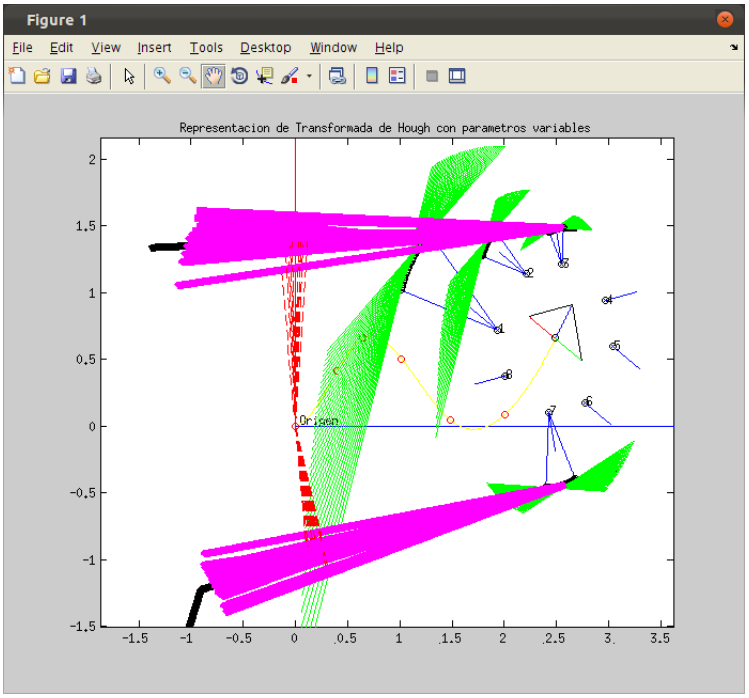


Figura 4.11: Representación de rectas probables tras completar recorrido y evaluación de matriz

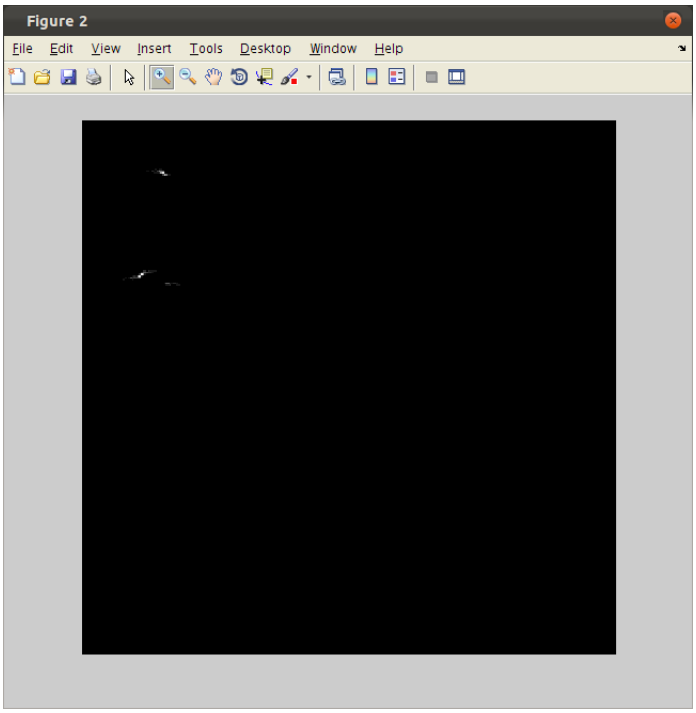


Figura 4.12: Matriz de la Transformada de Hough resultante en escala de grises

A continuación se citan los detalles más relevantes del procedimiento, considerando incluidas las funciones principales del prototipo inicial:

- La forma de almacenar los obstáculos es mediante 2 matrices, una para el eje x y otra para el eje y. Éstas contendrán tantas columnas como rectas tengan los obstáculos, y tantas filas como puntos contengan éstas.
- La interpolación de los nodos de recorrido se efectúa mediante el método de *Splines cúbicos*, haciendo uso entonces de las funciones nativas *spline()* y *ppval()*.
- Para calcular la inclinación del robot en cada posición, se ha de obtener el punto inmediatamente posterior al nodo correspondiente, excepto en el caso de la posición inicial, donde el robot no tiene inclinación alguna, y cuando se encuentra en el nodo final, caso en el que cabe calcular el punto inmediatamente anterior. Una vez calculado dicho punto, el ángulo de inclinación será aquel que formen el susodicho y el nodo en cuestión.
- La función *núcleo()* genera, para cada sensor y en base a la distancia máxima de percepción, todos los puntos de cada uno de los arcos de barrido, con tal de compararlos con cada uno de los puntos de los obstáculos. De esta forma, se determinará la distancia que devolvería el cada sensor en un caso real simplemente quedándonos con el valor mínimo de distancia entre las coincidencias. Para ello, cuando coincida algun punto de los generados con alguno de los obstáculos, se acumulará la distancia de dicho punto con respecto a la posición del sensor.
- Una vez ejecutada la función *núcleo*, se pueden determinar los puntos discretizados en base a la distancia a la que cada sensor ha detectado el obstáculo. La forma de proceder a partir de ahi coincide con la del prototipo inicial a la hora de calcular los parámetros de cada *Punto q*. Cabe señalar que, en este caso, las transformaciones de sistemas de referencia se encapsulan en la función denominada *lleva_a_origen()*.
- Cuando la matriz de la Transformada de Hough sea completada, las rectas probables serán aquellas vinculadas a aquellos *Puntos Q* cuyas celdas tengan más de 1 voto. En la figura 4.11 son las que aparecen en color fucsia, y como se puede apreciar, coinciden considerablemente con los obstáculos establecidos a priori.

4.2. Prototipo final en CoolBOT

Una vez concluido el prototipo en Matlab, se procede a crear el *bundle* de CoolBOT, cuyo nombre es **coolbot-sonar-bundle**. Como se citó en la introducción a CoolBOT del capítulo de *Metodología y Plan de trabajo*(2.2.3), un *bundle* contiene todos los elementos necesarios para el sistema. En el caso de este proyecto, atenderá a lo siguiente:

- Dos componentes.
- Dos vistas.
- Paquetes de puertos.
- Integración.

A partir de este punto y hasta el final del capítulo, se realizará un análisis exhaustivo de los distintos componentes software, paquetes e integración sustentados por el *bundle*.

4.2.1. Componentes

Los componentes conforman la gran parte de la funcionalidad otorgada por el sistema. En este caso concreto, se crearon dos con objetivos bien delimitados, que serán especificados en lo sucesivo: **SonarHT** y **LaserIEPF**. Adicionalmente se incluirá una sección para explicar la clase **Merge**, utilizada por ambos componentes.

En CoolBOT, los componentes se ejecutan a modo de máquina de flujo de datos [Arvind,1981], y por esta razón se modelan como autómatas de puertos, para luego ser implementados como clases. Asi pues, el diagrama de estados de cualquier componente en CoolBOT sigue el modelo de autómata de la figura 4.13.

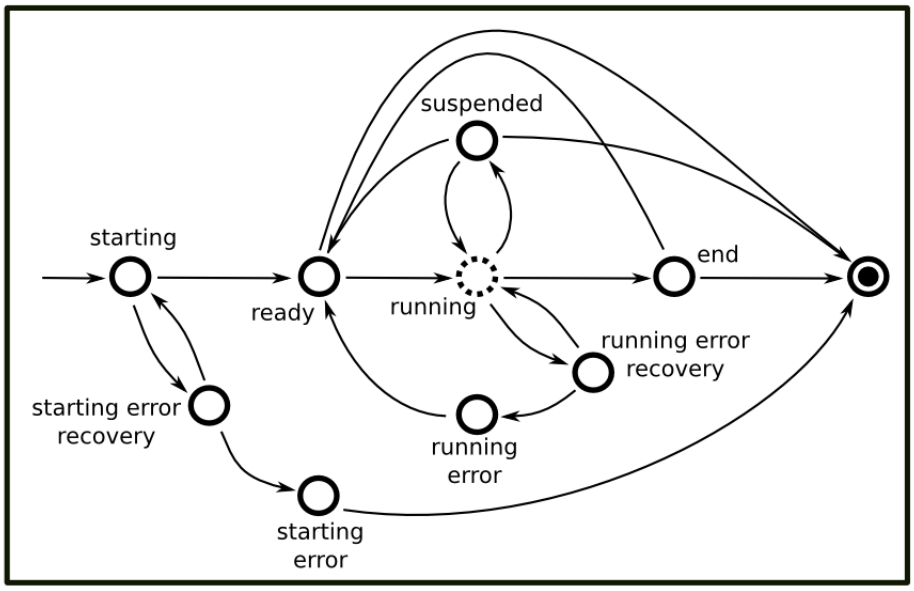


Figura 4.13: Autómata por defecto para componentes en CoolBOT (extraído de [Domínguez-Brito, 2012a])

Cada nuevo estado que se incorpore a un componente, que como se verá se añaden dentro del estado *running*, dispone de las siguientes funciones por defecto:

- *Entry*: establece qué se hace cuando se accede al estado al que pertenezca.
- *Exit*: establece qué se hace cuando se sale del estado al que pertenezca.
- *Funciones propias de transiciones a sí mismo*: habrá una función por cada una de las transiciones que no impliquen un cambio de estado.

SonarHT

Si hubiese que determinar un núcleo de procesamiento entre los componentes software, sin duda sería **SonarHT**. En una visión muy general y aludiendo a contenido ya explicado anteriormente, este componente se encarga de todas las etapas de procesamiento que conciernen a los sensores ultrasónicos, desde la etapa explicada en la sección 3.2.1 hasta la propia de la sección 3.2.7. No obstante, la etapa explicada en la sección 3.2.6 es la única que exige una llamada a cierta clase independiente, donde se encuentra integrada, y que se explicará en la sección 4.2.1.

Desde un punto de vista externo, de cara al sistema de entrada y salida, la interfaz de este componente es la que se muestra en la figura 4.14.

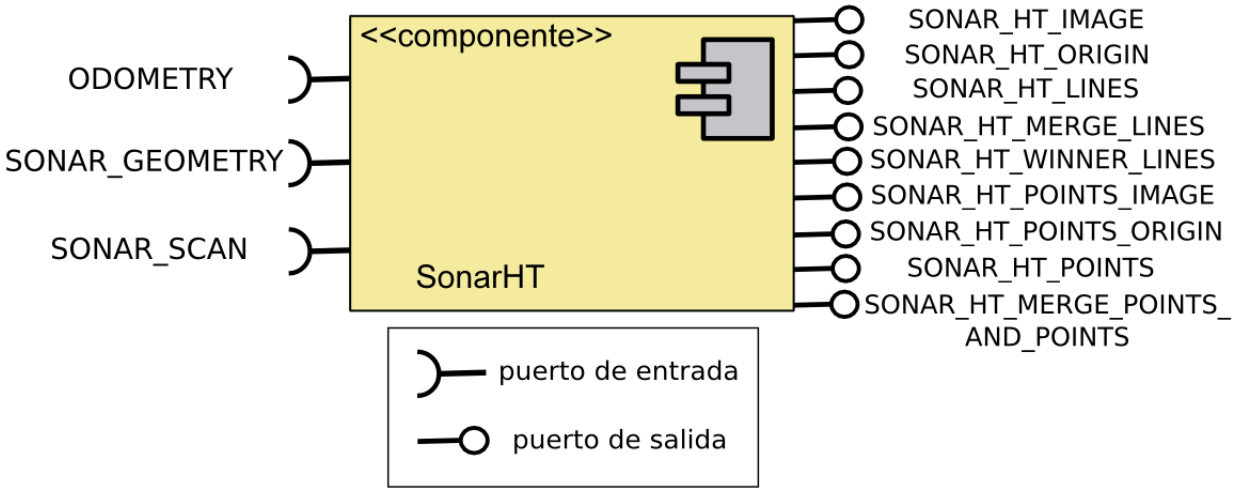


Figura 4.14: Interfaz de sistema de entrada/salida en SonarHT

Los diferentes puertos permiten la entrada y/o salida de ciertos tipos de paquetes que encapsulan determinadas clases de datos. Dichos tipos son los que siguen a continuación:

- **Propios de PlayerRobotPackets** (externo a este *bundle*)
  - *OdometryPacket*: encapsula conjuntos de tres datos: dos de tipo *Frame2D* (nativo de CoolBOT) y uno de tipo *double*. Aparte añade ciertas funciones para el manejo de estos datos. Este tipo de paquetes está destinado a trasladar valores de posición, velocidad y odometría o distancia recorrida del robot. Los paquetes del puerto **ODOMETRY** son de este tipo.
  - *SonarGeometryPacket*: encapsula datos de tipo *Frame2D*, y en concreto se utiliza para indicar la posición de cada uno de los sensores ultrasónicos con respecto al robot. Los paquetes del puerto **SONAR_GEOMETRY** son de este tipo.
  - *SonarPacket*: encapsula datos de tipo *double*, y particularmente su utilidad es contener las diferentes distancias detectadas en los arcos de barrido por los sensores ultrasónicos. Los paquetes del puerto **SONAR_SCAN** son de este tipo.

- **Propios de SonarHTPackets**

  - *HTImagePacket*: encapsula los datos necesarios para trasladar imágenes en formato RGB de matrices. Los paquetes de los puertos **SONAR_HT_IMAGE** y **SONAR_HT_POINTS_IMAGE** son de este tipo.
  - *HTFramePacket*: encapsula datos cuyo tipo es *Frame2D*. Los paquetes de los puertos **SONAR_HT_ORIGIN** y **SONAR_HT_POINTS_ORIGIN** son de este tipo.
  - *HTLinesPacket*: encapsula líneas, entendiendo como líneas estructuras con dos campos de tipo *Coordinates2D* (nativo de CoolBOT) y un campo de tipo *unsigned long*. Aparte, aporta ciertas funciones útiles para el manejo de las líneas. Los paquetes de los puertos **SONAR_HT_LINES**, **SONAR_HT_MERGE_LINES** y **SONAR_HT_WINNER_LINES** son de este tipo.
  - *HTPointsPacket*: igual que el tipo anterior, pero en este caso los puntos son estructuras con un único campo de tipo *Coordinates2D* y otro campo de tipo *unsigned long*. El puerto **SONAR_HT_POINTS** atiende a paquetes de este tipo.
  - *HTSuperPointsAndPointsPacket*: encapsula estructuras que contienen un campo cuyo tipo es la estructura de los puntos del tipo de paquetes anterior, y otro campo que consiste en una lista de estas estructuras. Los paquetes del puerto **SONAR_HT_MERGE_POINTS_AND_POINTS** son de este tipo.

A continuación se añade el contenido del fichero descriptivo de la clase SonarHT, cuyo nombre en los directorios del bundle es *sonar-ht.coolbot-component*, dentro de la carpeta *sonar-ht*, donde también se encuentran los archivos fuentes de la implementación de esta clase. En este fichero se definen todas las constantes propias de la clase del componente, los puertos de entrada/salida y los estados del autómata.

```
component SonarHT
{
  header
  {
    author "Javier Hernández Trujillo <phybos86@gmail.com>";
    description "Sonar Hough Transform component";
    institution "Modelado de sensores sónar en robótica móvil -PFC-";
    version "0.1"
  };

  constants
  {
    SONAR_HALF_WIDE_ANGLE=15; // degrees
    SONAR_TANGENT_POINTS=75; //number of points to evaluate
    RO_CENTIMETERS_PER_CELL=3;   //to discretize ro
    THETA_DEGREES_PER_CELL=3;   //to discretize theta
    RO_MAX=8; //ro maximum value (m)
```

```
  SONAR_RETURN_THRESHOLD=2000;  //on millimeters
  LINES_VOTES_THRESHOLD=275;  //number of votes that would have a cell to
                                be sent as probably line.
  POINTS_VOTES_THRESHOLD=350;
  LINES_TIME_TO_DECREMENT=1000;  // which has to passed without make vote to
                                   decrement (on milliseconds)
  POINTS_TIME_TO_DECREMENT=1000;
  LINES_PERSISTENCE_DECREMENT=7; // number of votes that will be decrement
  POINTS_PERSISTENCE_DECREMENT=10;
  CROSS_RADIUS=2; // grid cells
  PLH_MINIMUM_DISTANCE=5; //minimum distance to consider a conflict in
                          Points Lines Hypothesis case (on centimeters)
  PLH_LINE_X_DISTRIBUTION=20; //minimum value of typical deviation on X
                                to consider a line
  PLH_LINE_Y_DISTRIBUTION=10; //maximum value of typical deviation on Y
                                to consider a line

  private DEFAULT_PERIOD=200; // milliseconds
  private DEFAULT_PERSISTENCE_PERIOD=1000; // milliseconds
  private CIRCULAR_FIFO_LENGTH=100; // in odometry periods (about 100-200
                                      millisecond each)
};

// input ports

input port ODOMETRY type last port packet PlayerRobot::OdometryPacket;
input port SONAR_GEOMETRY type poster port packet PlayerRobot::SonarGeometryPacket;
input port SONAR_SCAN type last port packet PlayerRobot::SonarPacket;

// output ports

output port SONAR_HT_IMAGE type poster port packet SonarHTPackets::HTImagePacket;
output port SONAR_HT_ORIGIN type poster port packet SonarHTPackets::HTFramePacket;
output port SONAR_HT_LINES type generic port packet SonarHTPackets::HTLinesPacket;
output port SONAR_HT_MERGE_LINES type generic port packet
                            SonarHTPackets::HTLinesPacket;
output port SONAR_HT_WINNER_LINES type generic port packet
                             SonarHTPackets::HTLinesPacket;
output port SONAR_HT_POINTS_IMAGE type poster port packet
                            SonarHTPackets::HTImagePacket;
output port SONAR_HT_POINTS type generic port packet
                       SonarHTPackets::HTPointsPacket;
output port SONAR_HT_POINTS_ORIGIN type poster port packet
                             SonarHTPackets::HTFramePacket;
output port SONAR_HT_MERGE_POINTS_AND_POINTS type generic port
           packet SonarHTPackets::HTSuperPointsAndPointsPacket;
```

```
  /*
   * State's definition.
   */

  controllable variables
  {
    NEW_PERIOD: port packet PacketLong;
  };

  observable variables
  {
    PERIOD: port packet PacketLong;
  };

  entry state waiting
  {
    transition on ODOMETRY, SONAR_GEOMETRY;
  };

  state theMain
  {
    transition on
      TIMER, PERSISTENCE_TIMER,
      ODOMETRY, SONAR_GEOMETRY,
      SONAR_SCAN, NEW_PERIOD;
  };

};
```

En cuanto al autómata, en la parte inferior de la figura 4.15 se muestra como el estado *running* se desglosa en dos nuevos estados concretos para el caso de este componente en particular.

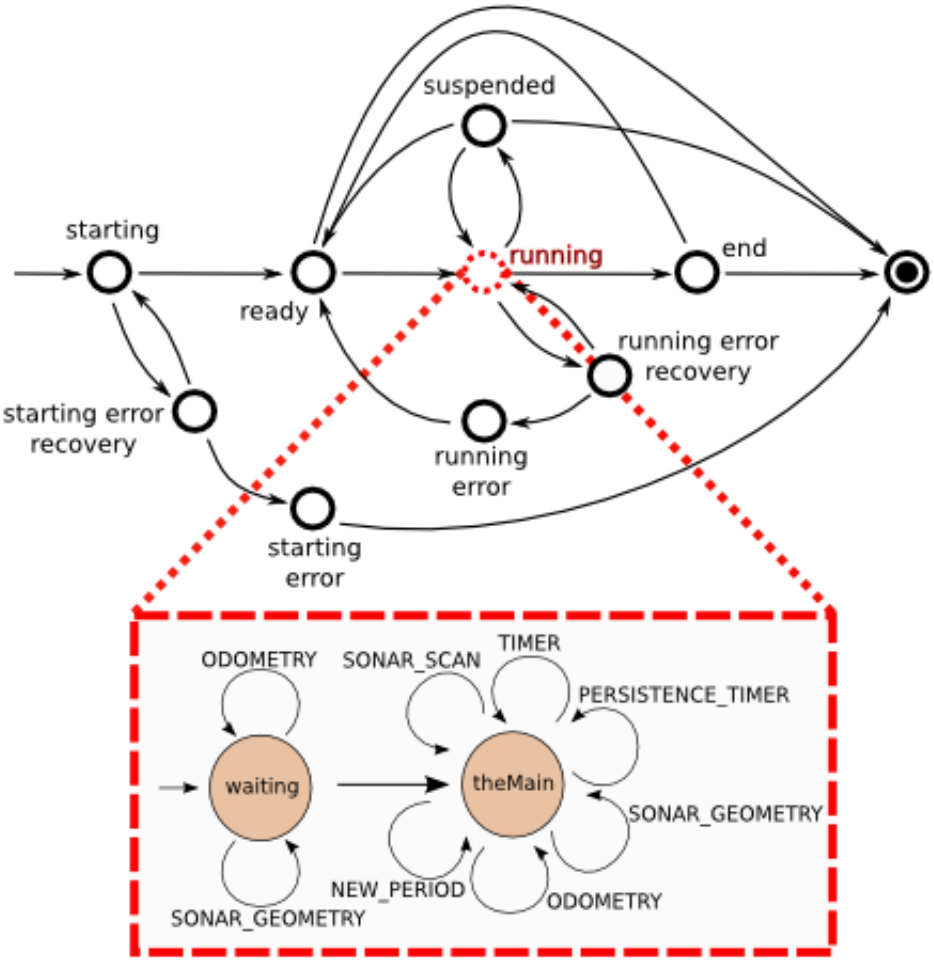


Figura 4.15: Diagrama de estados genérico y para caso concreto de SonarHT

El estado *Waiting* es el inicial cuando se accede al componente al conectarlo al sistema, y su objetivo es almacenar ciertos datos mientras no se den las condiciones necesarias para pasar al estado principal *theMain*. Las funciones asociadas a este estado son:

- *_waitingEntry_*: se encarga de inicializar a *falso* dos variables de control, cada una vinculada a una función de transición distinta.
- *_waitingExit_*: para esta implementación no tiene utilidad alguna.
- *_waitingODOMETRY_*: esta función es llamada cada vez que llega al componente un paquete de *odometría*. Asimismo, almacena el valor de odometría del robot, esto es, su posición con respecto al origen del que partió. Los dos orígenes de la Transformada de Hough correspondientes a las matrices de líneas y puntos se inicializarán con este valor (ver capítulo de *Análisis del problema*).
- *_waitingSONAR_GEOMETRY_*: la función es llamada cuando llega un paquete de *sonar_geometry* al componente. En este caso almacena la posición de cada uno de los sensores ultrasónicos con respecto al robot, o lo que es lo mismo, el contenido de tal paquete.

Las condiciones para pasar al estado *theMain*, referidas en el párrafo anterior, están relacionadas con las dos funciones anteriores: cada vez que se entra en alguna de éstas, se activa su correspondiente variable de control y se comprueba si la propia de la otra función también está activada; es en esta circunstancia cuando el autómata pasa al estado principal *theMain*.

El estado *theMain* se corresponde con las siguientes funciones:

- *_theMainEntry_*: al entrar en el estado *theMain*, se activan los dos contadores de tiempo que servirán de período de ejecución para las funciones *_theMainTimer_* y *_theMain_PERSISTENCE_TIMER_*, respectivamente. El valor de estos dos contadores dependen de ciertas variables, cuya inicialización cabe comentar:

  ```
  _period_=(period<=0)? _DEFAULT_PERIOD_: period;
  _persistence_period_=(persistence_period<=0)?
                       _DEFAULT_PERSISTENCE_PERIOD_: persistence_period;
  ```

  Esta inicialización sugiere que existen dos formas de darles valor a dichas variables: mediante paso de parámetros a la hora de instanciar el componente en la integración (period y persistence_period), o bien a través de constantes inicializadas en el esqueleto del propio componente (_DEFAULT_PERIOD_ y _DEFAULT_PERSISTENCE_PERIOD_). Dado el orden de la inicialización condicional, por defecto se atiende al valor pasado por parámetro en el constructor.

- *_theMainExit_*: ambos contadores de tiempo -activados en *_theMainEntry_*-, se detienen.

- *_theMainTimer_*: en el capítulo de *Análisis del problema* se indicaba que las etapas 5 (sección 3.2.5), 6 (sección 3.2.6) y 7 (sección 3.2.7) de las etapas de procesamiento para los sónars se ejecutaban de forma secuencial cada cierto período de tiempo, y concretamente se incorporan dentro de esta función, que pasará a ejecutarse según el contador de tiempo correspondiente.

  Para esta implementación se ha optado por un orden diferente en cuanto a la ejecución de las diferentes etapas, con respecto a lo expuesto en dicho capítulo. Básicamente se cambia el carácter secuencial por la ejecución de las etapas 5 y 6 para líneas, a continuación para puntos y, finalmente, la etapa 7. El motivo por el que se detalla de forma diferente en el capítulo de *Análisis del problema* es meramente didáctico, puesto que la explicación se hace más inteligible de ese modo.

  Por otra parte, existe un detalle que no procedía mencionar en el análisis de las etapas por tratarse de algo propio de la implementación: las matrices de la Transformada de Hough a modo de imágenes. No sólo se inserta y extrae la información que proceda en éstas, sino que una de las vistas implementadas para este *Bundle* hace uso de las imágenes de sendas matrices, con tantos píxeles como celdas contengan las originales para una asociación directa. En la etapa 5, cuando se extraen las líneas y puntos probables de las matrices correspondientes, se establece el color del píxel en *RGB*, ligado a cada celda, según el número de votos de ésta y siguiendo este criterio:

  - Celda correspondiente a línea probable: si la celda tiene un número suficiente de votos para la matriz de líneas, el píxel se pintará en rojo.
  - Celda correspondiente a punto probable: si ocurre el mismo caso para la matriz de puntos, el píxel se pintará de color azul.

- Celda con número insuficiente de votos: para cualquiera de las dos matrices, si la celda en cuestión no dispone del número suficiente de votos para considerar la línea o punto asociado como probable, su color tendrá un valor proporcional en escala de grises, tendiendo al color blanco entre mayor sea el número de votos.

Del mismo modo, tras la etapa 6 se calculan las celdas de las líneas y puntos fusionados, y esta vez se siguen las siguientes reglas:

- Celda correspondiente a línea resultado de la fusión: se dibuja una cruz de color **amarillo** de cierto tamaño con centro en el píxel de la celda de la matriz de líneas.
- Celda correspondiente a punto resultado de la fusión: igual que en el caso de las líneas, pero para la matriz de puntos y con una cruz de color rojo.

En esta función se efectúa todo el proceso de salida del componente a través de los puertos antes mencionados. De esta forma, cada uno de dichos puertos está orientado a cierta utilidad, sugerida por su propio nombre, y que se detallan seguidamente a partir del orden que mantienen en el propio código:

- **SONAR_HT_LINES**: por este puerto se envían las líneas probables extraídas de su matriz.
- **SONAR_HT_MERGE_LINES**: por donde se envían las líneas obtenidas tras aplicar la fusión.
- **SONAR_HT_IMAGE**: por donde se envía la imagen en RGB de la matriz de líneas.
- **SONAR_HT_ORIGIN**: por donde se envía el origen de la Transformada de Hough para las líneas.
- **SONAR_HT_POINTS**: por donde se envía los puntos probables extraídos de su matriz.
- **SONAR_HT_POINTS_ORIGIN**: por donde se envía el origen de la Transformada de Hough para los puntos.
- **SONAR_HT_POINTS_IMAGE**: por donde se envía la imagen en RGB de la matriz de puntos.
- **SONAR_HT_MERGE_POINTS_AND_POINTS**: por donde se envían los puntos obtenidos tras aplicar la fusión con cada subgrupo de puntos que lo generaron.
- **SONAR_HT_WINNER_LINES**: por donde se envían las líneas que hayan superado la etapa 7.

- *_theMainPERSISTENCE_TIMER_*: la ejecución de esta función, que se encarga de la etapa 4 (sección 3.2.4), dependerá del otro contador de tiempo activado en la función *_theMainEntry_*.

- *_theMainODOMETRY_*: esta función se ejecutará cuando llegue un paquete de odometría estando en el estado *Main*. Su objetivo es ir almacenando los paquetes de odometría que vayan llegando, algo indispensable para la función de interpolación de odometría que, dado su uso en *_theMainSONAR_SCAN_*, se explicará en su apartado.

- *_theMainSONAR_GEOMETRY_*: en esta implementación no tiene utilidad alguna, más que volver al mismo estado cuando llegue un paquete de SonarGeometry.

- *_theMainSONAR_SCAN_*: función que puede considerarse el núcleo del componente Sonar-HT, ya que realiza las etapas 1 (sección 3.2.1), 2 (sección 3.2.2) y 3 (sección 3.2.3). En el caso de esta implementación, el proceso se lleva a cabo secuencialmente para cada punto discretizado de cada sensor ultrasónico.

  Como se mencionó anteriomente, esta función hace uso de la interpolación de odometría. La interpolación permite hallar el valor odométrico más cercano (posición, velocidad y distancia recorrida por el robot) al instante en el que se evalúa la distancia devuelta por cada sónar, evitando así problemas de correspondencia y optimizando los cálculos. El fundamento del método es comparar cada una de las marcas de tiempo de los paquetes de odometría -almacenados por *_theMainODOMETRY_*- con el instante en el que se captó la distancia devuelta por el arco de barrido del sensor correspondiente.

- *_theMainNEW_PERIOD_*: esta función de transición esta preparada para cambiar el período de ejecución de *_theMainTimer_* en tiempo de ejecución, pero en el caso de esta implementación no se hace nada.

LaserIEPF

Este segundo componente se encarga de las 3 etapas vinculadas al láser, explicadas en la sección 3.3: Etapa 1: Detección de puntos del sensor, Etapa 2: División de conjuntos de puntos y generación de líneas y Etapa 3: Fusión de líneas. Su interfaz de entrada/salida se presenta en la figura 4.16.

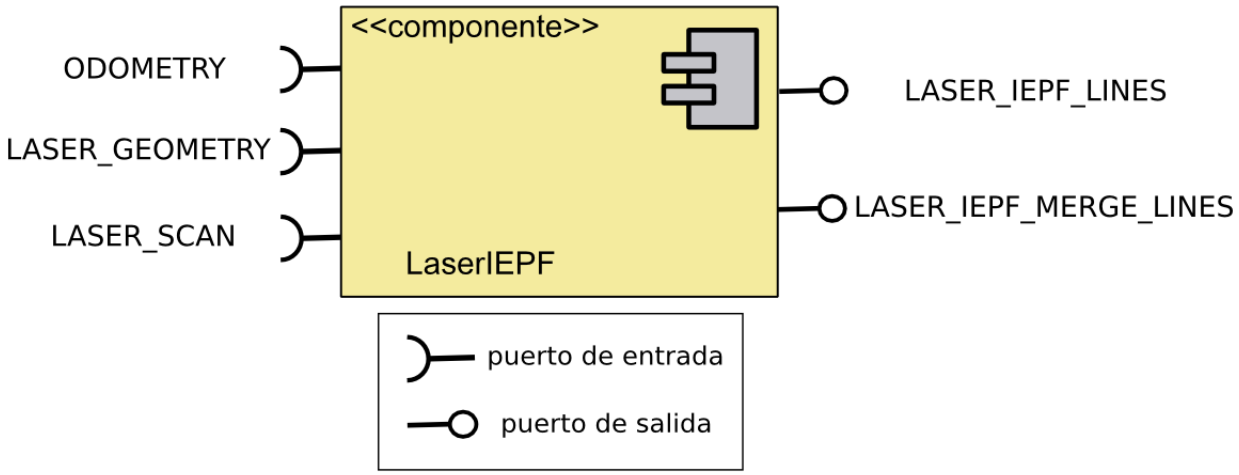


Figura 4.16: Interfaz de sistema de entrada/salida en LaserIEPF

Como en el caso de Sonar-HT, se procederá a detallar los diferentes tipos de paquetes que manejan los puertos de esta interfaz:

- *PacketFrame3D*: encapsula datos de tipo *Frame3D* (nativo de CoolBOT), y en concreto se utiliza para indicar la posición del transductor láser con respecto al robot. Los paquetes del puerto **LASER_GEOMETRY** son de este tipo.

- **Propios de PlayerRobotPackets** (externo a este *bundle*)

  - *OdometryPacket*: encapsula conjuntos de tres datos: 2 de tipo *Frame2D* (nativo de CoolBOT) y uno de tipo *double*. Aparte añade ciertas funciones para el manejo de estos datos. Este tipo de paquetes está destinado a trasladar valores de posición, velocidad y odometría o distancia recorrida del robot. Los paquetes del puerto **ODOMETRY** son de este tipo.
  - *LaserPacket*: encapsula conjuntos de tres datos: uno de tipo *Coordinates2D* (nativo de CoolBOT), otro de tipo *entero* y otro de tipo *booleano*. Además añade ciertas funciones para el manejo de estos datos. Este tipo está concebido para trasladar las distancias devueltas por el sensor láser, así como valores de intensidad e invalidez asociados. Los paquetes del puerto **LASER_SCAN** son de este tipo.

- **Propios de SonarHTPackets**

  - *HTLinesPacket*: encapsula líneas, entendiendo como líneas estructuras con dos campos de tipo *Coordinates2D* (nativo de CoolBOT) y un campo de tipo *unsigned long*. Aparte, aporta ciertas funciones útiles para el manejo de las líneas. Los paquetes de los puertos **LASER_IEPF_LINES** y **LASER_IEPF_MERGE_LINES** son de este tipo.

En el listado anterior se puede apreciar como hay dos tipos de paquetes que también estaban vinculados a ciertos puertos en la interfaz del componente SonarHT: OdometryPacket y HTLinesPacket. La reutilización de este último tipo demuestra la importancia de concebir tipos de paquetes lo más versátiles posibles.

El fichero descriptivo de la clase LaserIEPF se halla en el fichero *laser-iepf.coolbot-component*, dentro del directorio *laser-iepf*, donde están ubicados los archivos fuente de esta clase. Su contenido se presenta seguidamente:

```
component LaserIEPF
{

  header
  {
    author "Javier Hernández Trujillo <phybos86@gmail.com>";
    description "Iterative End-Point Fit (IEPF) implementation using laser component"
    institution "Modelado de sensores sónar en robótica móvil -PFC-";
    version "0.1"
  };

  constants
```

```
  {
    LINE_POINTS_THRESHOLD=10; //minimum scan points per line
//     MERGE_RO_THRESHOLD=3;   //on centimeters
//     MERGE_THETA_THRESHOLD=5;  //on degrees

    private DEFAULT_PERIOD=200; // milliseconds
    private CIRCULAR_FIFO_LENGTH=100; // in odometry periods
                              (about 100-200 millisecond each)

  };

  //input ports
  input port ODOMETRY type last port packet PlayerRobot::OdometryPacket;

  input port LASER_GEOMETRY type poster port packet PacketFrame3D;

  input port LASER_SCAN type last port packet PlayerRobot::LaserPacket;

  //output ports

  output port LASER_IEPF_LINES type poster port packet
                          SonarHTPackets::HTLinesPacket;

  output port LASER_IEPF_MERGE_LINES type poster port packet
                               SonarHTPackets::HTLinesPacket;

  controllable variables
  {
    NEW_PERIOD: port packet PacketLong;
  };

  observable variables
  {
    PERIOD: port packet PacketLong;
  };

  entry state waiting
      {
          transition on ODOMETRY, LASER_GEOMETRY;
      };

  state theMain
  {
    transition on
      TIMER, ODOMETRY, LASER_GEOMETRY,
      LASER_SCAN, NEW_PERIOD;
```

```
  };

};
```

En el caso de LaserIEPF, el estado *running* del autómata por defecto también se desglosa en estados homónimos con respecto a los de SonarHT, aunque con transiciones diferentes, como aparece en la figura 4.17.

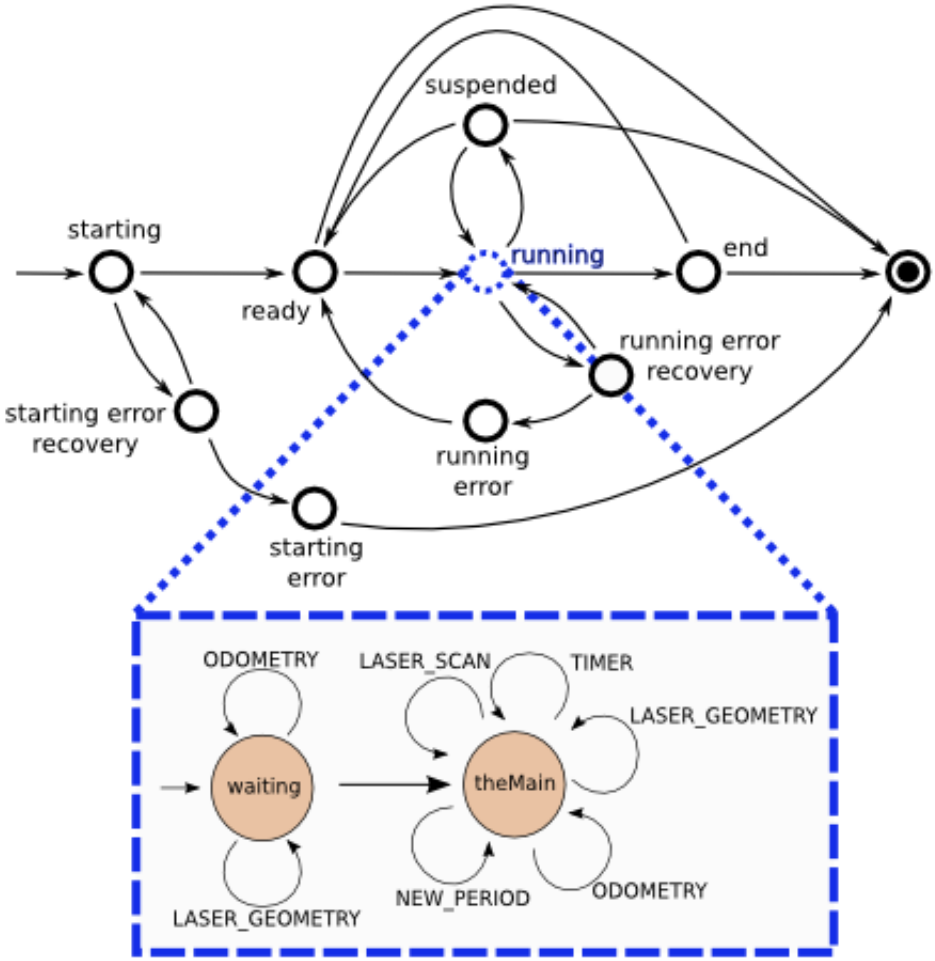


Figura 4.17: Diagrama de estados genérico y para caso concreto de LaserIEPF

El estado *waiting* es el de entrada, y sus funciones son las siguientes:

- *_waitingEntry_*: se encarga de inicializar a *falso* dos variables de control, cada una vinculada a una función de transición distinta.
- *_waitingExit_*: para esta implementación no tiene utilidad alguna.
- *_waitingODOMETRY_*: esta función es llamada cada vez que llega al componente un paquete de *odometría*.

- *_waitingLASER_GEOMETRY_*: la función es llamada cuando llega un paquete de *laser_geometry* al componente. En este caso almacena la posición del sensor láser con respecto al robot, o lo que es lo mismo, el contenido de tal paquete.

La transición al estado principal *theMain* tiene lugar del mismo modo que en SonarHT: cada vez que se ejecuta alguna de las funciones *_waitingODOMETRY_* o *_waitingLASER_GEOMETRY_*, se activa su correspondiente variable de control y se evalúa si la otra está activada; en ese caso, se pasará al estado *theMain*.

Por otro lado, el estado *theMain* dispone de las siguientes funciones referentes a las transiciones:

- *_theMainEntry_*: al entrar en el estado *theMain*, se activa un contador de tiempo que servirá de período de ejecución para la función *_theMainTimer_*. El valor de este contador dependerá de cierta variable, cuya inicialización sigue el mecanismo que fue aplicado en SonarHT.
- *_theMainExit_*: el contador de tiempo, activado en *_theMainEntry_*, se detiene.
- *_theMainTimer_*: esta función se ejecuta cuando se cumple el período de tiempo previsto para el contador inicialmente activado. En cuanto a su labor, aquí se efectúa todo el proceso de salida del componente a través de los puertos antes mencionados. De esta forma, cada uno de dichos puertos está orientado a cierta utilidad, sugerida por su propio nombre, y que se detallan seguidamente a partir del orden que mantienen en el propio código:
  - **LASER_IEPF_LINES**: por este puerto se envían las líneas obtenidas tras aplicar la etapa 2 (sección 3.3.2), esto es, el algoritmo de IEPF o *Iterative End-Point Fit*.
  - **LASER_IEPF_MERGE_LINES**: por este puerto se envían las líneas que resultan de la etapa 3 (sección 3.3.3), o lo que es lo mismo, de la fusión de las líneas obtenidas tras la etapa 2.
- *_theMainODOMETRY_*: esta función se ejecutará cuando llegue un paquete de odometría estando en el estado *Main*. Su objetivo es ir almacenando los paquetes de odometría que vayan llegando para la función de interpolación de odometría, ya citada en el análisis de SonarHT.
- *_theMainLASER_GEOMETRY_*: en esta implementación no tiene utilidad alguna, más que volver al mismo estado cuando llegue un paquete de LaserGeometry.
- *_theMainLASER_SCAN_*: función que puede considerarse el núcleo del componente LaserIEPF, ya que implementa todas las etapas relacionadas con el láser exceptuando la fusión, que únicamente invoca -como en el caso de SonarHT-. Cabe indicar que hace uso de la interpolación de odometría, del mismo modo que el componente anterior.
- *_theMainNEW_PERIOD_*: esta función de transición esta preparada para cambiar el período de ejecución de *_theMainTimer_* en tiempo de ejecución, pero en el caso de esta implementación no se hace nada.

Merge

La clase Merge incorpora las funciones relacionadas con la fusión de líneas y puntos de forma independiente respecto a CoolBOT. Es por ello que ofrece portabilidad para una posible reutilización en otros proyectos que requieran de estas utilidades. En el caso de este *bundle*, únicamente se ha utilizado en los 2 componentes ya estudiados: SonarHT y LaserIEPF. Sus archivos fuente, *merge.h* y *merge.cpp*, están ubicados dentro de la carpeta *sonar-ht-packets*; aunque no tiene una relación directa con este directorio, deben incluirse en algún directorio del bundle con tal de poder compilarse junto al resto de archivos de éste.

En su definición declara varias constantes y diversas funciones para asociarlas a variables a las que se recurre en su código. Seguidamente, añade las estructuras de datos necesarias para las diferentes funciones principales. Por un lado, las estructuras de datos son estas:

- *winnerParam*: concebida para contener pares de parámetros polares junto con el número de votos.
- *cartesianParam*: esta estructura se creó con la misma finalidad que *winnerParam*, pero orientada a coordenadas cartesianas. A diferencia de la anterior estructura, incluye ciertas funciones como un constructor y útiles para los envíos de paquetes vía red, dado que también es aprovechada para el tipo de paquetes *HTSuperPointsAndPointsPacket*.
- *pointsToMerge*: su fin es poder encapsular los puntos resultantes de la fusión junto con el grupo de puntos que lo generaron.

Las funciones principales de esta clase son:

- *polarMerge*: es la encargada de la fusión de líneas.
- *mergeForPoints*: encargada de la fusión de puntos.
- *polarToCartesian*: esta función auxiliar se encarga de convertir una lista de parámetros polares a cartesianos.
- *cartesianToPolar*: realiza el proceso inverso a la función *polarToCartesian*.
- *returnTheMax*: realiza exactamente lo establecido por el algoritmo 13 del capítulo de *Análisis del problema*.
- *returnMidPoint*: implementa el algoritmo 16.

4.2.2. Vistas

Las vistas son aquellos componentes software que implementan interfaces para la visualización y el control de los sistemas CoolBOT. En el caso de este bundle se han implementado dos, a describir en los siguientes apartados: **SonarHTView** y **SonarView**.

Antes de proceder con la descripción de cada vista, es necesario introducir el modelo de interfaz estándar que se sigue en CoolBOT para esta clase de componentes software, representado en la figura 4.18.

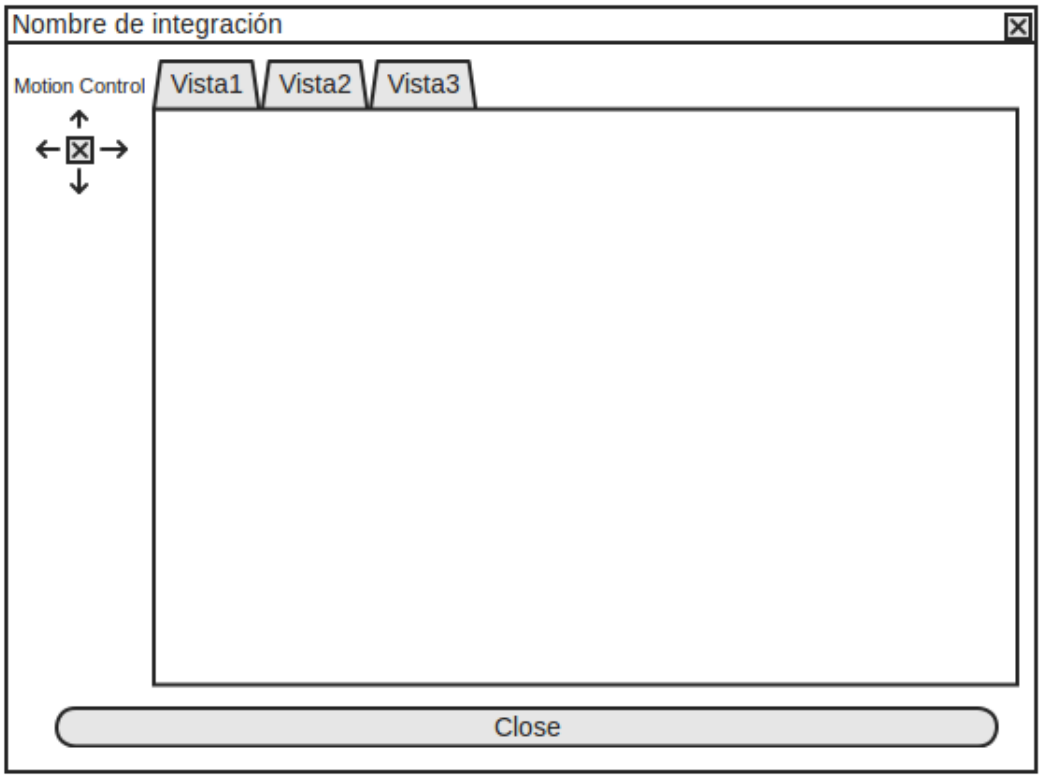


Figura 4.18: Interfaz estándar de las vistas en CoolBOT

La interfaz se comprende de diferentes elementos:

- *Vertical box*: la parte izquierda de la pantalla consiste en un panel vertical común. Es común en las integraciones de CoolBOT incluir un (término que se explica en esta lista) con ciertos botones para controlar el movimiento del robot de forma manual (ver "Motion Control" en sección 4.2.2).

- *Notebook*: cada una de las pestañas (vista1, vista2,...) están vinculadas a la ventana inferior a éstas cuando son seleccionadas, donde se dibujan los elementos de la vista correspondiente; este conjunto es lo que se conoce como *notebook*.

- *Table*: cada vez que se inserta algo en una interfaz, ha de asociarse a una estructura de cierto tamaño, establecido tanto horizontal como verticalmente.

- *Frame*: se conoce así a cierta "subventana" embebida dentro del *vertical box* o del *notebook*, con ciertos elementos dispuestos según cierta *tabla* predefinida.

- *Drawing area*: consiste en un área donde dibujar elementos más allá de primitivas de la librería GTK.

- *Botón de Close*: DUDA.

El nombre de la ventana que engloba a toda la interfaz depende del nombre de la integración que relacione los diferentes componentes software implicados. Por su parte, el nombre de cada una de las pestañas, o títulos de los *notebooks*, vendrá definido en el esqueleto o fichero de configuración de dicha integración.

En otro ámbito, se detallan las funciones genéricas que rigen ambas vistas:

- *widgetCreation*: se trata de la función general de las vistas, y la que es invocada desde la propia integración. Su objetivo es crear, ubicar y mostrar los diferentes elementos externos al *Drawing area* de la vista a la que pertenece, así como llamar a las funciones manejadoras de eventos bajo ciertas condiciones.

- *expose*: se trata de una de las manejadoras de eventos por defecto, y se encarga principalmente de invocar a las diferentes funciones destinadas a utilizar los paquetes de entrada de la vista, según vayan llegando a los puertos de ésta. En general realiza la llamada a las funciones que dibujan en el Drawing area, y de hecho la inicial se suele denominar *drawGrid*.

- *configure*: esta es la otra manejadora de eventos por defecto, y se activa tanto inicialmente como cada vez que se detecta un cambio en el tamaño de la ventana -por ejemplo, al maximizarla-. Inicializa ciertas variables referentes a medidas y posiciones relevantes dentro del área de dibujo, que podrán ser de utilidad para otras funciones. Delega ciertas inicializaciones en una función cuyo nombre es *configureGrid*.

- *resto de manejadoras de eventos*: el resto de este tipo de funciones atienden a casos particulares, como por ejemplo eventos de pulsación de botones en pantalla o de movimiento de rueda de ratón.

- *encargadas del Drawing Area*: tratan y dibujan elementos relacionados con el área de dibujo y hacen uso de los paquetes de entrada a la vista; son las funciones invocadas desde *expose*.

- *funciones auxiliares*: resto de funciones destinadas a servir de apoyo a las ya mencionadas.

SonarHTView

Esta vista fue concebida inicialmente para depurar el contenido de las matrices de la Transformada de Hough, dado que presenta éstas a modo de imágenes RGB junto con cierta información adicional. Sin embargo, se mantuvo principalmente porque ofrece datos numéricos que pudieran ser de interés para el usuario.

Su interfaz de entrada/salida sigue el esquema de la figura 4.19

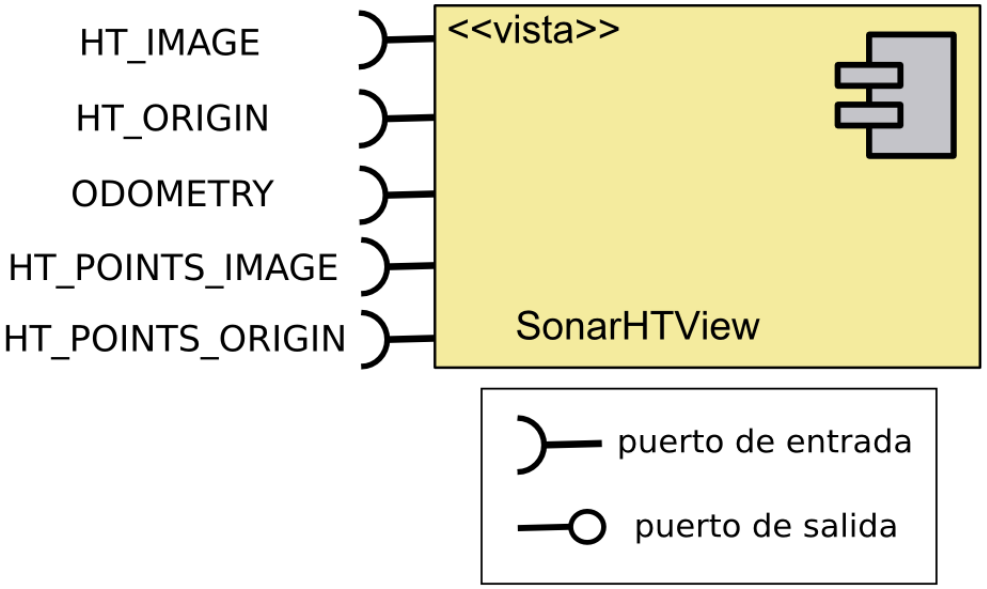


Figura 4.19: Interfaz de sistema de entrada/salida en SonarHTView

Estos son los tipos de paquetes relacionados con los puertos de dicha interfaz:

- **Propios de PlayerRobotPackets** (externo a este *bundle*)
  - *OdometryPacket*: encapsula conjuntos de tres datos: dos de tipo *Frame2D* (nativo de CoolBOT) y uno de tipo *double*. Aparte añade ciertas funciones para el manejo de estos datos. Este tipo de paquetes está destinado a trasladar valores de posición, velocidad y odometría o distancia recorrida del robot. Los paquetes del puerto **ODOMETRY** son de este tipo.
- **Propios de SonarHTPackets**
  - *HTImagePacket*: encapsula los datos necesarios para trasladar imágenes en formato RGB de matrices. Los paquetes de los puertos **HT_IMAGE** y **HT_POINTS_IMAGE** son de este tipo.
  - *HTFramePacket*: encapsula datos cuyo tipo es *Frame2D*. Los paquetes de los puertos **HT_ORIGIN** y **HT_POINTS_ORIGIN** son de este tipo.

Para facilitar la comprensión de la interfaz de entrada/salida, se incluye el fichero descriptivo de la clase SonarHTView, nombrado como *sonar-ht-view.coolbot-view*, dentro del directorio *sonar-ht-view* del bundle, donde se encuentran también los fuentes de esta clase. Su contenido es el siguiente:

```
 view SonarHTView
{

  header
  {
```

```
    author "Javier Hernández Trujillo <phybos86@gmail.com>";
    description "SonarHTGtk view";
    institution "Modelado de sensores sónar en robótica móvil -PFC-";
    version "0.1"
  };

  constants
  {
    private DEFAULT_REFRESHING_PERIOD=300; // milliseconds
    private DEFAULT_SIZE_X=500; // pixels
    private DEFAULT_SIZE_Y=550; // pixels
    private DRAW_AREA_XY_DIVS=4; // should be an even number
    private ARROW_WIDTH=6; // pixels (should be an even number)
    private ARROW_DEEP=6; // pixels
    private FONT_SIZE=12;
  };

  input port HT_IMAGE type poster port packet SonarHTPackets::HTImagePacket;
  input port HT_ORIGIN type poster port packet SonarHTPackets::HTFramePacket;
  input port ODOMETRY type last port packet PlayerRobot::OdometryPacket;
  input port HT_POINTS_IMAGE type poster port packet SonarHTPackets::HTImagePacket;
  input port HT_POINTS_ORIGIN type poster port packet SonarHTPackets::HTFramePacket;

};
```

Una vez descrita la estructura de entrada/salida de esta vista, se procede a detallar los componentes de la interfaz propiamente dicha de la misma, partiendo de la captura 4.20.

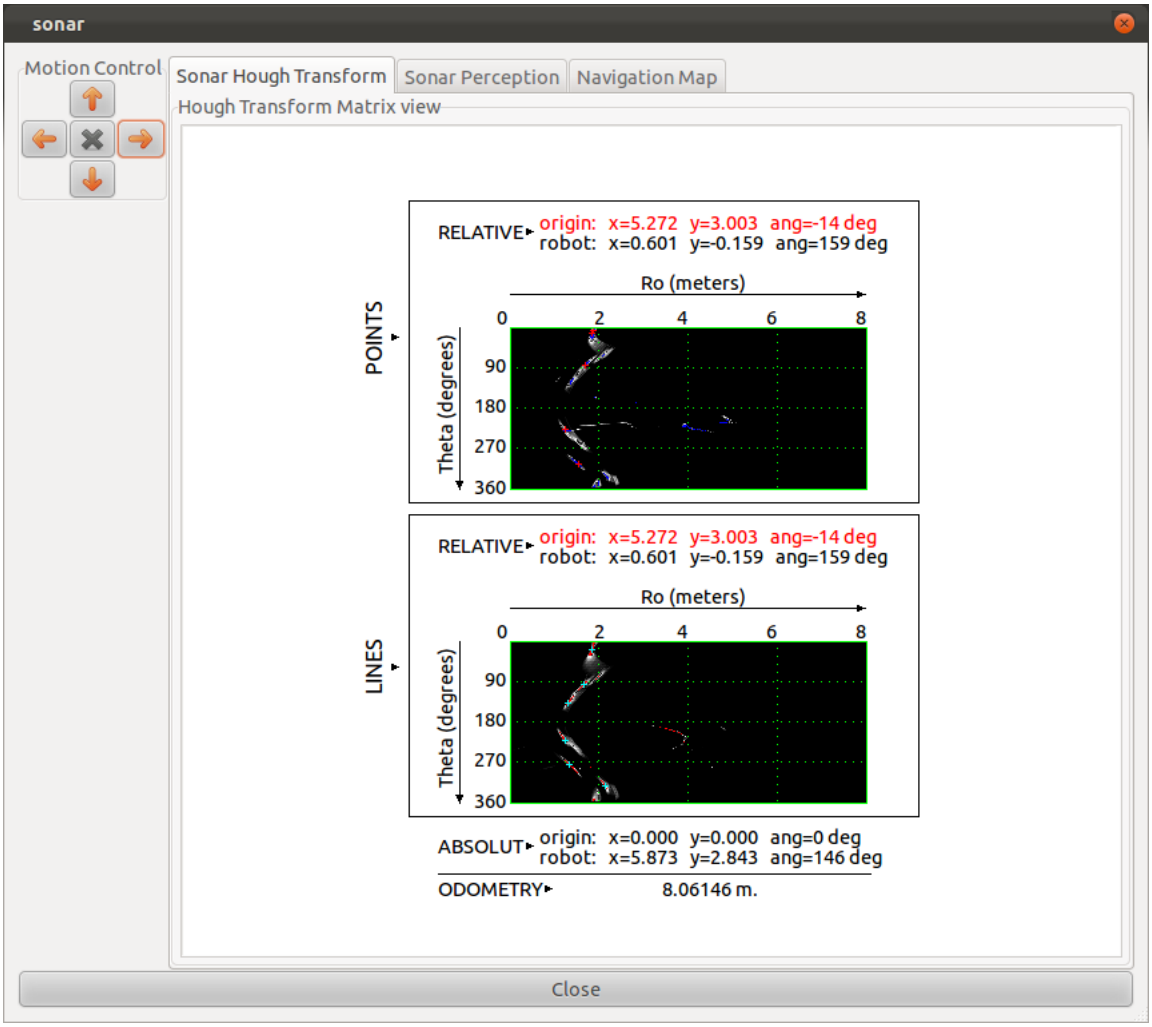


Figura 4.20: Ejemplo de captura de vista SonarHTView

Tal y como se puede apreciar, se muestran dos imágenes de matrices rodeadas por un rectángulo, aparte de otros datos adicionales. Las matrices son las propias de los puntos y las líneas, respectivamente y siguiendo el criterio de dibujo del componente SonarHT, y los datos mostrados aluden a la siguiente información:

- RELATIVE:
  - *origin*: muestra los datos del origen de la Transformada de Hough con respecto al origen absoluto del cual partió el robot; dicho de otro modo, registra los datos del robot con respecto al origen absoluto cuando se desencadena una transformación en la matriz. La matriz dibujada justo debajo está referenciada con respecto a dicho origen.
  - *robot*: muestra la posición actual del robot con respecto a dicho origen de la Transformada de Hough.

- ABSOLUT:
  - *origin*: muestra los datos del origen absoluto, esto es, tanto la posición como el ángulo a 0. Esta es la posición de la que partió el robot inicialmente.
  - *robot*: muestra la posición y el ángulo actual del robot con respecto al origen absoluto.
- *ODOMETRY*: presenta la distancia total recorrida por el robot desde que inició su trayectoria.

Las funciones a destacar en sonar-HT-view son las que siguen:

- *drawGrid*: su uso concreto en este caso es el de dibujar todos los elementos relacionados con los dos rectángulos, inclusive.
- *odometryReturn*: esta función se centra en el resto de elementos, esto es, la parte de ABSOLUT y de ODOMETRY, desde el dibujo de las strings hasta de los propios datos asociados.

SonarView

En esta vista se dibujan todos los elementos resultantes de las etapas del *Análisis del problema*, con respecto al sistema de referencia del robot. Para hacerla más amigable, incluye botones asociados a los diferentes elementos a dibujar para ofrecer al usuario una vista selectiva, entre otras utilidades. Mientras que la vista anterior tenía un cometido meramente informativo y no ofrecía interactividad al usuario, ésta si lo permite hasta cierto nivel.

Siguiendo la forma de proceder hasta ahora, se comenzará describiendo su interfaz de entrada/salida, en la figura 4.21.

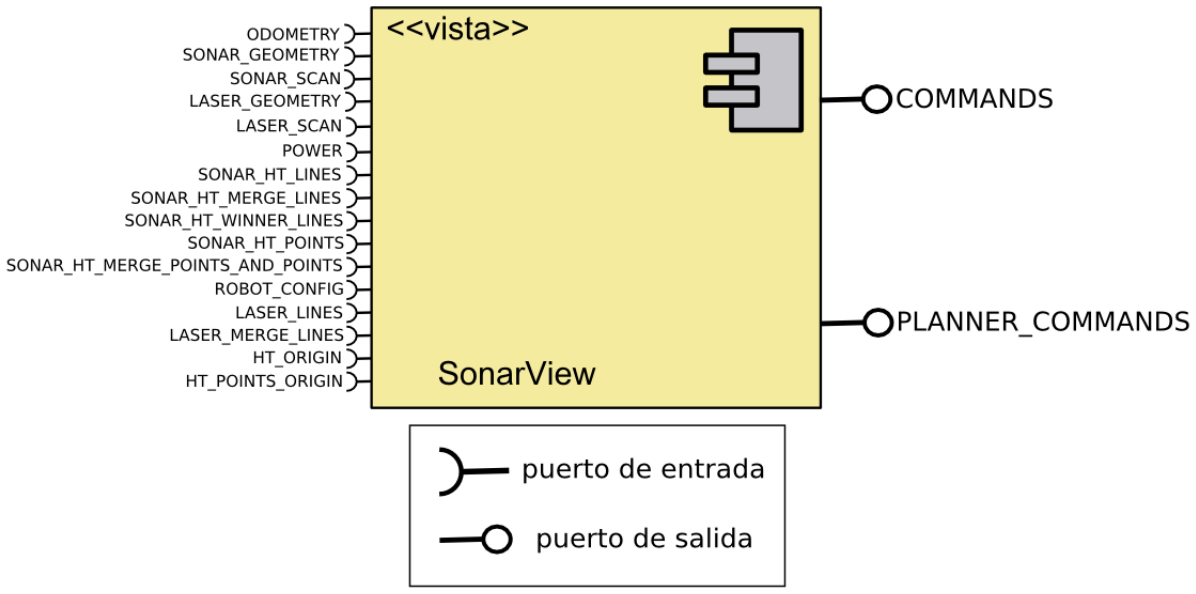


Figura 4.21: Interfaz de sistema de entrada/salida en SonarView

Los tipos de paquetes en este caso se detallan en la siguiente lista:

- *PacketFrame3D*: encapsula datos de tipo *Frame3D* (nativo de CoolBOT), y en concreto se utiliza para indicar la posición del transductor láser con respecto al robot. Los paquetes del puerto **LASER_GEOMETRY** son de este tipo.
- *PacketDouble*: encapsula datos de tipo *double*, y en este ámbito se utiliza para informar sobre el estado de la batería del robot. Los paquetes del puerto **POWER** son de este tipo.
- **Propios de PlannerPackets** (externo a este *bundle*)
  - *CommandPacket*: encapsula conjuntos de dos datos: uno de tipo *entero* y otro de tipo *goal*; *goal* es una estructura con un campo de tipo *Coordinates2D* (nativo de CoolBOT) y otro de tipo *double*. Se utiliza para enviar instrucciones al planificador de rutas. Los paquetes del puerto **PLANNER_COMMANDS** son de este tipo.
- **Propios de PlayerRobotPackets** (externo a este *bundle*)
  - *OdometryPacket*: encapsula conjuntos de tres datos: dos de tipo *Frame2D* (nativo de CoolBOT) y uno de tipo *double*. Aparte añade ciertas funciones para el manejo de estos datos. Este tipo de paquetes está destinado a trasladar valores de posición, velocidad y odometría o distancia recorrida del robot. Los paquetes del puerto **ODOMETRY** son de este tipo.
  - *SonarGeometryPacket*: encapsula datos de tipo *Frame2D*, y en concreto se utiliza para indicar la posición de cada uno de los sensores ultrasónicos con respecto al robot. Los paquetes del puerto **SONAR_GEOMETRY** son de este tipo.
  - *SonarPacket*: encapsula datos de tipo *double*, y particularmente su utilidad es contener las diferentes distancias detectadas en los arcos de barrido por los sensores ultrasónicos. Los paquetes del puerto **SONAR_SCAN** son de este tipo.
  - *LaserPacket*: encapsula conjuntos de tres datos: uno de tipo *Coordinates2D* (nativo de CoolBOT), otro de tipo *entero* y otro de tipo *booleano*. Además añade ciertas funciones para el manejo de estos datos. Este tipo está concebido para trasladar las distancias devueltas por el sensor láser, así como valores de intensidad e invalidez asociados. Los paquetes del puerto **LASER_SCAN** son de este tipo.
  - *ConfigPacket*: encapsula conjuntos de seis datos: uno de tipo *Frame2D*, otro de tipo *Coordinates2D*, otro de tipo *Time* y tres de tipo *double*. Su cometido es contener datos relacionados con las medidas del robot, principalmente. Los paquetes del puerto *ROBOT_CONFIG* son de este tipo.
  - *CommandPacket*: encapsula conjuntos de cinco datos: uno de tipo *entero*, dos de tipo *Frame2D* y dos de tipo *PTZJoints*. Este último tipo es una estructura con tres campos de tipo *double* y un campo de tipo *entero*, y está relacionada con el "pantilt" de la cámara del robot. Este tipo de paquetes sirven para comandar

instrucciones de movimiento para el robot. Los paquetes del puerto *COMMANDS* son de este tipo.

- **Propios de SonarHTPackets**
  - *HTFramePacket*: encapsula datos cuyo tipo es *Frame2D*. Los paquetes de los puertos **HT_ORIGIN** y **HT_POINTS_ORIGIN** son de este tipo.
  - *HTLinesPacket*: encapsula líneas, entendiendo como líneas estructuras con dos campos de tipo *Coordinates2D* (nativo de CoolBOT) y un campo de tipo *unsigned long*. Aparte, aporta ciertas funciones útiles para el manejo de las líneas. Los paquetes de los puertos **SONAR_HT_LINES**, **SONAR_HT_MERGE_LINES**, **SONAR_HT_WINNER_LINES**, **LASER_LINES** y **LASER_MERGE_LINES** son de este tipo.
  - *HTPointsPacket*: igual que el tipo anterior, pero en este caso los puntos son estructuras con un único campo de tipo *Coordinates2D* y otro campo de tipo *unsigned long*. El puerto **SONAR_HT_POINTS** atiende a paquetes de este tipo.
  - *HTSuperPointsAndPointsPacket*: encapsula estructuras que contienen un campo cuyo tipo es la estructura de los puntos del tipo de paquetes anterior, y otro campo que consiste en una lista de estas estructuras. Los paquetes del puerto **SONAR_HT_MERGE_POINTS_AND_POINTS** son de este tipo.

El fichero descriptivo en el caso de la clase SonarView se denomina *sonar-view.coolbot-view*, y se encuentra junto a los fuentes de la clase en el directorio *sonar-view* del bundle. El fichero descriptivo contiene el siguiente código en cuestión:

```
 view SonarView
{

  header
  {
    author "Javier Hernández Trujillo <phybos86@gmail.com>";
    description "SonarView view";
    institution "Modelado de sensores sónar en robótica móvil -PFC-";
    version "0.1"
  };

  constants
  {
    private FIFO_LENGTH=5;  //packets
    private DEFAULT_REFRESHING_PERIOD=250; // milliseconds
    private DRAW_AREA_MARGIN=3; // pixels
    private DRAW_AREA_WIDTH=300; // pixels
    private DRAW_AREA_HEIGTH=560; // pixels
    private DRAW_AREA_XY_DIVS=8; // should be an even number
    private GRID_AXES_FONT_SIZE=8; // font points
```

```
    private ARROW_WIDTH=6; // pixels (should be an even number)
    private ARROW_DEEP=6; // pixels
    private ARROW_WIDTH_AUX=20; // pixels (should be an even number)
    private ARROW_DEEP_AUX=20; // pixels
    private HALF_ARC_ANGLE=5; // degrees
    private DIRECT_COMMAND_TRANSLATIONAL_SPEED= 250; // millim/sec
    private DIRECT_COMMAND_ROTATIONAL_SPEED= 20; // degree/sec
    private PERCEPTION_RADIUS = 2; //on meters
    private CUTTING_RADIUS = 8; //on meters
    private MINIMUM_RANGE_SCALE_VALUE = 1; //on meters
    private MAXIMUM_RANGE_SCALE_VALUE =
       "PlayerRobotSpace::PlayerRobot::LASER_MAX_RANGE/1000"; //on meters
  };

  input port ODOMETRY type last port packet PlayerRobot::OdometryPacket;
  input port SONAR_GEOMETRY type poster port packet PlayerRobot::SonarGeometryPacket;
  input port SONAR_SCAN type last port packet PlayerRobot::SonarPacket;
  input port LASER_GEOMETRY type poster port packet PacketFrame3D;
  input port LASER_SCAN type last port packet PlayerRobot::LaserPacket;
  input port POWER type last port packet PacketDouble;
  input port SONAR_HT_LINES type fifo port packet SonarHTPackets::HTLinesPacket
                                                          length FIFO_LENGTH;
  input port SONAR_HT_MERGE_LINES type last port packet SonarHTPackets::HTLinesPacket;
  input port SONAR_HT_WINNER_LINES type last port packet SonarHTPackets::HTLinesPacket;
  input port SONAR_HT_POINTS type fifo port packet SonarHTPackets::HTPointsPacket
                                                            length FIFO_LENGTH;
  input port SONAR_HT_MERGE_POINTS_AND_POINTS type last port packet
                       SonarHTPackets::HTSuperPointsAndPointsPacket;
  input port ROBOT_CONFIG type poster port packet PlayerRobot::ConfigPacket;
  input port LASER_LINES type poster port packet SonarHTPackets::HTLinesPacket;
  input port LASER_MERGE_LINES type poster port packet
                         SonarHTPackets::HTLinesPacket;
  input port HT_ORIGIN type poster port packet
                  SonarHTPackets::HTFramePacket;
  input port HT_POINTS_ORIGIN type poster port packet
                        SonarHTPackets::HTFramePacket;

  output port COMMANDS type generic port packet PlayerRobot::CommandPacket;
  output port PLANNER_COMMANDS type generic port packet Planner::CommandPacket;

};
```

La imagen 4.22 es una captura de esta vista, donde se puede observar dos *frames* a nivel general, el propio del área de dibujo y otro con diferentes botones.

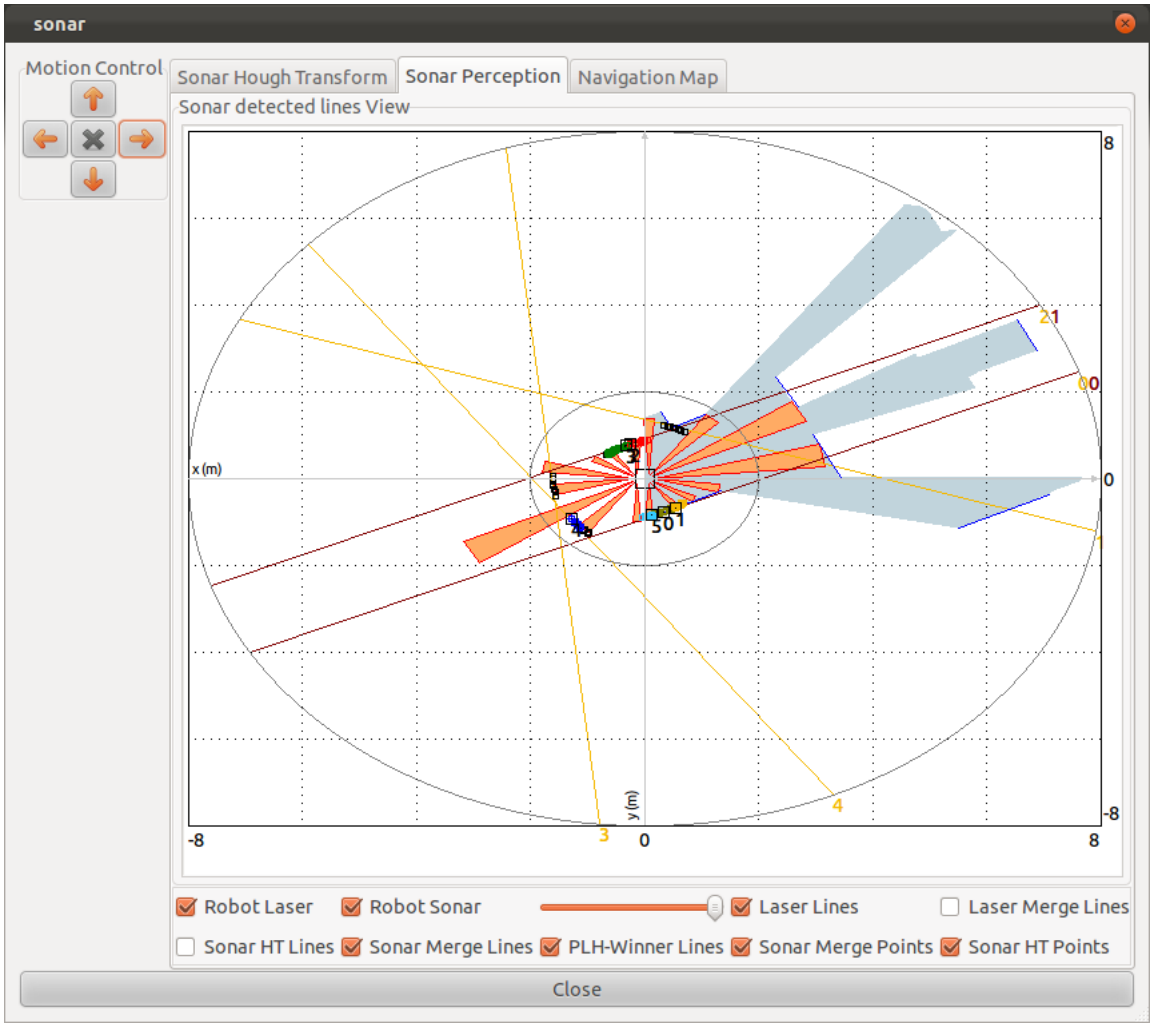


Figura 4.22: Ejemplo de captura de vista SonarView

En la parte superior se distinguen dos grandes círculos (aunque tienen forma de elipse por el escalado de la ventana). El mayor sirve de punto de corte de las líneas al dibujarlas, dado que son infinitas. El círculo interno marca el límite de detección impuesto para los sensores ultrasónicos, dado que, como se mencionaba en apartados anteriores, aquellos puntos cuya distancia con respecto a los sónars supere cierto umbral no serán registrados -en concreto, 2 metros-.

En el centro de ese *frame* hay un rectángulo, y sus medidas se corresponden con las del robot escalado según la distancia del resto de elementos que se dibujan. Como se citaba en la introducción a esta vista, todos los elementos estarán representados con respecto a este centro, es decir, con respecto al sistema de referencia del robot, que está orientado hacia la derecha.

En el *frame* de la parte inferior se ubican diferentes botones del tipo *toggle button*. A continuación se detallarán acompañados de capturas que los complementen, y cabe señalar que las capturas se hicieron exactamente para la rotación del robot que aparece en la figura 4.23. Las capturas de los cuatro primeros botones que se detallan se hicieron cuando el robot se encontraba en la posición de la izquierda, mientras que el resto de capturas fueron tomadas

tras haber realizado la rotación y ya en la posición de la derecha.

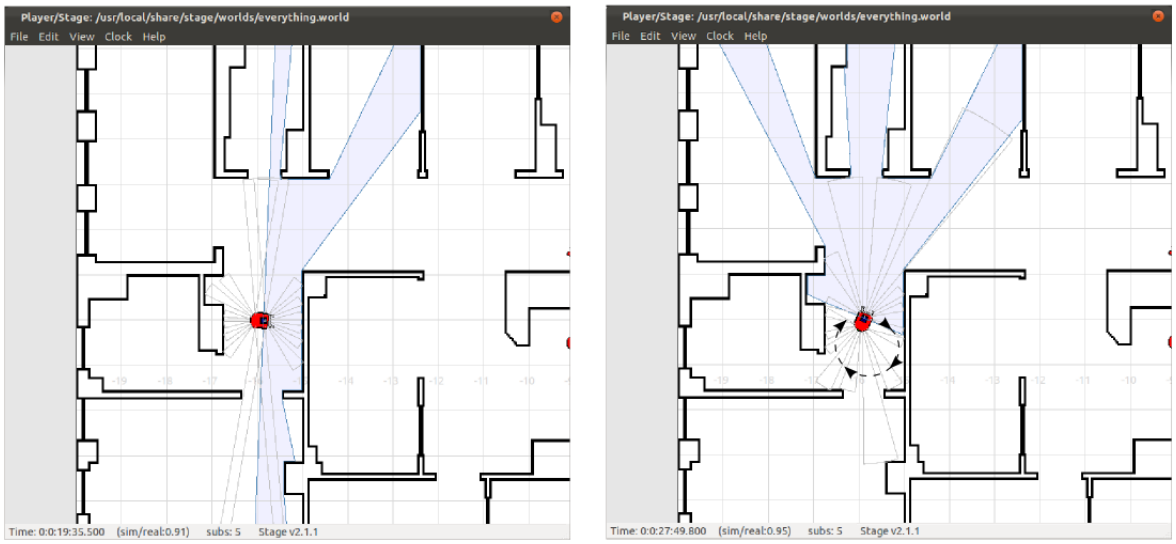


Figura 4.23: Posición inicial y final tras rotación del robot en Stage

- **Robot Laser** -figura 4.24-: la marca de color azul claro que acoge 180 grados por delante del robot representa los haces del sensor láser.

- **Robot Sonar** -figura 4.24-: los triángulos alargados de color naranja representan los arcos de barrido de los sensores ultrasónicos.

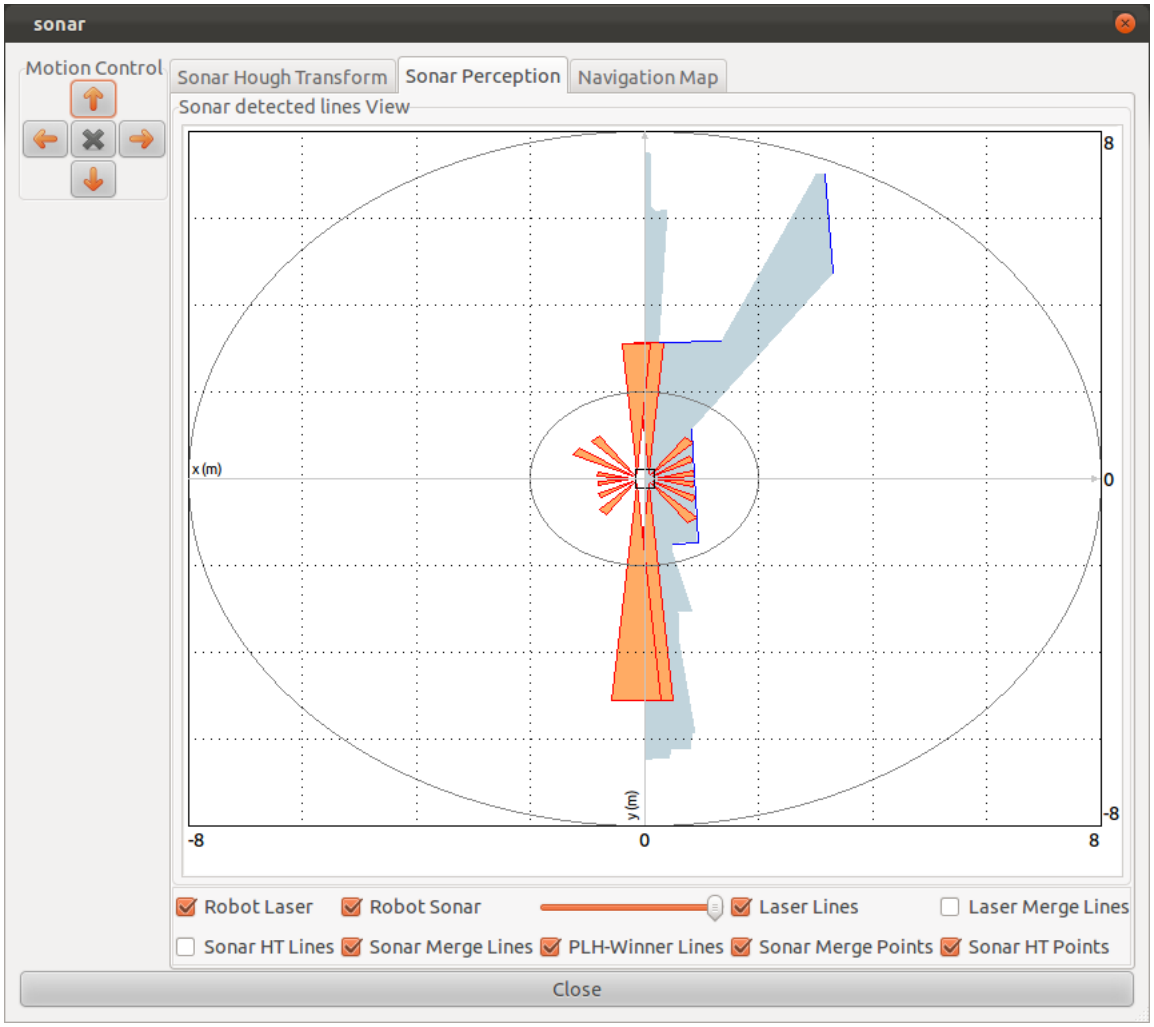
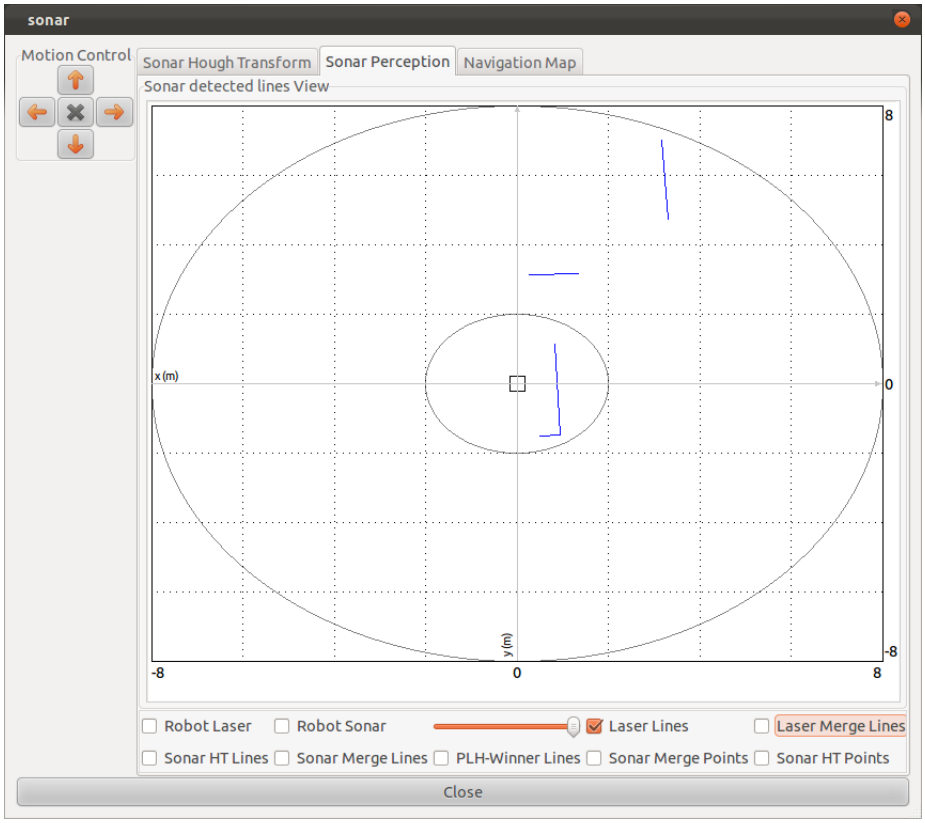


Figura 4.24: Vista original para posición inicial del robot

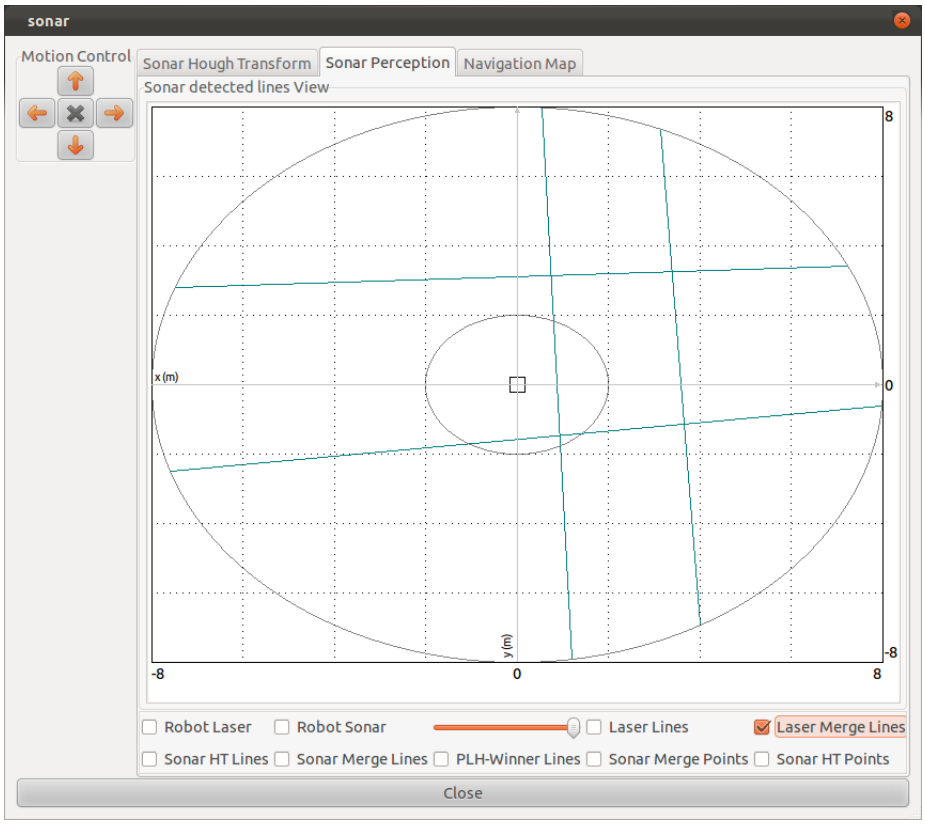
- **Laser Lines** -figura 4.25(a)-: las pequeñas rectas azules que delimitan los obstáculos se corresponden con la salida de la Etapa 2: División de conjuntos de puntos y generación de líneas (sección 3.3.2).
- **Laser Merge Lines** -figura 4.25(b)-: las líneas infinitas de color turquesa son el resultado de la Etapa 3: Fusión de líneas (sección 3.3.3).
- **Sonar HT Lines** -figura 4.26(a)-: las líneas rojas son las líneas obtenidas en la Etapa 5: Extracción de líneas y puntos en matrices de la Transformada de Hough (sección 3.2.5).
- **Sonar Merge Lines** -figura 4.26(b)-: las líneas amarillas son las propias de la Etapa 6: Fusión de las líneas y puntos extraídos (sección 3.2.6).
- **PLH-Winner Lines** -figura 4.26(c)-: las líneas marrones resultan de la Etapa 7: Hipótesis de líneas (sección 3.2.7).
- **Sonar Merge Points** -figura 4.27(a)-: los conjuntos de puntos se obtienen de la Etapa 5: Extracción de líneas y puntos en matrices de la Transformada de Hough (sección

3.2.5), o lo que es lo mismo, los subconjuntos asociados a los puntos fusionados en la Etapa 6: Fusión de las líneas y puntos extraídos (sección 3.2.6). Los colores permiten distinguir grupos de puntos que generan los puntos fusionados de **Sonar HT Points**, y de hecho los puntos negros son aquellos que no contribuyen a generar puntos fusionados.

- **Sonar HT Points** -figura 4.27(b)-: los puntos con un identificador y rodeados de un cuadrado son aquellos que surgen de la Etapa 6: Fusión de las líneas y puntos extraídos (sección 3.2.6).
- **Barra horizontal**: la barra horizontal dispuesta entre los botones *Robot Sonar* y *Laser Lines* ejerce de zoom para los elementos del área de dibujo hasta cierto límite inferior (1 metro) y superior (8 metros) -cuando se intentan rebasar, se avisa al usuario mediante un mensaje por consola-. Hacia la derecha disminuye el zoom, por lo que permite ver una mayor cantidad de elementos en pantalla; hacia la izquierda produce el efecto opuesto. El valor de esta barra también está sujeto al movimiento de la rueda del ratón en el área de dibujo (evento de *scroll*): girando la rueda hacia arriba/adelante se aumenta el zoom, mientras que para disminuirlo habrá de girarse la rueda en sentido contrario. Un ejemplo del efecto conseguido se muestra en la figura 4.28.

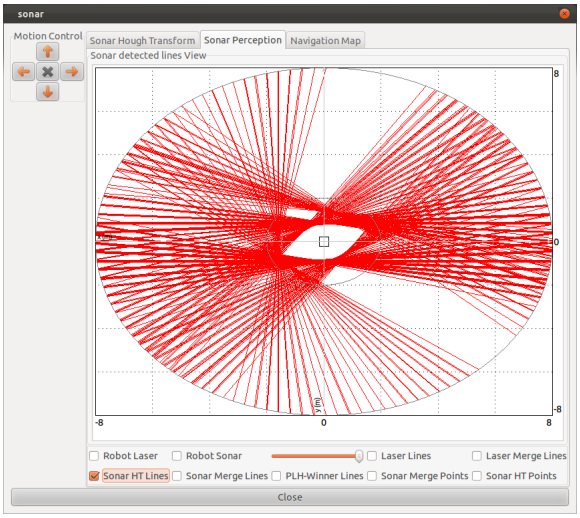


(a)

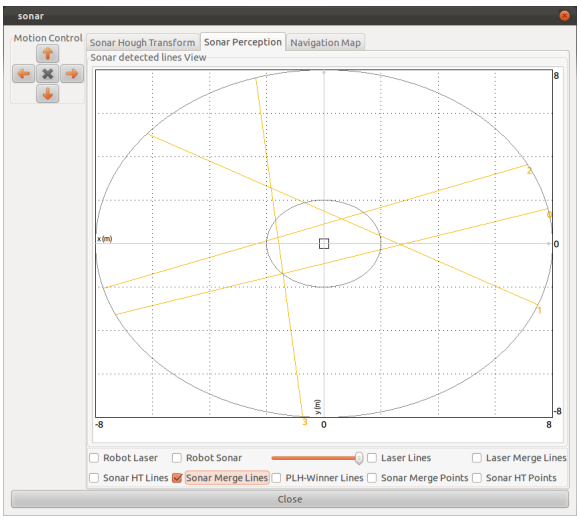


(b)

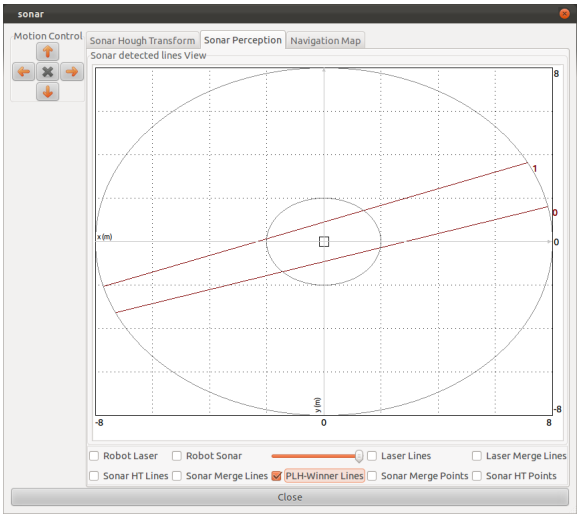
Figura 4.25: Vistas asociadas a los diferentes *toggle buttons* vinculados al láser



(a)

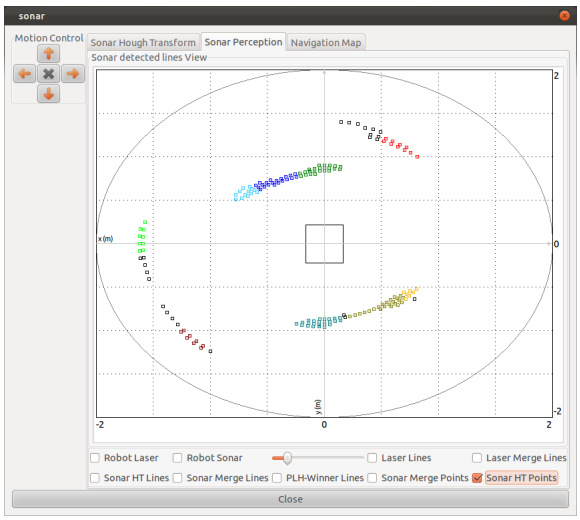


(b)

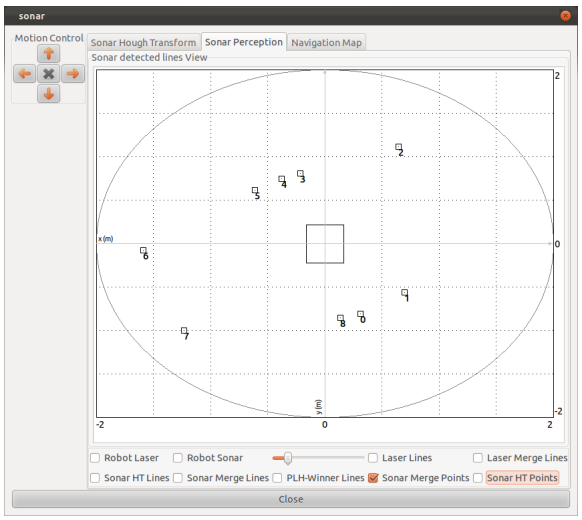


(c)

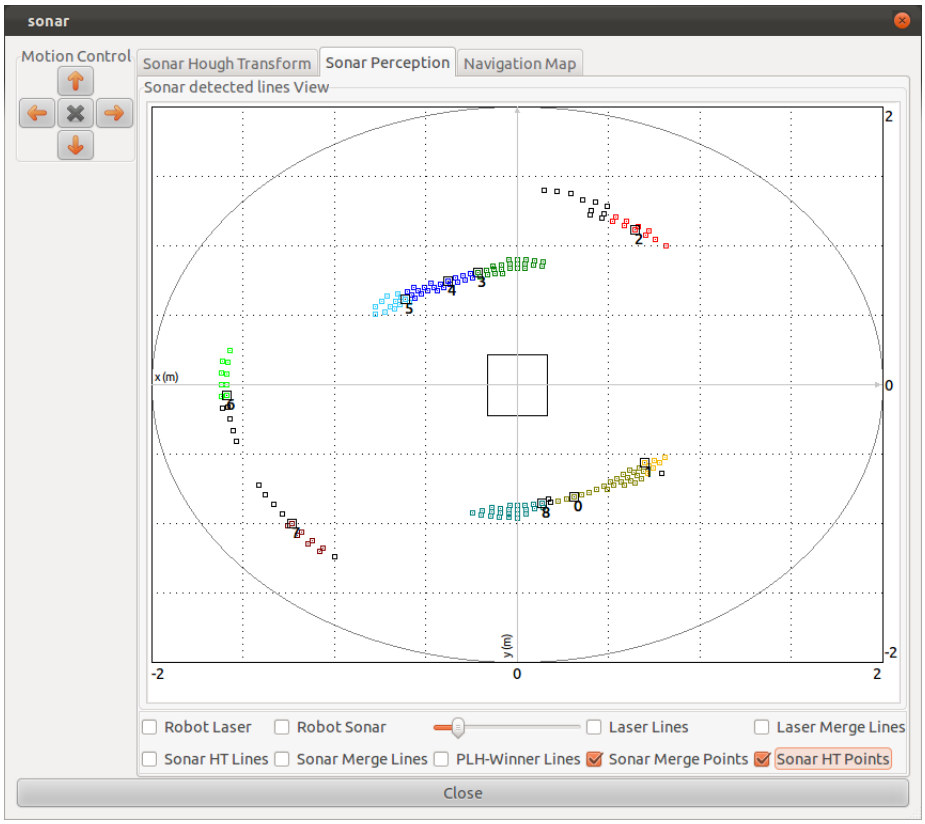
Figura 4.26: Vistas asociadas a los diferentes *toggle buttons* vinculados a las líneas del sonar



(a)

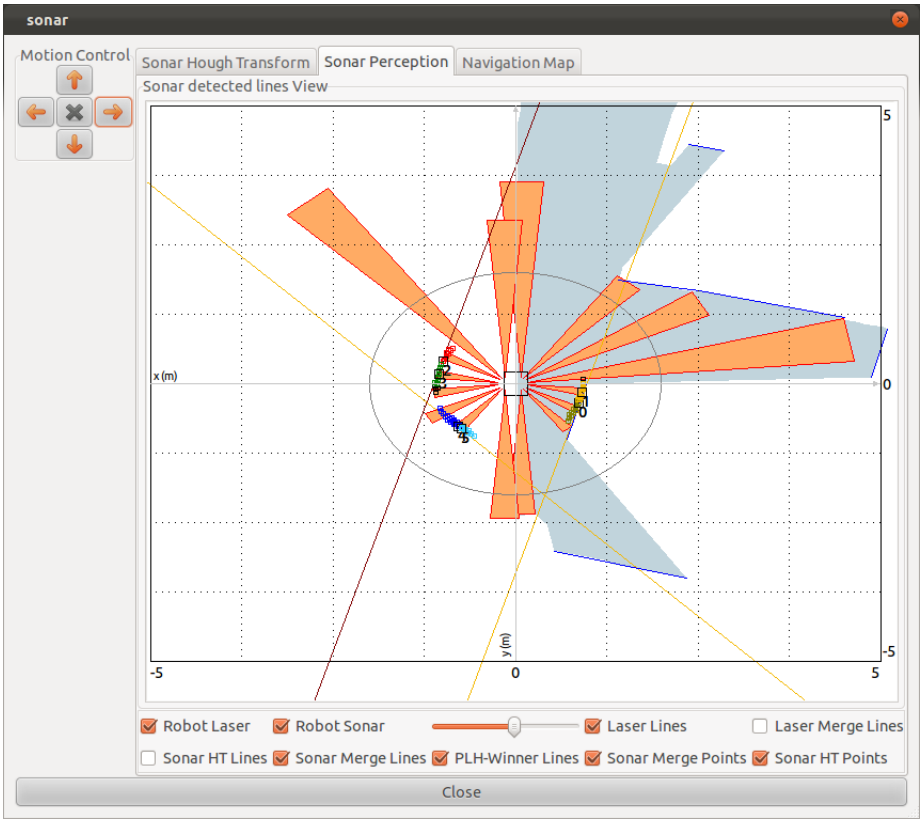


(b)

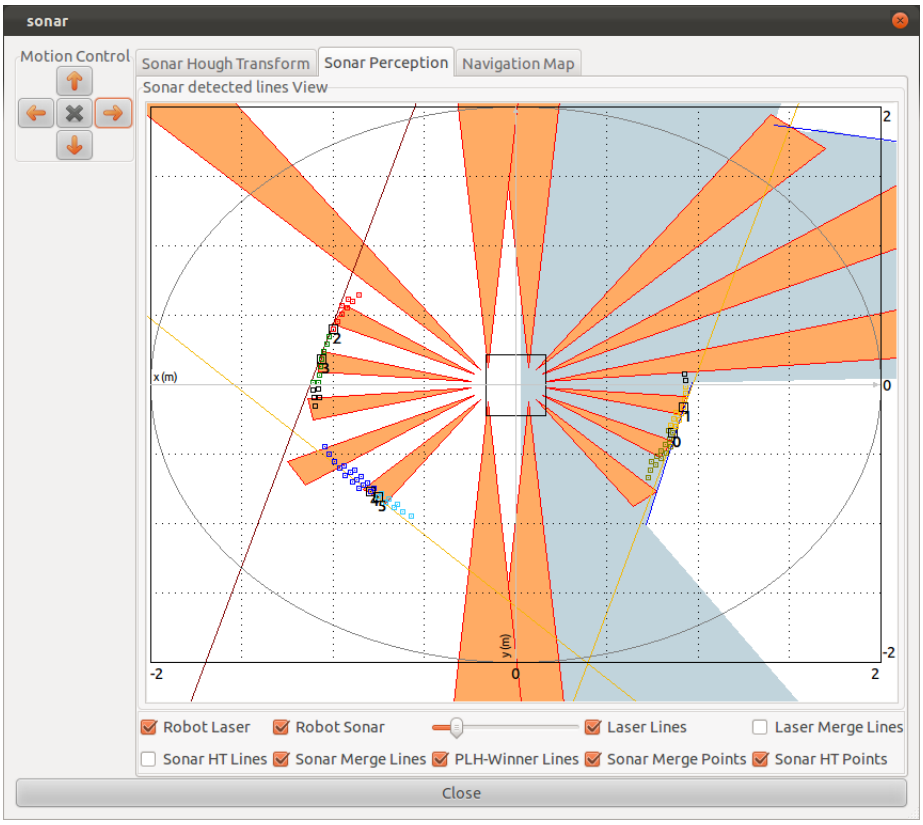


(c)

Figura 4.27: En (a) y (b) se muestran las vistas asociadas a los diferentes *toggle buttons* vinculados a los puntos del sonar, aplicando la utilidad de *zoom* de la vista para aclarar la captura. En (c) se muestra la superposición de las imágenes generadas por sendos botones.



(a)



(b)

Figura 4.28: Escalado de elementos del área de dibujo mediante la utilidad de zoom

Las funciones más relevantes de SonarView son las siguientes:

- *drawGrid*: su función en este caso es pintar el marco y las líneas discontinuas que indican las distintas distancias, así como las diferentes strings, dentro del área de dibujo.
- *sonarScan*: pinta los elementos vinculados al botón **Robot Sonar**.
- *laserScan*: idem para el botón **Robot Laser**.
- *drawLines*: se encarga de pintar el resto de elementos del área de dibujo.
- *mouseButton*: maneja las interrupciones provocadas por botones del ratón. En esta implementación sólo está preparado para comandar al planificador de rutas que el robot se detenga cuando se pulse el botón derecho sobre el área de dibujo (puerto PLANNER_COMMANDS).
- *mouseScroll*: maneja las interrupciones provocadas por la rueda del ratón. Dependiendo de hacia dónde gire la rueda, principalmente incrementa o decrementa el *factor de escala*, una variable a la que atiende la función *configure* para establecer las medidas de los elementos en el área de dibujo. Adicionalmente actualiza la posición de la barra horizontal de zoom según el valor actual de dicho factor de escala.
- *zoomChanged*: se activa cuando se mueve la barra horizontal del zoom, y tiene la misma utilidad que la función *mouseScroll*.
- *pointsColorId*: esta función se basa en el orden en el que lleguen los paquetes con los puntos del puerto SONAR_HT_MERGE_POINTS_AND_POINTS, con tal de que cada subconjunto de puntos tenga un color distinto, dentro del abanico de los 10 posibles colores.

Las funciones que manejan los eventos de pulsación de los botones sencillamente manipulan *flags* o variables de control globales. Cada vez que se invocan, cambian el valor de estas variables, de las que a su vez depende el que los elementos asociados a cada botón se pinten o no.

El *frame* "Motion control", así como sus botones (nativos de librería GTK) y el manejo de éstos, se implementan también en esta vista, dado que es necesario hacerlo en una de las vistas del bundle a implementar si se desea hacer uso de esta utilidad. Los botones con la flecha hacia arriba y hacia abajo permiten mover el robot hacia delante y hacia atrás, respectivamente. Los botones con las flechas hacia la izquierda y derecha permiten el giro del robot sobre si mismo. El botón central con la X dibujada no tiene una función predefinida, pero está preparado para que pueda serle asociada. Las funciones encargadas de los eventos de estos botones envian comandos a través del puerto COMMANDS según el botón que haya provocado la interrupción.

Paquetes de puertos

En este apartado se centralizará la información relacionada con los tipos de paquetes implementados para este *bundle*, desde para qué fueron concebidos en un principio hasta los tipos de datos que encapsulan. Las diferentes clases implementan funciones adicionales tanto para el protocolo de transporte como para facilitar el manejo de las variables vinculadas a los puertos. Asi pues, los tipos son los siguientes:

- **HTImagePacket**: para enviar las imágenes de las matrices de la Transformada de Hough del componente *SonarHT* a la vista *SonarHTView* era necesario implementar un tipo de paquete como éste. Encapsula datos de tipo GridType, una redefinición del tipo RGBGridType, utilizado para imágenes de matrices en formato RGB con celdas de tipo CellType, una redefinición del tipo RawRGB.
- **HTFramePacket**: este tipo fue implementado para enviar los orígenes relativos de las matrices de la Transformada de Hough, tanto para líneas como para puntos, desde el componente *SonarHT* a las dos vistas. Encapsula datos de tipo *Frame2D*, nativo de CoolBOT.
- **HTLinesPacket**: su utilidad es registrar los datos necesarios de las líneas enviadas desde los dos componentes (SonarHT y LaserIEPF) a la vista *SonarView*. En concreto encapsula conjuntos de tuplas con dos campos de tipo *Coordinates2D* -nativo de CoolBOT- para representar los dos puntos que establecen el vector director de cada línea, aparte de un campo de tipo *unsigned long* para los votos de cada una de éstas.
- **HTPointsPacket**: en su caso, se encarga de registrar los datos referentes a los puntos, que asimismo se envian desde el componente *SonarHT* a la vista *SonarView*. Encapsula conjuntos de tuplas de dos campos: uno de tipo *Coordinates2D* para representar el punto en sí y otro de tipo *unsigned long* para almacenar su número de votos.
- **HTSuperPointsAndPointsPacket**: este último tipo permite enviar regiones de puntos junto al punto obtenido de la fusión de cada una de éstas, particularmente desde el componente *SonarHT* a la vista *SonarView*. Encapsula conjuntos de tuplas con dos campos: uno de tipo *cartesianParam* para registrar el punto resultado de la fusión junto a sus votos -propio de la clase Merge, ya explicada en la sección 4.2.1- y otro que consiste en una lista con elementos de este tipo, para almacenar cada uno de los puntos de cada región con su respectivo número de votos.

4.2.3. Integración sonar

La integración denominada **sonar** es la encargada del conexionado y comunicación entre los componentes software de este *bundle*, aparte de entre éstos y determinados componentes software externos al *bundle*.

En la ilustración 4.29 se presenta un diagrama que representa a grandes rasgos el conexionado entre los diferentes componentes software. Los propios del bundle *coolbot-sonar-bundle*, implementado en este proyecto, están pintados en color verde. El resto pertenece al bundle

coolbot-sns-bundle. Ambos se encuentran disponibles en la web de CoolBOT *http://coolbotproject.dis.ulpgc.es/coolbot-project/download/*.

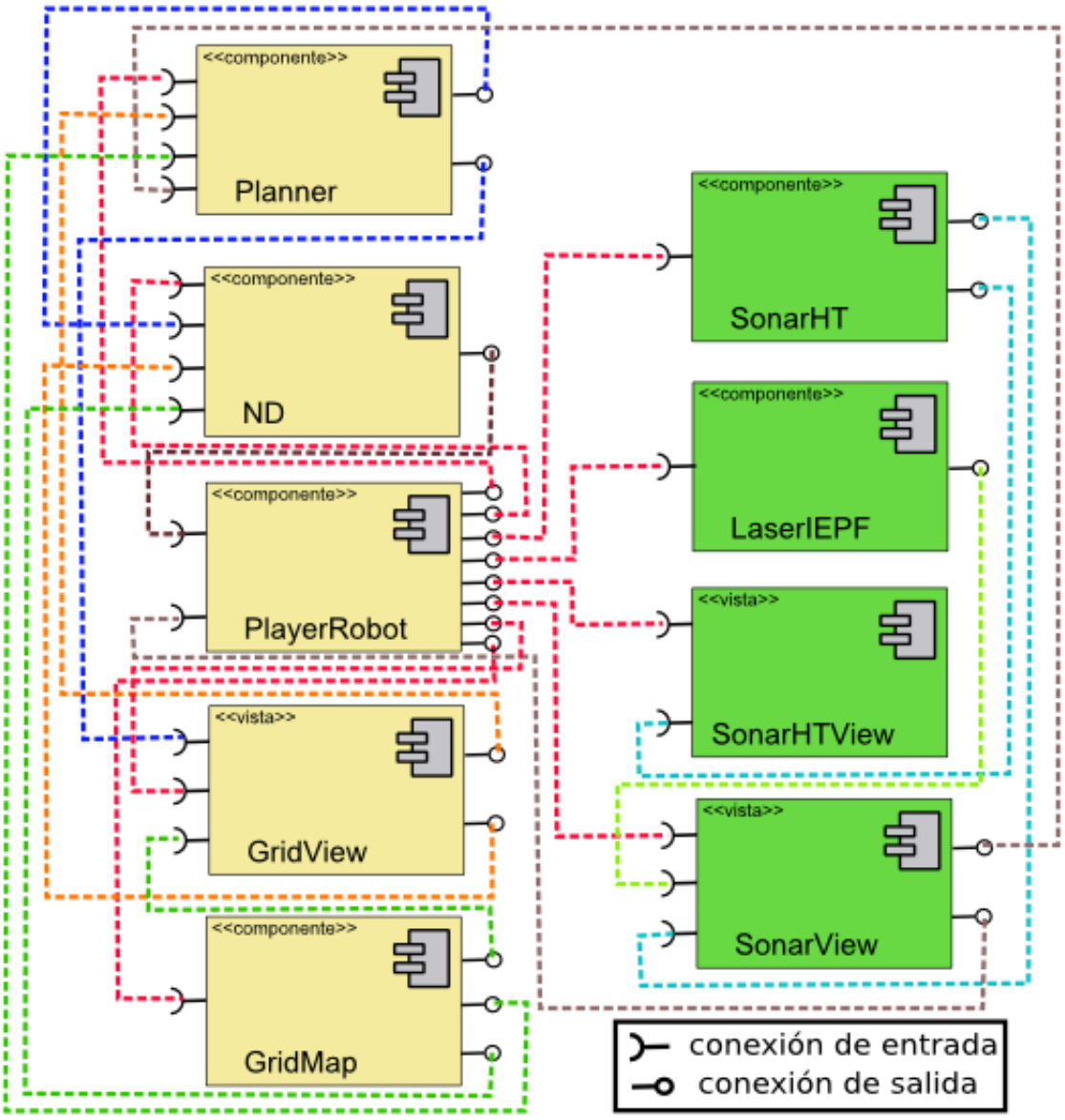


Figura 4.29: Esquema ilustrativo de conexiones para la integración sonar

La figura 4.29 representa el contenido del fichero descriptivo de la integración *sonar*, cuyo nombre es *sonar.coolbot-integration* y que se encuentra en el directorio *sonar* junto con el fichero fuente de la integración, dentro del bundle:

```
 integration sonar-
{

  header
  {
```

```
  author "Javier Hernández Trujillo <phybos86@gmail.com>";
  description "Sonar Hough Transform integration";
  institution "Modelado de sensores sónar en robótica móvil -PFC-";
  version "0.1"
};

machine addresses
{
  /*
   * Machine addresses definition.
   */
  local javier_mac: "127.0.0.1";
};

local instances
{
  /*
   * Local instances definition.
   */
  component robot: PlayerRobot;
  component sonar_HT: SonarHT;
  component laser_IEPF: LaserIEPF;

  component navigationMap: GridMap;
  component nd: ND;
  component navigationPlanner: Planner;

  view sonarHTView: SonarHTView with description "Sonar Hough Transform";
  view sonarView: SonarView with description "Sonar Perception";
  view navigationMapView: GridView with description "Navigation Map";
};

port connections
{
  connect robot:ODOMETRY to sonarView:ODOMETRY;
  connect robot:SONARGEOMETRY to sonarView:SONAR_GEOMETRY;
  connect robot:SONARSCAN to sonarView:SONAR_SCAN;
  connect robot:LASERGEOMETRY to sonarView:LASER_GEOMETRY;
  connect robot:LASERSCAN to sonarView:LASER_SCAN;
  connect robot:POWER to sonarView:POWER;
  connect sonar_HT:SONAR_HT_LINES to sonarView:SONAR_HT_LINES;
  connect sonar_HT:SONAR_HT_MERGE_LINES to sonarView:SONAR_HT_MERGE_LINES;
  connect sonar_HT:SONAR_HT_WINNER_LINES to sonarView:SONAR_HT_WINNER_LINES;
  connect sonar_HT:SONAR_HT_MERGE_POINTS_AND_POINTS to
            sonarView:SONAR_HT_MERGE_POINTS_AND_POINTS;
  connect sonar_HT:SONAR_HT_POINTS to sonarView:SONAR_HT_POINTS;
```

```
connect sonar_HT:SONAR_HT_MERGE_POINTS to sonarView:SONAR_HT_MERGE_POINTS;
connect sonar_HT:SONAR_HT_WINNER_POINTS to sonarView:SONAR_HT_WINNER_POINTS;
connect sonar_HT:SONAR_HT_ORIGIN to sonarView:HT_ORIGIN;
connect sonar_HT:SONAR_HT_POINTS_ORIGIN to sonarView:HT_POINTS_ORIGIN;
connect robot:ROBOTCONFIG to sonarView:ROBOT_CONFIG;
connect laser_IEPF:LASER_IEPF_LINES to sonarView:LASER_LINES;
connect laser_IEPF:LASER_IEPF_MERGE_LINES to sonarView:LASER_MERGE_LINES;

connect sonarView:COMMANDS to robot:COMMANDS;
connect sonarView:PLANNER_COMMANDS to navigationPlanner:COMMANDS;

connect robot:ODOMETRY to sonar_HT:ODOMETRY;
connect robot:SONARGEOMETRY to sonar_HT:SONAR_GEOMETRY;
connect robot:SONARSCAN to sonar_HT:SONAR_SCAN;

connect robot:ODOMETRY to laser_IEPF:ODOMETRY;
connect robot:LASERGEOMETRY to laser_IEPF:LASER_GEOMETRY;
connect robot:LASERSCAN to laser_IEPF:LASER_SCAN;

connect sonar_HT:SONAR_HT_IMAGE to sonarHTView:HT_IMAGE;
connect sonar_HT:SONAR_HT_ORIGIN to sonarHTView:HT_ORIGIN;
connect sonar_HT:SONAR_HT_POINTS_IMAGE to sonarHTView:HT_POINTS_IMAGE;
connect sonar_HT:SONAR_HT_POINTS_ORIGIN to sonarHTView:HT_POINTS_ORIGIN;

connect robot:ODOMETRY to sonarHTView:ODOMETRY;

connect robot:ROBOTCONFIG to nd:ROBOTCONFIG;

connect robot:ROBOTCONFIG to navigationMap:ROBOTCONFIG;
connect robot:ODOMETRY to navigationMap:ODOMETRY;
connect robot:LASERGEOMETRY to navigationMap:LASERGEOMETRY;
connect robot:LASERSCAN to navigationMap:LASERSCAN;

connect robot:ROBOTCONFIG to navigationPlanner:ROBOTCONFIG;
connect robot:ODOMETRY to navigationPlanner:ODOMETRY;

connect navigationMap:GRIDCONFIG to nd:GRIDCONFIG;
connect navigationMap:MAP to nd:MAP;
connect navigationMap:GRIDCONFIG to navigationPlanner:GRIDCONFIG;
connect navigationMap:MAP to navigationPlanner:GRIDMAP;

connect nd:ROBOTCOMMANDS to robot:COMMANDS;

connect navigationPlanner:NAVIGATIONCOMMANDS to nd:COMMANDS;

connect robot:ROBOTCONFIG to navigationMapView:ROBOT_CONFIG;
```

```
    connect navigationMap:MAP to navigationMapView:GRID_MAP;

    connect navigationPlanner:PLANNERPATH to navigationMapView:PLANNER_PATH;
    connect navigationPlanner:MATCHINGREGIONS to navigationMapView:MATCHING_REGIONS;

    connect navigationMapView:PLANNER_COMMANDS to navigationPlanner:COMMANDS;
    connect navigationMapView:ND_COMMANDS to nd:COMMANDS;
  };

};
```

Finalmente se van a introducir los diferentes componentes incluidos en el bundle *coolbot-sns-bundle*:

- **PlayerRobot:** componente para la abstracción de hardware usando el framework Player-Stage (sección 2.2).

- **ND:** componente que implementa el algoritmo *ND+* de navegación segura [Minguez,2004].

- **Planner:** componente que usa el *grid* generado por el componente GridMap (que se explicará en el siguiente punto) para planificar rutas en el entorno de los alrededores del robot usando una modificación de la función numérica de navegación *NF2* [Latombe,1991].

- **GridMap:** componente que construye un *grid* con el entorno de los alrededores del robot usando los datos del sensor de rango láser del mismo. Además genera periódicamente un escaneo virtual de 360 grados para el algoritmo *ND+*.

- **GridView:** vista añadida al bundle para incorporar en la misma interfaz sus prestaciones. En la figura 4.30 se aprecia como se incorpora a la interfaz de la integracion *sonar*, propia del bundle implementado en este proyecto (*coolbot-sonar-bundle*).

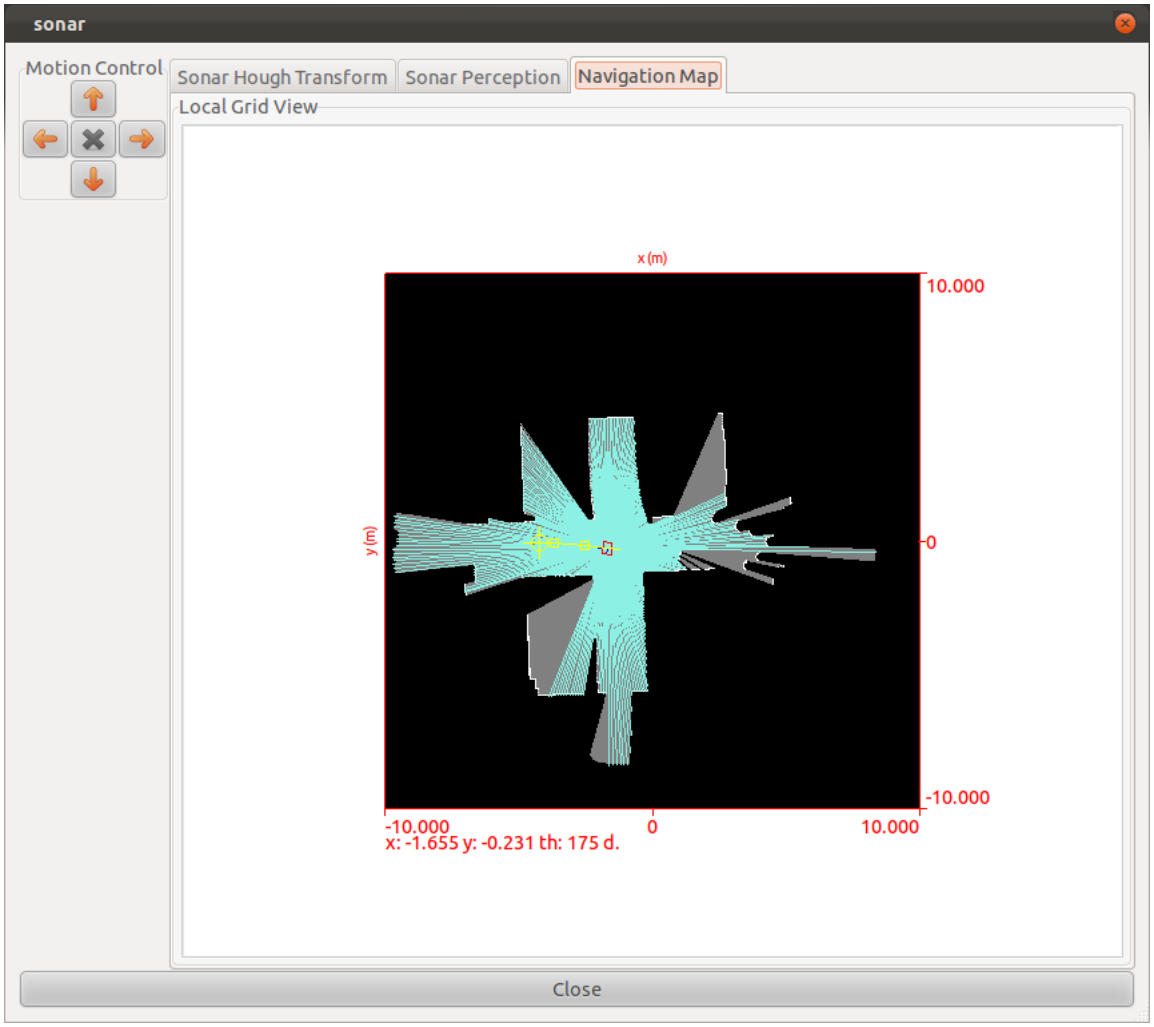


Figura 4.30: Captura de vista GridView, embebida en integración sonar

A grandes rasgos, la vista muestra el mapa generado por GridMap y permite dos interrupciones mediante ratón: pulsando el botón izquierdo, el usuario puede marcar el destino que pretende que alcance el robot en el mapa, y de esta forma la vista invoca a los componentes ND y Planner para lograr alcanzar el objetivo; pulsando el botón derecho del ratón, el usuario puede detener el movimiento del robot en cualquier momento.

4.3. Pruebas

Dado el proceso de desarrollo iterativo e incremental, las pruebas de los distintos prototipos se iban realizando en cada iteración, a medida que éstos alcanzaban los objetivos propuestos para su desarrollo. El tipo de pruebas sigue el modelo empírico, dado que se dan por concluidas cuando visualmente se aprecian los resultados previstos.

Las pruebas del prototipo en Matlab perseguían lo citado a continuación:

- Cálculo correcto del *Punto q*: visualmente tendría que coincidir con la normal entre el origen absoluto y la tangente que contenía a los puntos. En consecuencia, un buen cálculo del *Punto q* garantiza un cálculo correcto de las transformaciones entre los distintos sistemas de coordenadas para ambos puntos de cada tangente con respecto a cada sensor.
- Detección correcta de obstáculos: si se detectan bien los obstáculos, dentro claro está de cierto margen de error, todos los pasos previos serán correctos, es decir:
  - Cálculo de *Punto q*.
  - Cálculo de discretización de parámetros polares de cada *Punto q* para la inserción en la matriz de la Transformada de Hough.

En el capítulo de *Diseño de prototipos, implementación y pruebas* (capítulo 4), las diferentes capturas se fundamentan en los datos iniciales que se presentan, de modo que se puede considerar un ejemplo de prueba y consecución de los objetivos planteados para el prototipo en Matlab.

Por otro lado, las pruebas vinculadas al prototipo final en CoolBOT tenían como objeto el ajuste de los algoritmos implementados y de los umbrales, con tal de que resultara la mayor coincidencia posible entre las detecciones de los sensores ultrasónicos y los objetos realmente presentes, tanto en el entorno simulado de *Stage* como en el real. En lo sucesivo se hará hincapié en aquellas constantes más relevantes para cada uno de los cuatro componentes software y de la clase Merge, descritos anteriormente. Estas constantes suponen la implementación de los umbrales antes mencionados, y en el apéndice A se incluyen tablas que aportan datos concretos de todas las constantes utilizadas.

SonarHT

Para el caso de este componente (tabla A.1), las constantes más destacadas son las citadas a continuación:

- SONAR_HALF_WIDE_ANGLE: define el ángulo de apertura del arco de barrido de cada sónar, de lo cual depende el número de objetos potencialmente detectables en un solo barrido.
- SONAR_TANGENT_POINTS: define el número de puntos que se discretizan de cada arco de barrido. Ha de ser un valor equilibrado, dado que si es muy alto aumentará el tiempo de procesamiento y si es muy bajo, habrá menos cantidad de puntos a evaluar y, consecuentemente, menos información evaluada acerca del entorno.
- SONAR_RETURN_THRESHOLD: define la distancia a partir de la cual no se consideran los objetos detectados por los sónars. Si su valor es demasiado bajo, menos información se captará en cada barrido; si por el contrario es demasiado alto, más posibilidades habrá de introducirse ruido -datos erróneos asociados a la precisión de los sensores- entre los datos captados.

- LINES_VOTES_THRESHOLD y POINTS_VOTES_THRESHOLD: define el número de votos mínimo para considerar una línea o punto probable al evaluar las respectivas matrices de la Transformada de Hough. Un valor muy bajo conllevará demasiados "falsos positivos", mientras que un valor muy alto podría ser demasiado restrictivo, anulando líneas y puntos potencialmente existentes.

- LINES_TIME_TO_DECREMENT y POINTS_TIME_TO_DECREMENT: define el tiempo que ha de pasar para considerar que una línea o punto lleva demasiado tiempo sin ser detectada/o nuevamente. Un valor demasiado bajo implicará decrementar votos en líneas o puntos de forma precipitada en más de una ocasión, mientras que un valor muy alto no penalizará -o tardará en penalizar- a líneas y puntos que no han sido nuevamente votadas en un tiempo considerable.

- LINES_PERSISTENCE_DECREMENT y POINTS_PERSISTENCE_DECREMENT: define el número de votos a decrementar en esas líneas o puntos que hace demasiado tiempo que no han sido detectadas/os nuevamente. Un valor demasiado bajo podría suponer que no se marcara la diferencia entre líneas/puntos persistentes y aquellas/os que no se detectaran con frecuencia. Un valor muy alto, por contra, podría eliminar ciertas líneas y puntos potencialmente útiles demasiado pronto.

LaserIEPF

En el caso de este componente (tabla A.2) se podría decir que hay una constante o umbral realmente clave:

- LINE_POINTS_THRESHOLD: define el mínimo número de puntos para considerar una línea en el algoritmo utilizado en la etapa 2 (sección 3.3.2), dentro de las etapas de procesamiento correspondientes al láser. Si hay un número insuficiente de puntos, podrían resultar líneas que realmente no existieran. Por el contrario, si se exigiera demasiados puntos para cada línea, posiblemente se omitirían ciertas líneas válidas con pocos puntos.

SonarHTView

Para el caso de esta vista (tabla A.3), no hay ningún umbral sumamente importante. Sin embargo, la siguiente constante influye de cara a la veracidad de información ofrecida al usuario:

- DEFAULT_REFRESHING_PERIOD: este umbral define cada cuánto tiempo se refrescan los datos en la vista. Si tarda demasiado en ofrecer la información, podría darse una falta de sincronización entre los datos reales y los mostrados por la propia vista, y de ahi que influya en la veracidad de información mostrada al usuario.

SonarView

En el caso de esta vista (tabla A.4) hay varios umbrales que destacan:

- DEFAULT_REFRESHING_PERIOD: como en el caso de la vista SonarHTView, este umbral define cada cuánto tiempo se refrescan los datos en la vista. Si tarda demasiado en ofrecer la información, podría darse una falta de sincronización entre los datos reales y los mostrados por la propia vista, derivando en una falta de veracidad en cuanto a la información mostrada.
- DIRECT_COMMAND_TRANSLATIONAL_SPEED: indica la velocidad de traslación del robot cuando el usuario pulsa los botones del frame "Motion control" (mirar al final de sección 4.2.2). Una velocidad demasiado baja podría suponer un punto en contra para la propia interfaz, y una demasiado alta, por su parte, podría desembocar en una menor capacidad de detección en general por parte del robot.
- DIRECT_COMMAND_ROTATIONAL_SPEED: similar al umbral anterior, pero en este caso referente a la velocidad de rotación del robot. Las implicaciones relacionadas con los valores del umbral son similares.

Merge

Todas las constantes de la tabla A.5 son igualmente relevantes, dada la relación directa entre éstas y el número de líneas y puntos resultantes de la fusión.

Capítulo 5

Conclusiones y trabajo futuro

En este capítulo se abordarán las conclusiones acerca del trabajo realizado, tanto a nivel de consecución de objetivos como de satisfacción de expectativas personales. Por otra parte, se analizarán las líneas de trabajo que pudieran seguir proyectos futuros relacionados con el ámbito de este PFC.

Tras analizar los resultados obtenidos, se puede concluir en que se han alcanzado los objetivos específicos previstos de este PFC:

- Se ha estudiado la necesidad e importancia del uso de técnicas de modelado adecuadas para interpretar los datos extraídos de los sensores ultrasónicos.
- Se ha analizado la técnica de la Transformada de Hough tanto teórica como matemáticamente, desglosando su forma de proceder en diferentes etapas para una mayor organización y entendimiento. Dichas etapas han sido descritas de la forma más intuitiva y transparente posible, con tal de facilitar una implementación adecuada.
- Se ha aplicado la metodología de desarrollo seleccionada para crear el prototipo en Matlab. Éste ha facilitado la posterior implementación del prototipo final en CoolBOT, tanto por la familiarización con el marco del problema como por el planteamiento de cálculos como el del *Punto q* y la discretización para la inserción en matriz, entre otros.
- Se ha estudiado el framework CoolBOT, tanto desde una perspectiva genérica como a nivel de generación e implementación de los diferentes componentes software del bundle, de paquetes de puertos y de integraciones.
- Se ha aplicado la metodología de desarrollo seleccionada para crear el prototipo final en CoolBOT. La implementación ha supuesto adaptarse a la sintaxis concreta [Domínguez-Brito, 2011a] (reglas de nomenclatura y comentarios en inglés al tratarse de un framework de código abierto) y aprender a utilizar tanto librerías como recursos -por ejemplo, otros bundles- inherentes a CoolBOT y a las vistas propiamente dichas -librería Gtk+ 2.0 [GTK+ 2.0 Web Page]-.

En lo que refiere a los objetivos académicos, cabe una perspectiva subjetiva y en primera persona, puesto que se trata de evaluar sobre qué ámbitos se ha aprendido y en qué medida.

En resumidas cuentas, mis conocimientos eran genéricos y poco especializados, de forma que he aprendido muchos detalles durante el desarrollo de este PFC al tener que ahondar en las diferentes materias que ocupa. Por dar ejemplos, mis conocimientos de robótica se ceñían en la práctica a los Lego NXT, y en C++ nunca había trabajado con una distribución tan grande de ficheros, que a su vez tenían en su mayoría un tamaño considerable y ciertas reglas sintácticas a seguir. He aprendido la gran utilidad de los Sistemas de Control de Versiones como el Git (y de comandos concretos como *diff*), de herramientas como LaTeX-con un gran resultado pese a la curva de aprendizaje que exige- y de la relevancia de seguir una buena organización -derivada de una metodología de software bien seleccionada- cuando se trata de un proyecto de estas dimensiones, sumamente alejado de lo realizado en prácticas durante la carrera por la finitud de las materias.

Desde un punto de vista más formal u objetivo, se han llevado a cabo exitosamente las siguientes tareas:

- Seleccionar y utilizar una metodología de desarrollo de software adecuada.
- Aplicar conocimientos de programación en lenguaje *M* para Matlab y en *C++*.
- Aplicar conocimientos matemáticos, sobre todo a nivel de trigonometría y cálculo vectorial.
- Aplicar conocimientos acerca de *GUI* (Interfaces Gráficas de Usuario).
- Aplicar conocimientos sobre Sistema Operativo Linux.
- Aplicar conocimientos relacionados con sistemas robóticos.
- Manejar Sistemas de Control de Versiones.
- Manejar herramientas para generación de documentación.

Atendiendo a cómo ha concluido este PFC y en relación al contexto que ocupa, se presentan a continuación diferentes propuestas de posibles líneas de desarrollo a abordar en un futuro:

- Como se señalaba en el capítulo anterior, las pruebas realizadas siguen el método empírico y tienen como fin ajustar los diferentes parámetros, con tal de lograr la mejor configuración posible para los entornos donde se han realizado las pruebas de los prototipos implementados. Por ende, en un futuro se podrían crear baterías de pruebas cuyo objeto fuera comparar formalmente el método de la Transformada de Hough con otras técnicas de modelado para sensores ultrasónicos -como RANSAC [Fischler and Bolles,1981] [Hartley and Zisserman,2000]-, es decir, más allá de pruebas destinadas al propio ajuste de este método.
- Aunque no era un objetivo previsto para este PFC, en la vista *SonarView* se pueden apreciar las diferencias entre las líneas captadas por el láser y las líneas más filtradas de los sónars, pero no a nivel analítico. Por este motivo se podría realizar un estudio exhaustivo y práctico comparando, mediante baterías de pruebas formales, los resultados del láser y de los sónars para diferentes tipos de entorno y texturas de objetos.

Parte I

Apéndices

Apéndice A

Tablas de constantes

A continuación se detallan las constantes utilizadas durante la implementación del bundle **coolbot-sonar-bundle**.

Cuadro A.1: Tabla de constantes de SonarHT

Constante	**Valor actual**	**Tipo**	**Descripción**
SONAR_HALF_WIDE_ANGLE	15 grados	Pública	Ángulo de apertura de cada mitad del arco de percepción con respecto al centro
SONAR_TANGENT_POINTS	75	Pública	Número de puntos discretizados a evaluar en cada arco de percepción
RO_CENTIMETERS_PER_CELL	3 cm.	Pública	Centímetros que acoge cada celda al discretizar ρ para la matriz de la Transformada de Hough
THETA_DEGREES_PER_CELL	3 grados	Pública	Grados que acoge cada celda al discretizar θ para la matriz de la Transformada de Hough
RO_MAX	8 m.	Pública	Límite del valor Ro para la matriz de la Transformada de Hough, cuyo exceso lleva a una traslación de ésta
SONAR_RETURN_THRESHOLD	2000 mm.	Pública	Límite de percepción de sensores, a partir del cual los retornos no son considerados
LINES_VOTES_THRESHOLD	275	Pública	Mínimo número de votos en la matriz de la Transformada de Hough para considerar una línea como probable
POINTS_VOTES_THRESHOLD	350	Pública	Mínimo número de votos en la matriz de la Transformada de Hough para considerar un punto como probable
LINES_TIME_TO_DECREMENT	1000 ms.	Pública	Tiempo que habrá de pasar sin que una línea sea votada para aplicar un decremento en sus votos
POINTS_TIME_TO_DECREMENT	1000 ms.	Pública	Tiempo que habrá de pasar sin que un punto sea votado para aplicar un decremento en sus votos
LINES_PERSISTENCE_DECREMENT	7	Pública	Número de votos a decrementar en línea si supera el umbral LINES_TIME_TO_DECREMENT
POINTS_PERSISTENCE_DECREMENT	10	Pública	Número de votos a decrementar en punto si supera el umbral POINTS_TIME_TO_DECREMENT
CROSS_RADIUS	2	Pública	Número de celdas en horizontal y vertical para dibujar cruces en matriz de Transformada de Hough
PLH_MINIMUM_DISTANCE	5 cm.	Pública	Mínima distancia para considerar un conflicto entre líneas y puntos en PLH
PLH_LINE_X_DISTRIBUTION	20	Pública	En PLH, mínimo valor de desviación típica en X para considerar válida una línea
PLH_LINE_Y_DISTRIBUTION	10	Pública	En PLH, máximo valor de desviación típica en Y para considerar válida una línea
DEFAULT_PERIOD	200 ms.	Privada	Período por defecto de ejecución de función _theMainTimer_
DEFAULT_PERSISTENCE_PERIOD	1000 ms.	Privada	Período por defecto de ejecución de función _theMainPERSISTENCE_TIMER_
CIRCULAR_FIFO_LENGTH	100	Privada	Longitud de la cola donde se insertan los valores odométricos para la interpolación posterior

Cuadro A.2: Tabla de constantes de LaserIEPF

Constante	**Valor actual**	**Tipo**	**Descripción**
LINE_POINTS_THRESHOLD	10	Pública	Mínimo número de puntos para considerar una línea
DEFAULT_PERIOD	200 ms.	Privada	Período por defecto de ejecución de función _theMainTimer_
CIRCULAR_FIFO_LENGTH	100	Privada	Longitud de la cola donde se insertan los valores odométricos para la interpolación posterior

Cuadro A.3: Tabla de constantes de SonarHTView

Constante	**Valor actual**	**Tipo**	**Descripción**
DEFAULT_REFRESHING_PERIOD	300 ms.	Privada	Período de refresco por defecto de los elementos de la vista en pantalla
DEFAULT_SIZE_X	500 pixels	Privada	Tamaño en horizontal por defecto de ventana de vista
DEFAULT_SIZE_Y	550 pixels	Privada	Tamaño en vertical por defecto de ventana de vista
DRAW_AREA_XY_DIVS	4	Privada	Número de divisiones en imágenes de las matrices de la Transformada de Hough en esta vista
ARROW_WIDTH	6 pixels	Privada	Parámetro para dibujar las flechas en la vista
ARROW_DEEP	6 pixels	Privada	Parámetro para dibujar las flechas en la vista
FONT_SIZE	12	Privada	Tamaño de fuente dibujada en la vista

Cuadro A.4: Tabla de constantes de SonarView

Constante	**Valor actual**	**Tipo**	**Descripción**
FIFO_LENGTH	5 paquetes	Privada	Longitud de la cola para los paquetes de entrada con puerto tipo FIFO
DEFAULT_REFRESHING_PERIOD	250 ms.	Privada	Período de refresco por defecto de los elementos de la vista en pantalla
DRAW_AREA_MARGIN	3 pixels	Privada	Margen establecido por defecto para área de dibujo en ventana de vista
DRAW_AREA_WIDTH	300 pixels	Privada	Tamaño en horizontal por defecto de área de dibujo en ventana de vista
DRAW_AREA_HEIGTH	560 pixels	Privada	Tamaño en vertical por defecto de área de dibujo ventana de vista
DRAW_AREA_XY_DIVS	8	Privada	Número de divisiones de mapa representado en esta vista
GRID_AXES_FONT_SIZE	8	Privada	Tamaño de fuente para caracteres de área de dibujo en ventana de vista
ARROW_WIDTH	6 pixels	Privada	Parámetro para dibujar las flechas en la vista
ARROW_DEEP	6 pixels	Privada	Parámetro para dibujar las flechas en la vista
ARROW_WIDTH_AUX	20 pixels	Privada	Parámetro para dibujar flechas especiales en la vista
ARROW_DEEP_AUX	20 pixels	Privada	Parámetro para dibujar flechas especiales en la vista
HALF_ARC_ANGLE	5 grados	Privada	Parámetro para dibujar los arcos de los sensores ultrasónicos del robot
DIRECT_COMMAND_TRANSLATIONAL_SPEED	250 mm/s	Privada	Indica velocidad de traslación cuando se oprime botón correspondiente en interfaz
DIRECT_COMMAND_ROTATIONAL_SPEED	20 grados/s	Privada	Indica velocidad de rotación cuando se oprime botón correspondiente en interfaz
PERCEPTION_RADIUS	2 m.	Privada	Radio del círculo que representa el límite de percepción permisible para el robot
CUTTING_RADIUS	8 m.	Privada	Radio del círculo de corte de las líneas en el mapa del área de dibujo
MINIMUM_RANGE_SCALE_VALUE	1 m.	Privada	Extensión mínima del mapa en el área de dibujo
MAXIMUM_RANGE_SCALE_VALUE	LASER_MAX_RANGE/1000	Privada	Extensión máxima del mapa en el área de dibujo

Cuadro A.5: Tabla de constantes de clase Merge

Constante	**Valor actual**	**Tipo**	**Descripción**
DEFAULT_MINIMUM_LINES_THRESHOLD	5	Pública	Mínimo número de líneas por defecto para poder fusionar
DEFAULT_MINIMUM_POINTS_THRESHOLD	7	Pública	Mínimo número de puntos por defecto para poder fusionar
DEFAULT_LINES_RO_THRESHOLD	5	Pública	Máxima diferencia de parámetro ρ entre líneas para considerarlas potencialmente fusionables
DEFAULT_LINES_THETA_THRESHOLD	5	Pública	Máxima diferencia de parámetro θ entre líneas para considerarlas potencialmente fusionables
DEFAULT_POINTS_RO_THRESHOLD	3	Pública	Máxima diferencia de parámetro ρ entre puntos para considerarlos potencialmente fusionables
DEFAULT_POINTS_THETA_THRESHOLD	3	Pública	Máxima diferencia de parámetro θ entre puntos para considerarlos potencialmente fusionables
DEFAULT_MIDPOINT_MAX_DISTANCE	20	Pública	Distancia máxima entre punto medio de lista actual y punto que se está evaluando

Bibliografía

[George,2001] George T. Heineman; William T. Councill *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley Professional, Reading 2001

[Jacobson, Booch and Rumbaugh, 2000] Jacobson, Ivar, Booch Grady and Rumbaugh James *El proceso unificado de desarrollo de software.* Addison-Wesley, 2000.

[Lamport, 1986] Lamport, Leslie *LaTex : A document Preparation System.* Addison-Wesley, 1986.

[Roland, 1983] Rolland, Christian *LaTex guide pratique.* Addison-Wesley, 1993.

[GTK+ 2.0 Web Page] GTK+ 2.0 Tutorial: *http://library.gnome.org/devel/gtk-tutorial/stable/c24.html.*

[Stroustrup,1997] Stroustrup, B. The C++ programming language: Special Edition. *Addison Wesley, 3ª edición, 1997.*

[Vandevoorde,2003] Vandevoorde, D; Josuttis, N. M. C++ Templates: the complete guide. *Addison Wesley, 2003.*

[Player-Stage] The Player Project. http://playerstage.sourceforge.net.

[Minguez,2004] Minguez, J.; Montano, L. Nearness Diagram Navigation (ND): Collision Avoidance in Troublesome Scenarios IEEE Transactions on Robotics and Automation, Volume 20, Issue 1, Pages:45 - 59, 2004.

[Arvind,1981] Arvind, Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. *Proceedings of the 8th annual symposium on Computer Architecture, p.291-302, May 12-14, 1981, Minneapolis, Minnesota, United States*

[Latombe,1991] . Latombe, Jean-Cloude. Robot-motion planning, The Kluwer International Series in Engineering and Computer Science, Kluwer Academic.

[Asada,1986] Asada, Haruhiko; Slotine, Jean-Jacques E. "Robot Analysis and Control".

[Domínguez-Brito, 2012a] A. C. Domínguez-Brito. "CoolBOT Starting Guide", www.coolbotproject.org.

[Domínguez-Brito, 2011a] A. C. Domínguez-Brito. "CoolBOT: C++ Coding Programming Conventions", www.coolbotproject.org.

[Domínguez-Brito,2011b] A. C. Domínguez-Brito, F. J. Santana-Jorge, S. Santana-de-la-Fe, J. M., Martínez-García, J. Cabrera-Gámez, J. D. Hernández-Sosa, J. Isern-González and E. Fernández-Perdomo. "CoolBOT: An Open Source Distributed Component Based Programming Framework for Robotics" International Symposium on Distributed Computing and Artificial Intelligence (DCAI'2011), Salamanca, 6th - 8th April 2011.

[Domínguez-Brito,2012b] A. C. Domínguez-Brito, J. Cabrera-Gámez, J. D. Hernández-Sosa, J. Isern-González and E. Fernández-Perdomo. "An Approach to Distributed Component-Based Software for Robotics" Robotic Systems - Applications, Control and Programming, Ashish Dutta (Ed.), ISBN: 978-953-307-941-7, InTech. February 2012

[Domínguez-Brito,2012c] A. C. Domínguez-Brito, F. J. Santana-Jorge, J. Cabrera-Gámez, J. D. Hernández-Sosa, J. Isern-González and E. Fernández-Perdomo. "Exploring Interfaces in a Distributed Component-Based Programming Framework for Robotics" 4th International Conference on Agents and Artificial Intelligence, Special Session on Intelligent Robotics (ICAART-SSIR'2012), February 2012, Vilamoura (Algarve), Portugal.

[Fernández-Perdomo,2009] Fernández-Perdomo, Enrique. "Test and Evaluation of the Fast-SLAM Algorithm in a Mobile Robot", A Thesis Submitted for the Degree of MSc. Sistemas Inteligentes y Aplicaciones Numéricas en Ingeniería (SIANI)

[Burguera,2009] Burguera, Antoni. "A Contribution to Mobile Robot Localization Using Sonar Sensors" (Dec-2009), Phd Dissertation, Univertitat de les Illes Balears.

[Censi,2005] A. Censi, L. Iocchi, G. Grisetti. "Scan Matching in the Hough Domain", International Conference on Robotics and Automation 2005 (ICRA 2005), Barcelona, Spain.

[Tardós,2002] J. D. Tardós, J. Neira, P. M. Newman and J. J. Leonard, Robust Mapping and Localization in Indoor Environments Using Sonar Data", The International Journal of Robotics Research, 21, 4, 311-330, April, 2002.

[Fischler and Bolles,1981] Fischler, M.A. and Bolles, R.C. Random sample consensus: a paradigm form model fitting with applications to image analysis and automated cartography, Comm. Asocc. Comp. Mach. 24(6), 381-395

[Hartley and Zisserman,2000] Hartley, R. and Zisserman, A. Multiple View Geometry in Computer Vision, Cambridge University Press, Cambridge, U.K.

[Ballard and Brown,1982] Ballard, D.H. and Brown C.M. Computer Vision, Prentice Hall, Englewood Cliffs, N.J.

[Illinworth and Kittler,1988] A survey of the Hough Transform, Computer Vision, Graphics and Image Processing 44, 87-116.