UNIVERSIDAD DE EXTREMADURA

Escuela Politécnica
I. T. Telecomunicación. Sonido e Imagen

Proyecto Fin de Carrera

“Scan Matching Algorithms For Robot Localization Using an UTM-30LX Laser Range Sensor”

Carlos Municio Serrano
Junio, 2010

UNIVERSIDAD DE EXTREMADURA
Escuela Politécnica
I. T. Telecomunicación. Sonido e Imagen

Proyecto fin de carrera

“Scan Matching Algorithms For Robot Localization Using an UTM-30LX Laser Range Sensor”

Autor: Carlos Municio Serrano
Fdo.:
Director: Pedro M. Núñez Trujillo
Fdo.:

Tribunal Calificador

Presidente:
Fdo.:

Secretario:
Fdo.:

Vocal:
Fdo.:

CALIFICACIÓN:
FECHA:

Agradecimientos

Quiero expresar mi agradecimiento a Pedro Núñez, tutor de este Proyecto Fin de Carrera, por toda la ayuda, dedicación y consejo que me ha prestado a lo largo de estos cinco meses, destacando especialmente la disponibilidad y amabilidad con las que siempre me ha tratado.

También quiero agradecer especialmente la colaboración y ayuda prestada por todas las personas que están a diario en Robolab, ya que sin ellas no habría sido posible este proyecto.

A todos mis amigos, porque siempre han estado ahí interesándose por cómo lo llevaba y por hacer más ameno el camino.

Y por último y de forma muy especial, a mis padres, a mi hermano y a Alba, por ser las personas más importantes en mi vida y por el total apoyo que me han prestado.

Muchas gracias a todos,

Municio.

Resumen

En los últimos años, la inteligencia artificial ha experimentado un gran desarrollo de forma exponencial gracias, principalmente, a los avances tecnológicos y a su expansión en muchos ámbitos de la vida cotidiana. Este progreso vertiginoso, hace que se esté investigando continuamente en cada uno de sus campos como son la visión artificial, la robótica, la realidad aumentada...

El empleo de la robótica en entornos tanto domésticos como industriales es una práctica cada vez más generalizada. Para la realización de tareas, va a ser necesario que el robot efectúe un desplazamiento, y es justo aquí, donde entra la necesidad, del mismo robot, de conocer la posición en la que está situado en su entorno de la forma más precisa posible; es lo que se refleja en el título del proyecto como *self-localization*.

El presente Proyecto Fin de Carrera consiste en el desarrollo de una aplicación que permita la estimación de la posición global de un robot móvil en el entorno que le rodea en tiempo real.

La forma más sencilla de realizar esta tarea es utilizando para ello la odometría; pero ésta tiene algunos defectos que veremos más adelante. Nosotros vamos a afrontar la auto-localización del robot mediante s*can matching*, método que consiste en la comparación de los scans procedentes de un láser, y que permite estimar la posición del robot mediante la correspondencia entre un scan actual y otro de referencia.

Para ello se van a implementar tres algoritmos de emparejamiento de scans, que se detallan minuciosamente en los capítulos venideros. Asimismo, se implementará una interfaz gráfica que mostrará la estimación de la posición que ofrece cada algoritmo, así como la que ofrece la odometría.

La memoria del proyecto, indica los problemas afrontados y cómo han sido resueltos, mostrando los resultados obtenidos mediante los experimentos e introduciendo las plataformas hardware y software utilizadas, RobEx y RoboComp, respectivamente.

El código del proyecto fin de carrera que conforma el nuevo componente creado, se ha incorporado a RoboComp (repositorio de componentes software para la robótica).

Índice de Contenidos

Índice de Figuras

Capítulo 1: Introducción

La Robótica es una rama de la tecnología, que estudia el diseño y construcción de máquinas capaces de desempeñar tareas repetitivas o peligrosas para el ser humano.

Su historia ha estado unida a la construcción de "artefactos", tratando de materializar el deseo humano de crear seres semejantes a él, que realizasen el trabajo duro o le facilitaran el mismo.

El término robot fue empleado por primera vez por el dramaturgo checo Karel Capek en 1921 [1], en su obra de teatro Rossum's Universal Robots a partir de la palabra checa *Robota*, cuyo significado es servidumbre o trabajo forzado. En este relato, un brillante científico llamado Rossum y su hijo comienzan a construir robots para que sirvan a la raza humana y realicen todos los trabajos físicos. Continúan mejorando estos robots hasta que son capaces de desarrollar un ser perfecto. Finalmente estos seres perfectos dejan de cumplir su papel original de servidumbre y se rebelan contra sus dueños, llegando prácticamente a destruir la raza humana.

Años más tarde, sería el escritor bioquímico estadounidense de origen ruso, Isaac Asimov, quién acuñaría el vocablo robótica para describir a la ciencia que estudia a estos seres.

Fue Isaac Asimov el que enunció las **tres leyes de la robótica**, que aunque de gran simplicidad y escaso criterio científico, siguen vigentes hoy en día como ética de la robótica. Son tres normas escritas que establecen lo siguiente:

1. Un robot no debe dañar a un ser humano o, por su inacción, dejar que un ser humano sufra daño.

2. Un robot debe obedecer las órdenes que le son dadas por un ser humano, excepto si estas órdenes entran en conflicto con la Primera Ley.

3. Un robot debe proteger su propia existencia, hasta donde esta protección no entre en conflicto con la Primera o la Segunda Ley.

Atendiendo de forma breve a la cronología, podemos destacar que durante la segunda mitad del siglo XX proliferaron las industrias basadas en la ciencia y, gracias a ellas y a las mejoras tecnológicas en la electrónica, se hizo posible la aparición del ordenador, el cual revolucionó el modo de procesamiento y comunicación de la información.

A principios de los años 60, se empleó el primer robot industrial de la historia en la factoría General Motors en Nueva Jersey. A partir de entonces, y especialmente a partir de 1980, el llamado primer año de la era robótica, se empezó a extender el uso y desarrollo de robots en los diferentes tipos de industrias para realizar pesados y repetitivos trabajos físicos.

Con la introducción de los microprocesadores en los años 70, los modernos ordenadores han ejercido como cerebro de los robots mecánicos. Esta fusión hizo posible la aparición de los robots autónomos.

1.1 APLICACIONES DE LA ROBÓTICA AUTÓNOMA

A diferencia de los robots fijos industriales, los robots autónomos móviles pueden realizar tareas en un gran número de entornos distintos, como pueden ser el ocio, la educación, bienestar personal y social, etc. Son los llamados "*robots de servicio*": robots adecuados para el trabajo en áreas peligrosas para los seres humanos, como puede ser la desactivación de explosivos o la exposición en los entornos contaminados radioactiva o químicamente; y para explorar lugares inaccesibles, como pueden ser excavaciones mineras de gran profundidad o misiones espaciales no tripuladas.

Figura 1: Robot en misión espacial en Marte

Para que un robot realice correctamente todas esas funciones ha de estar equipado con diferentes sensores que adquieran información del entorno donde se encuentra, como son la visión o el láser. La visión artificial o por computador tiene como finalidad reproducir artificialmente el sentido de la vista mediante el procesamiento e interpretación de imágenes. Hay que utilizar para ello las prestaciones de los ordenadores, constituyendo ésta una tarea de gran complejidad para los computadores.

El uso de sensores de rango en robótica tiene una serie de ventajas. En primer lugar, un láser – ya sea de dos dimensiones, como los que se han usado en el presente trabajo, o tridimensional - tiene como característica una alta resolución, acompañado también de una alta precisión. Ello permite utilizarlos de una forma fiable en la navegación de robots; por ejemplo, en entornos dinámicos, es decir, con presencia de seres humanos u otros robots, o en tareas más complejas como la construcción de mapas (*mapping*) o en la localización del robot en el entorno.

Por su parte, la visión artificial es una gran herramienta para establecer la relación entre el mundo tridimensional y sus vistas bidimensionales. Además, nos permite conocer el entorno que visualiza el robot, permitiéndole a él mismo conocer su posición global y, de este modo, poderse desplazar sin que haya un desconocimiento de su nueva ubicación.

La tendencia futura en centra en la fusión de los datos procedentes de varios sensores. Ello permitiría aprovechar las ventajas de cada uno, así como limitar sus inconvenientes.

En cuanto a las aplicaciones reales de la robótica autónoma, la industrialización y automatización de los almacenes es un recurso cada vez más utilizado. Las funciones que se desarrollan en estos entornos pueden englobarse en dos: el almacenaje propiamente dicho y el manejo de cargas.

Para conseguir estos fines se ha desarrollado un creciente apoyo tecnológico a las actividades de almacenaje a partir del empleo de equipos cada vez más sofisticados y automatizados.

El control, supervisión y gestión de este tipo de equipos puede suponer una tarea muy compleja. Existen numerosos tipos de robots encargados de colocar y recuperar mercancías con los que se optimiza el uso del espacio disponible

dentro de los almacenes (véase *figura* 2); ganando, además, seguridad y velocidad en el tratamiento de las mismas.

Figura 2: Robots industriales para almacenaje

1.2 ESTUDIO DEL PROBLEMA

Se está trabajando mucho en el campo de la robótica y de la visión artificial, encaminándose muchas de las investigaciones a que, gracias a esta última, un robot móvil pueda estimar la posición global en la que se encuentra de una forma bastante precisa.

Con ello se alberga la posibilidad de que un robot autónomo se desplace por su entorno y sepa en todo momento la posición en la que se encuentra. Esto tiene una importancia máxima, ya que para llevar a cabo cualquier orden (ya sea un desplazamiento hasta un lugar o una simple rotación sobre sí mismo) es imprescindible conocer la posición en la que está ubicado en su entorno y la nueva localización que tendrá cuando finalice su trayectoria.

Esto está comenzando ya a tener aplicación en hospitales, museos, industrias (ver *figura* 3)... para distribución y almacenaje, vigilancia de una determinada zona, desplazamientos hasta un punto en concreto y regresar; o en la optimización del espacio de los almacenes, así como en la gestión automática de los mismos.

Figura 3: Robot enfermero *RIBA*, en Japón.

Estos sistemas integran diferentes mecanismos robóticos interconectados entre sí y con ordenadores, siendo cada vez sistemas más robustos, rápidos e inteligentes.

La forma más sencilla de estimar la posición de los robots móviles venía siendo mediante la odometría, es decir, utilizando la información sobre la rotación de las ruedas para estimar cambios en la posición a lo largo del tiempo. Sin embargo, ésta tiene algunos defectos que vamos a ver a continuación.

Como modelo de robot móvil se utilizará la plataforma robótica RobEx, que va a ser explicada en detalle en el capítulo siguiente.

1.2.1 Problema de la Odometría

Se denomina odometría al estudio de la estimación de la posición de vehículos con ruedas durante la navegación. Para realizar esta estimación se usa información sobre la rotación de las ruedas a lo largo del tiempo. Este término también suele usarse para referirse a la distancia que se ha recorrido. La palabra en sí, se compone de las palabras griegas *hodos* (trayecto) y *metron* (medida).

- Odometría en robótica

Los robots móviles usan la odometría para estimar (y no determinar) su posición relativa a su localización inicial. Es bien sabido que la odometría proporciona una buena precisión a corto plazo, es barata de implantar, y permite tasas de muestreo muy altas. Sin embargo la idea fundamental de la odometría es la integración de información incremental del movimiento a lo largo del tiempo, lo cual conlleva una inevitable acumulación de errores. En concreto, la acumulación de errores de orientación, causa grandes errores en la estimación de la posición, los cuales van aumentando proporcionalmente con la distancia recorrida por el robot.

A pesar de estas limitaciones, muchos investigadores están de acuerdo en que la odometría es una parte importante del sistema de navegación de un robot, y que debe usarse con medidas del posicionamiento absolutas para proporcionar una estimación de la posición más fiable.

La odometría se basa en ecuaciones simples que se pueden implementar fácilmente y que utilizan datos de encoders situados en las ruedas del robot. Sin embargo, la odometría también está basada en la suposición de que las revoluciones de las ruedas pueden ser traducidas en un desplazamiento lineal relativo al suelo. Esta suposición no tiene una validez absoluta.

Un ejemplo extremo es cuando las ruedas patinan: si por ejemplo, un rueda patina sobre una mancha de aceite y la otra no, entonces se registrarán revoluciones en la rueda, a pesar de que éstas no correspondan a un desplazamiento lineal de la misma. Además de este ejemplo, hay muchas otras razones más sutiles por las cuales se pueden producir imprecisiones en la traducción de las lecturas del encoder de la rueda.

Todos estos errores se pueden agrupar en dos categorías: errores sistemáticos y errores no sistemáticos.

a) Errores sistemáticos

Entre los que cabe destacar:

- Incertidumbre en el diámetro de las ruedas.
- Mal alineamiento de las ruedas.
- Incertidumbre en la distancia entre los puntos de apoyo de ruedas.

b) Errores no sistemáticos

Se pueden clasificar en los siguientes:

i) Desplazamiento en suelos desnivelados.

ii) Desplazamiento sobre objetos inesperados que se encuentren en el suelo.

iii) Patinaje de las ruedas debido a:

- Suelos resbaladizos.
- Sobre-aceleración.
- Derrapes (debidos a una rotación excesivamente rápida).
- Fuerzas externas (interacción con cuerpos externos).

Una clara distinción entre errores sistemáticos y no sistemáticos es de gran importancia a la hora de reducir los errores en la odometría. Por ejemplo, los errores sistemáticos se pueden corregir con una buena calibración; pero, en superficies no rugosas de entornos interiores, son éstos los que contribuyen mucho más a los errores en la odometría que los no sistemáticos. Sin embargo, en superficies abruptas con irregularidades significativas, son los errores no sistemáticos los que predominan.

El problema de los errores no sistemáticos es que pueden aparecer inesperadamente (por ejemplo cuando el robot pasa por encima de un objeto que se encuentra en el suelo, como un cable) y pueden causar errores muy grandes en la estimación de la posición.

En la determinación de la incertidumbre que se produce a la hora de estimar la posición de un robot que utiliza odometría, cada posición calculada está rodeada por una *elipse de error* característica, la cual indica la región de incertidumbre para la posición actual del robot, como se puede apreciar en la siguiente figura:

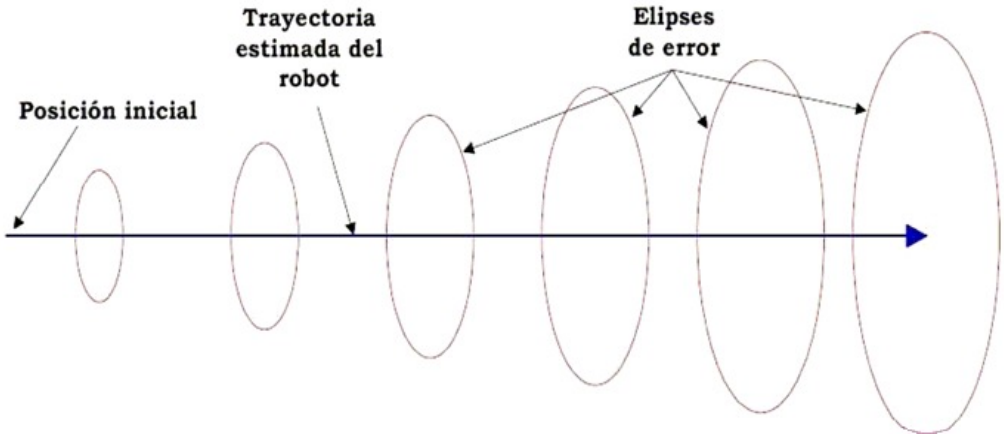

Figura 4: *Elipses de error* en torno a la trayectoria estimada de un robot

Estas elipses crecen a medida que la distancia recorrida aumenta, a no ser que un sistema de estimación absoluto ponga a cero su tamaño. Estas técnicas de estimación del error se basan en parámetros derivados de los errores sistemáticos, puesto que la magnitud de los no sistemáticos es siempre impredecible.

1.3 ADAPTACIÓN AL ENTORNO DEL LABORATORIO

Como plataforma de experimentación se va a disponer de un robot de la clase RobEx, que se explicará en detalle en el capítulo siguiente.

1.3.1 Plataforma robótica utilizada

Podemos avanzar que RobEx, robot del tipo *differential three wheeled*, es un robot móvil de tipo diferencial, es decir, el movimiento del robot depende únicamente de la velocidad de rotación de sus dos ruedas motrices separadas que se colocan a ambos lados del cuerpo del robot.

Por lo tanto, puede cambiar su dirección mediante la variación de la tasa de rotación de sus ruedas, sin requerir un movimiento de dirección adicional. Además de las ruedas motrices, el robot dispone de una tercera rueda, de giro libre, que sirve de apoyo.

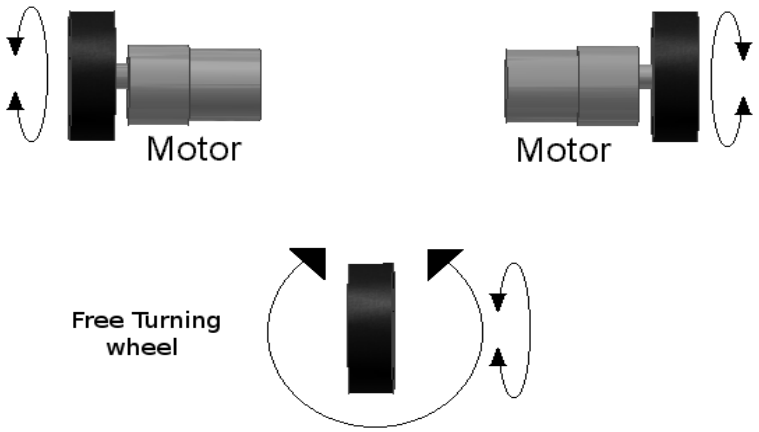

Figura 5: Mecánica de motricidad de RobEx

RobEx está diseñado para llevar uno o varios ordenadores portátiles a bordo para poder realizar los procesos complejos que se requieran. Su principal orientación es la investigación, aunque muy pronto veremos a estos robots realizando tareas en entornos diversos, como puede ser un museo, un instituto o un hospital.

A continuación se muestra una imagen de uno de los robots utilizados en nuestro proyecto y que cuenta con todas las características anteriormente mencionadas, a parte de otras características adicionales que se detallarán en el siguiente capítulo:

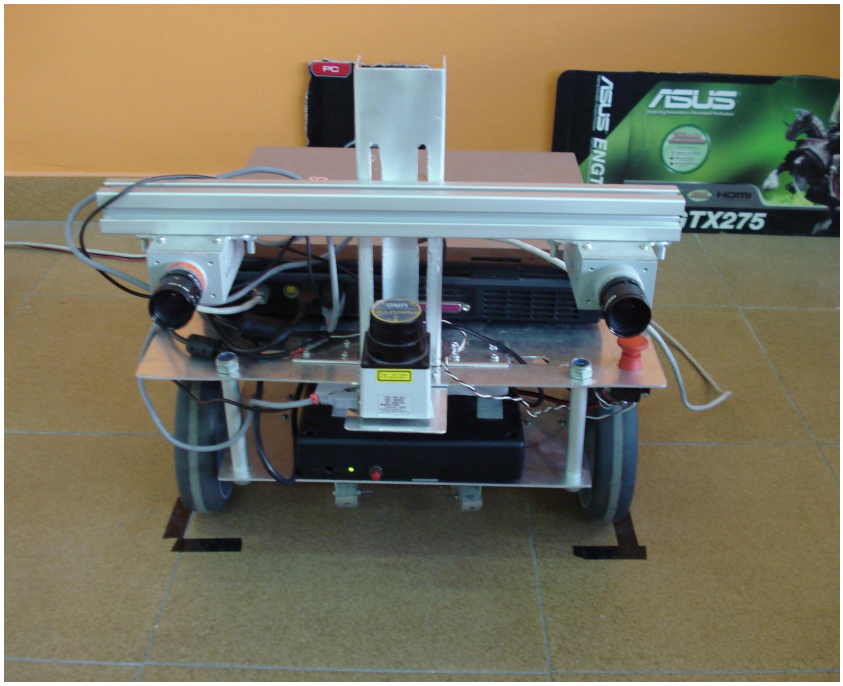

Figura 6: Robot móvil diferencial *RobEx, bender*.

1.3.2 Entorno de trabajo

El entorno por el que va a moverse el robot está ubicado concretamente en la sala del Laboratorio de Robótica de la Universidad de Extremadura (Robolab)

Es un espacio constituido principalmente por paredes poligonales y en el que se han introducido diferentes objetos que actúan como obstáculos y para dar una sensación de ser un entorno de la vida real.

Es en esta sala donde se han realizado los experimentos y a continuación se muestra una fotografía de la misma:

Figura 7: Entorno de trabajo

1.4 OBJETIVOS

El objetivo que persigue el Proyecto Final de Carrera descrito en este documento es el desarrollo de un sistema que permita la estimación de la posición de un robot móvil en tiempo real, basándonos en algoritmos de *scan matching*, que se explicarán detenidamente en los capítulos siguientes.

El objetivo fundamental del proyecto es implementar un sistema robusto capaz de mostrarnos por pantalla las estimaciones de la trayectoria recorrida por el robot, basándose en cada uno de los tres algoritmos que implementaremos y, por otra parte, la estimación de la posición basada en la odometría. No obstante, este objetivo global puede subdividirse en tareas más pequeñas para simplificar la acometida de la estructura completa, como son:

a) Implementar las funciones correspondientes que ejecuten los algoritmos de *scan matching*.

b) Conseguir obtener una buena estimación acerca de la posición real del robot utilizando la información extraída de los scans.

c) Implementar una interfaz gráfica, mediante la herramienta Qt4 designer, que sea intuitiva y sencilla de manejar y que muestre por pantalla los resultados, en tiempo real.

d) Crear un sistema de planificación para conseguir una elaboración estructurada por etapas.

Además, estos objetivos han de ser satisfechos intentando cumplir las siguientes restricciones:

i) La implementación ha de ser independiente del entorno; no tiene sentido limitar la funcionalidad del sistema a entornos controlados como el laboratorio, sino que ha de abordarse el problema desde una perspectiva más amplia.

ii) Se debe diseñar teniendo siempre presente el modelo de programación basado en componentes, de forma que los nuevos componentes diseñados puedan interactuar con los ya existentes.

iii) Aunque el sistema se vaya a implementar y probar sobre RobEx y la plataforma RoboComp (comentados en posteriores capítulos) se ha de intentar generalizar los algoritmos de forma que puedan ser usados sobre otras plataformas.

iv) Debe funcionar en tiempo real. Todos los cálculos y procesos necesarios se ejecutarán a medida que sean requeridos.

v) El sistema debe ser lo suficientemente robusto como para adaptarse a cambios en el entorno.

1.5 ESTRUCTURA DEL DOCUMENTO

La estructura de este documento se divide en una serie de capítulos bien diferenciados, para facilitar el entendimiento y seguimiento del trabajo realizado a lo largo del proyecto. A continuación se muestra una pequeña descripción de cada uno de ellos.

La memoria del presente Proyecto Fin de Carrera va a ser desarrollada concretamente en seis capítulos y un anexo.

Capítulo 1 – Introducción: En esta parte del documento se hace un breve prolegómeno acerca de la robótica y de la visión artificial, comentando sus principales aplicaciones en la actualidad. Se hace una descripción del problema que se pretende resolver y se establecen los objetivos que se pretenden conseguir. Es el capítulo en el que se encuentra el lector.

Capítulo 2 – RobEx y RoboComp: En este apartado se va a proporcionar una toma de contacto con el robot sobre el que se realizarán las pruebas y con la arquitectura de componentes, estructura a seguir en la implementación de la solución.

Capítulo 3 – Descripción del sistema: En este capítulo se aborda de una forma general el conjunto de tareas necesarias para conseguir el funcionamiento de nuestro sistema, tal y como se ha planteado. También se describen los algoritmos empleados desde un punto de vista funcional. Es la implementación de estos algoritmos los que ha supuesto la mayor carga de trabajo.

Capítulo 4 – Diseño e Implementación: En este capítulo se detalla la estructura de componentes que se propone como solución al sistema, se comentan los componentes preexistentes a utilizar y se establecen las funcionalidades del sistema basándonos en el nuevo componente creado y en su implementación mediante programación en C++. También se muestra el diseño del interfaz gráfico mediante la herramienta Qt4 designer.

Capítulo 5 – Experimentos y Resultados: En este capítulo se comentan los principales experimentos llevados a cabo junto con los resultados obtenidos y se muestran visualizaciones del desarrollo de los mismos.

Capítulo 6 – Conclusiones y Líneas de trabajo futuras: Dentro de este capítulo se hace una valoración global del trabajo realizado, exponiendo algunas conclusiones, y se apuntan algunas posibles tareas a desarrollar en un futuro, tras la conclusión del actual proyecto.

Capítulo 7 – Anexos: en este apartado se añade el grueso del código desarrollado en la implementación de nuestro programa / componente scanmatchingComp.

Capítulo 8 – Bibliografía: en este último apartado se incluirán las referencias bibliográficas utilizadas para la realización de este proyecto.

Capítulo 2:
RobEx y RoboComp

El entorno en el que se desarrolla el proyecto queda caracterizado por el robot sobre el que será integrado (familia RobEx) y por el sistema de componentes (RoboComp).

El robot en el que se llevarán a cabo las pruebas es un robot de la serie RobEx. Estos son robots móviles con conectividad wireless y/o Ethernet, disponen de un ordenador de a bordo en el que se ejecutan algunos componentes y desde el que pueden interactuar con otros componentes ejecutados de forma remota. En general las tareas del ordenador de a bordo se centran en el control del hardware del robot: movimiento de la base y captura de las señales de los dispositivos incorporados, láser, cámaras, micrófonos, etc.

El sistema de componentes llamado RoboComp puede definirse como un grafo de componentes o procesos que pueden ser distribuidos en varios procesadores proporcionando un sistema de comunicación que permite el intercambio de información entre sí mediante una síntesis muy simple.

El proyecto consistirá en la creación de los componentes necesarios para llevar a cabo los objetivos planteados anteriormente. Para ello se aprovechan todas las funcionalidades ya cubiertas por los componentes existentes, así como los conceptos y metodología de desarrollo. Por la razón fundamental de la reutilización en la programación orientada a componentes, estos están pensados con la esperanza de que en un futuro no muy lejano, otros los utilizarán para incorporar nuevas características al robot, como ya se ha hecho en este proyecto.

Tanto RobEx como RoboComp son proyectos desarrollados en el Laboratorio de Robótica de la Universidad de Extremadura (Robolab). Son modelos libres y se puede acceder a ellos mediante sus correspondientes páginas web [11], [12]. Dado que la implementación del proyecto fin de carrera se ha incluido en RoboComp, este software se distribuirá bajo la misma licencia.

A lo largo de este capítulo, se describirá el entorno de desarrollo y herramientas usadas, se hará una breve descripción del robot RobEx, se introducirá la estructura de programación orientada a componentes y se describirá detalladamente el repositorio de componentes RoboComp.

2.1 RobEx

El robot RobEx [12] , [14] es una base robótica libre desarrollada en el Laboratorio de Robótica y Visión Artificial de la UEx, Robolab [16].

Es de tipo diferencial, es decir, el movimiento del robot depende únicamente de la velocidad de rotación de sus dos ruedas motrices, generándose el giro a partir de la diferencia de velocidad entre éstas (*figura* 8). Además de las motrices, el robot dispone de una tercera rueda, de giro libre, que sirve de apoyo.

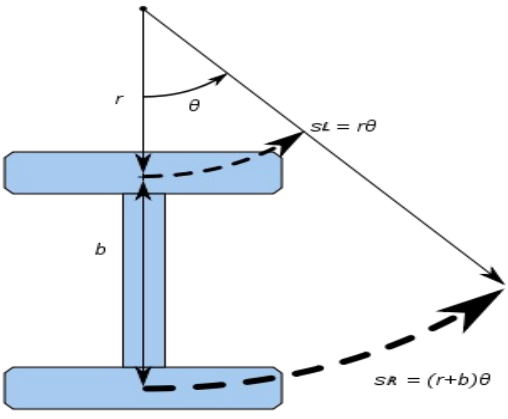

Figura 8: Dirección diferencial.

La simplicidad de diseño y de los cálculos asociados al modelo diferencial son las principales razones que han definido el diseño, no sólo de RobEx, sino de otros muchos robots y gamas de ellos, como Segway, Kephera, o Roomba (*figura* 9).

Sus planos se distribuyen bajo la licencia "Creative Commons Attribution-Share Alike 3.0". Por tanto, su diseño está abierto a cualquiera que lo quiera consultar o contribuir a él. Al ser de tipo diferencial su construcción es relativamente sencilla. Si se tienen conocimientos de electrónica, cualquiera puede llevarla a cabo.

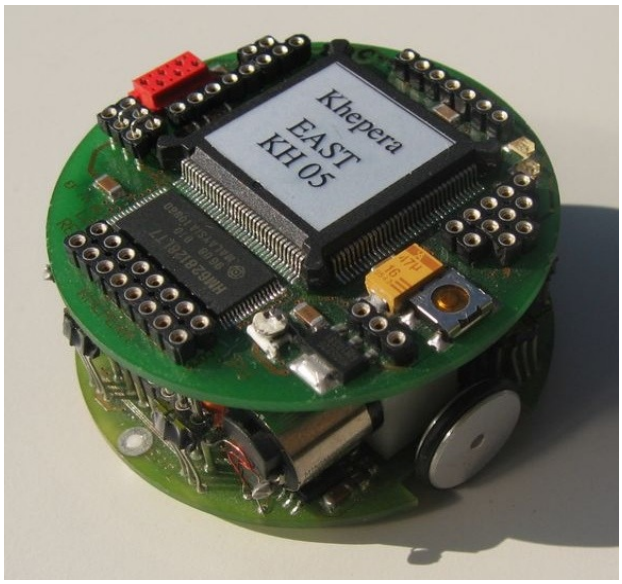

Figura 9: Robot móvil diferencial *Kephera, gama*_I.

RobEx está diseñado para llevar uno o varios ordenadores portátiles a bordo para realizar los procesos complejos que sean necesarios, hasta hoy relacionados con la visión, la audición o la inteligencia artificial.

Tanto los motores de la base, de corriente continua, como el resto de la electrónica, se alimentan de una batería de polímero de litio como las que suelen usarse para extender la autonomía de los ordenadores portátiles.

La base está controlada por un microcontrolador dedicado al que se accede a través de una interfaz RS232 sobre USB.

Su principal orientación es la investigación y la docencia. Aunque cada vez resultan más versátiles y pronto darán el salto a la realización de trabajos concretos en empresas. Llevan utilizándose más de cinco años a diario en las asignaturas de Robótica y Teoría de Sistemas que se imparten en la carrera de Ingeniería en Informática de la UEX. El origen de su diseño y desarrollo hasta su estado actual nació de los diferentes proyectos de investigación realizados en Robolab a partir de su creación en el año 1999.

Desde su primera versión, cada mejora y nueva funcionalidad que incorpora se prueba intensivamente tanto en el laboratorio como en las aulas, por lo que se consigue una robustez de las diferentes versiones considerable.

Figura 10: Robot móvil diferencial *RobEx, bender*_II.

Los objetivos de diseño de los robots RobEx son:

- Ser apropiado para entornos estructurados, pero que a la vez pueda ser modificado para otros tipos de terreno.
- Conseguir un robot de bajo coste y fácil de construir con capacidad de procesamiento intensivo a bordo.
- Que el precio no esté reñido con la calidad: fiabilidad y robustez.
- Poder ser ampliado con diversos accesorios de sensorización y de manipulación, así como llevar varios portátiles a bordo.
- Servir de plataforma para formar a futuros investigadores.
- Dar cabida a la experimentación utilizando hardware real en entornos controlados, una de las asignaturas pendientes en un plan de estudios tan teórico.

Como se indicó antes, el robot RobEx es hardware libre. Todo el software desarrollado se distribuye bajo licencia GPL. Con esto se pretende:

- Que cualquier persona tenga acceso al diseño y software del robot, y pueda fabricarlo por sí misma.
- Que la comunidad participe en la mejora y evolución del robot.
- Que cualquier empresa pueda usar o distribuir RobEx, pero que cualquier modificación hecha al robot o a su software se haga pública y mantenga la misma licencia.

2.1.1 Características técnicas

En la figura se muestra una representación del diseño de la parte mecánica de RobEx.

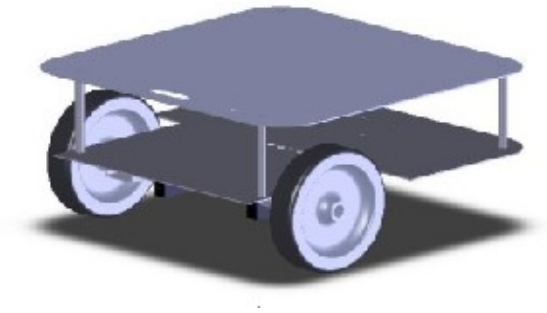

Figura 11: Representación del chasis de RobEx.

La estructura del robot está formada por dos planchas de aluminio que se separan y se afianzan entre sí mediante cuatro tubos de acero.

Para colocar los motores en el chasis se usan soportes de acero; unidos a estos motores se encuentran las dos ruedas motrices (las que aparecen en la figura), mientras que la rueda de giro libre va colocada en la parte posterior de la plataforma.

Para disponer de autonomía energética, el robot incorpora una batería recargable a bordo. Para ello, se utiliza un batería de polímero de litio de 21,6V y 3500mAh.

Estas baterías son de consumo doméstico común y se suelen utilizar para ampliar la autonomía de ordenadores portátiles, por lo que son asequibles y están sobradamente probadas. Estas baterías proveen al robot de una autonomía de algo menos de dos horas.

El sistema de control también se encarga de realizar las operaciones y cálculos necesarios para el control PID de los motores. El bucle de control PID de cada motor funciona a una frecuencia de 1 Khz. Por tanto, cada milisegundo se calcula la tensión de salida más apropiada para alcanzar el objetivo actual. En otros términos, el motor sólo está "sin control" o en lazo abierto por periodos de 1milisegundo.

La base dispone de un circuito integrado, LSI7266R1, que hace de contador de los pulsos procedentes de los dos codificadores ópticos. Éste se comunica con el microcontrolador mediante dos buses, uno de datos de 8 líneas de entrada y salida y otro de control de 5 líneas de entrada. Esto permite obtener los datos necesarios para llevar las cuentas de la Odometría, método que ha sido explicado detalladamente en el capítulo anterior y en el que se va a centrar buena parte de nuestro proyecto.

2.1.2 Características adicionales

A esta forma "básica" de RobEx se han incorporado una serie de componentes adicionales que aumentan sus capacidades, concretamente una torreta estéreo de visión mediante cámaras y un sensor láser.

Además de estos, RobEx dispone de una serie de componentes adicionales que no se comentarán puesto que no han sido usados durante el desarrollo del proyecto. Como ejemplo podemos destacar un sistema de captura y digitalización de audio, o un sistema inercial de 5 gdl's, 3 acelerómetros lineales y 2 giróscopos.

2.1.2.1 Torreta estéreo

La torreta estéreo (Figura 3: Torreta estéreo) está formada por una estructura en forma de U invertida con dos pies sobre los que se sitúan las cámaras, esta estructura esta diseñada en aluminio lo que le confiere una gran solidez. Cada

cámara está acoplada a su base a través de una sujeción ensamblada a un motor que proporciona un giro independiente sobre el eje vertical. En el interior de la U, en uno de sus laterales, se sitúa un tercer motor que hace girar toda la estructura, proporcionando un movimiento rotatorio simultáneo de ambas cámaras sobre el eje horizontal.

Este diseño admite, por lo tanto, 3 grados de libertad sobre la cabeza robótica: uno de elevación (tilt) común a las dos cámaras, y dos para el giro (pan) de cada cámara. No obstante, el rango de giro de cada cámara está limitado a causa del diseño mecánico para evitar el choque de éstas con la estructura, por lo que no todos los movimientos son posibles.

A continuación, se muestra la torreta con las cámaras que proporcionan la visión estéreo, que lleva incorporado nuestro robot móvil:

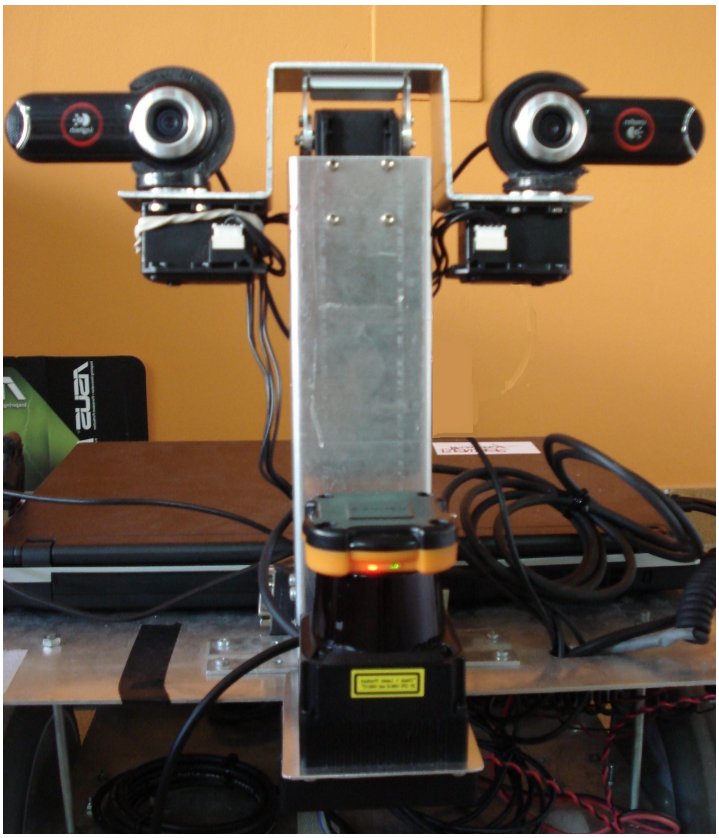

Figura 12: Torreta estéreo

Los motores utilizados son servos Dynamixel RX-10 con microcontrolador incorporado y bus digital de comunicación convertible a USB, mediante el que se conectan al ordenador de a bordo. Toda esta estructura es controlada por un componente "cammotionComp" diseñado específicamente para esta tarea.

2.1.2.2 Sensor láser

Nuestros robots van a disponer también de un par de sensores láser de la casa Hokuyo.

Se van a describir a continuación:

a) Láser HOKUYO URG-04LX

El láser Hokuyo URG-04LX pertenece a los llamados *Scanning Laser Range Finder*, uno de los mejores sensores optimizados para el reconocimiento de entornos.

CARACTERÍSTICAS destacables:

- Alta precisión, alta resolución y buena amplitud angular que proporcionan la mejor solución para los robots autónomos móviles en entornos desconocidos.

- Su tamaño compacto permite una mayor libertad de diseño. El peso ligero y bajo consumo de energía contribuyen a poder realizar operaciones de larga duración.

- No le influye cómo sea el brillo del medio ambiente. Excelente funcionamiento en la oscuridad.

- Reconocimiento del tamaño y la posición de un cuerpo humano sin incurrir en su privacidad.

Este sensor láser que hemos utilizado en nuestro proyecto es el que se muestra en la siguiente imagen:

Figura 13: Hokuyo URG-04LX Laser Sensor Ranger

A continuación, se detalla la tabla de especificaciones técnicas proporcionada por el fabricante para este modelo:

Model No.	*URG-04LX*
Power source	5VDC±5%
Current consumption	500mA or less(800mA when start-up)
Measuring area	60 to 4095mm(white paper with 70mm) / 240º
Accuracy	60 to 1,000mm: ±10mm, 1,000 to 4,095mm: 1% of measurement
Repeatability	60 to 1,000mm : ±10mm
Angular resolution	Step angle : approx. 0.36°(360°/1,024 steps)
Light source	Semiconductor laser diode (λ=785nm), Laser safety class 1 (IEC60825-1, 21 CFR 1040.10 & 1040.11)
Scanning time	100ms/scan
Noise	25dB or less
Interface	USB, RS-232C(19.2k, 57.6k, 115.2k, 250k, 500k, 750kbps), NPN open-collector (synchronous output of optical scanner : 1pce)
Vibration	10 to 55Hz, double amplitude 1.5mm Each 2 hour in X, Y and Z

resistance	directions
Impact resistance	196m/s^2, Each 10 time in X, Y and Z directions
Weight	Approx. 160g

Tabla 1. Especificaciones técnicas Hokuyo URG-04LX

b) Láser HOKUYO UTM-30LX

El láser Hokuyo UTM-30LX es también un *Scanning Laser Range Finder*, adecuado para robots inteligentes con alta velocidad de movimiento por su amplio rango y su rapidez de respuesta.

CARACTERÍSTICAS destacables:

- Apropiado para realizar medidas en exteriores.
- Gran ángulo de visión (270º), con una resolución de 0,25º.
- Compacto y ligero: 60x60xH87mm, con 370 gramos de peso.
- 12V de tensión de trabajo, con interfaz USB 2.0 de alta velocidad.

Este sensor láser que también hemos usado en el desarrollo de los experimentos, es el que se muestra en la siguiente imagen:

Figura 14: Hokuyo UTM-30LX Laser Sensor Ranger

A continuación, se detalla la tabla de especificaciones técnicas proporcionada por el fabricante para el modelo *UTM-30LX*:

Model No.	*UTM-30LX*
Power source	12VDC±10%
Current consumption	Max: 1A, Normal:0.7A
Measuring area	0.1 to 30m (White Square 500mm or more) / 270°
Accuracy	0.1 to 10m: ±30mm, 10 to 30m: ±50mm
Light source	Semiconductor laser diode (λ=785nm) Laser safety Class 1 (FDA)
Angular resolution	0.25° (360°/1,440 steps)
Scanning time	25msec/scan
Noise	Less than 25dB
Interface	USB2.0(Full Speed) Synchronous output (NPN open-collector)
(Temperature / Humidity)	-10 to +50 ℃, less than 85%RH (without dew and frost)
Vibration Resistance	Double amplitude 1.5mm 10 to 55Hz, 2 hours each in X, Y and Z directions
Impact resistance	196m/s^2, Each 10 time in X, Y and Z directions
Weight	Approx. 370g (with cable attachment)

Tabla 2. Especificaciones técnicas Hokuyo UTM-30LX

2.2 Programación orientada a Componentes

Dos de los principales problemas que se presentan cuando se crea software son la escalabilidad y la reusabilidad. Estos problemas son especialmente agudos cuando se trata de software que se va a emplear en robótica, pues la reutilización es algo excepcional en este campo. A pesar de la importancia de la reusabilidad, generalmente se suele perder de vista este aspecto y se acaba creando software monolítico y poco utilizable.

En el ámbito de la robótica es muy común que los investigadores implementen todos los algoritmos con un diseño rígido y orientado a una tarea y/o a un robot específico, en muchos casos debido a unos requerimientos de tiempo y funcionalidad muy específicos. De ser así, cuando finaliza la etapa de implementación el software desarrollado acaba siendo imposible de utilizar. Suele estar tan ligado a una plataforma o tarea específica que resulta más práctico empezar de cero (debido a las dependencias y efectos colaterales derivados de su rigidez).

La programación orientada a componentes [16] surge como solución a este tipo de problemas. Es un enfoque que no tiene necesariamente que ver con concurrencia o computación distribuida, sino con cómo se organiza el software.

La programación orientada a objetos representó un gran avance respecto a la programación estructurada, sin embargo, cuando el número de clases y sus interdependencias crece de forma considerable, resulta demasiado difícil entender el sistema globalmente. Es por tanto beneficioso disponer de un grado mayor de encapsulamiento, que aune diferentes clases relacionadas bajo una interfaz única, y permita comprender el sistema con menor grado de detalle.

Muchos ven la programación orientada a componentes, que se propuso para solucionar este tipo de problemas, como el siguiente paso tras la programación orientada a objetos [10].

Un componente es un programa que provee una interfaz que otros programas o componentes pueden utilizar. Es una pieza de código elaborado o a elaborar que codifica una determinada funcionalidad. Son los elementos básicos en la construcción de las aplicaciones en esta arquitectura, que se juntan y combinan para llevar a cabo una tarea concreta.

A su vez, estos programas hacen uso de programación orientada a objetos (así como en programación orientada a objetos se hace uso de programación estructurada). Esta división en piezas de software de mayor tamaño que las clases, implementadas como subprogramas independientes ayuda a mitigar los problemas de los que se ha hablado antes y a aislar errores. Además, ayudan a repartir la carga de cómputo entre núcleos, incluso, distribuyendo entre diferentes ordenadores en red.

Desde el punto de vista del diseño se pueden ver como una gran clase que ofrece métodos públicos. La única diferencia desde este punto de vista es que la complejidad introducida por las clases de las que depende el componente que no son del dominio del problema, desaparece, porque la interfaz del componente las esconde. Un componente puede ser arbitrariamente complejo, pero un paso atrás, lo único que se ve es la interfaz que ofrece. Esto es lo que lo define como componente.

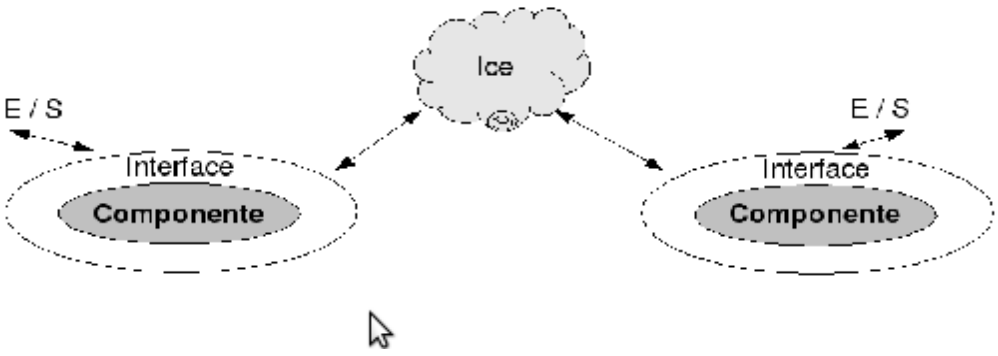

Figura 15: Representación genérica de componentes

Por tanto, si cada componente realiza una serie de tareas o responde a una serie de órdenes, necesitamos quien se encargue de esa comunicación. Es donde entra en juego Ice, pieza clave para la comprensión de la arquitectura y para la funcionalidad de la misma. Ice es un framework de comunicación muy interesante para su uso en robótica, sus características principales serán comentadas posteriormente. A pesar de estar desarrollado por una empresa privada está licenciado bajo GPL. Además de su carácter libre hay otras dos razones para elegir Ice.

La primera razón es su facilidad de uso: la filosofía de Ice es soportar sólo aquellas características que los usuarios vayan realmente a necesitar y evitar introducir otras que raramente se usan y dificultan el aprendizaje.

La segunda razón es la eficiencia: si bien Ice codifica la comunicación eficientemente tanto en términos de tiempo como de espacio, otras alternativas suelen codificar la comunicación en XML. El uso de XML tiene ventajas en otras aplicaciones donde es conveniente que los humanos puedan entender el tráfico y no sean factores críticos ni la latencia ni la eficiencia, pero dentro de un robot esto no ocurre.

Ice soporta dos tipos de comunicación, por llamada remota (tipo RPC) o por suscripción (mediante un servicio que hace de servidor de mensajes). Sin embargo, este último introduce un paso intermedio cuya latencia hace desaconsejable su uso en robótica donde la ejecución en tiempo real es fundamental.

Para realizar una conexión con un componente lo único que se ha de conocer es su “endpoint”, es decir, la información necesaria para realizar la conexión: la dirección o nombre del ordenador al que se va conectar, el protocolo, el puerto y el nombre de la interfaz. Por ejemplo:

< nombre >:< tcp/udp > –p puerto – h host

donde 'nombre' es el nombre de la interfaz del componente al que se quiere hacer la conexión; 'puerto' es el puerto tcp o udp en el que el componente esté escuchando y 'host' el nombre del ordenador donde el componente se está ejecutando.

Desde el punto de vista del programador, una vez la conexión está hecha, el uso de componentes Ice es extremadamente simple. Cuando se realiza una conexión se crea una instancia de un proxy al componente en forma de objeto.

Al ejecutar un método del componente proxy, el framework se encarga automáticamente de redirigir la llamada al componente remoto, por lo que la instancia del proxy se utiliza como si se tratase de una instancia de un objeto con la funcionalidad del componente.

Para usar programación orientada a componentes se ha de dividir el diseño del software en piezas que ofrezcan una interfaz. A cambio obtendremos mayor reusabilidad, utilizando los mismos componentes en diferentes contextos, de esta forma se reduce considerablemente el tiempo, el coste y el esfuerzo de desarrollo de nuevas aplicaciones, aumentando a la vez la flexibilidad, reutilización y fiabilidad de las mismas.

Para concluir, cabe destacar que será más fácil aislar y encontrar fallos, consiguiendo eliminar la necesidad de contemplar cientos de clases para comprender el software desarrollado.

2.3 RoboComp

RoboComp, es un repositorio de componentes basados en Ice, con aplicaciones en robótica y visión artificial. RoboComp se comenzó a desarrollar en Robolab en 2005. Actualmente el proyecto ha sido migrado a SourceForge,

donde, además de tener la página del proyecto (http://sf.net/projects/robocomp), dispone de un wiki (http://robocomp.wiki.sf.net/), donde hay documentación y un repositorio al que se puede acceder incluso directamente con un navegador web (https://robocomp.svn.sf.net/svnroot/robocomp).

La figura 16 muestra un grafo en el que se pueden ver los componentes de los que dispone RoboComp y las dependencias entre ellos:

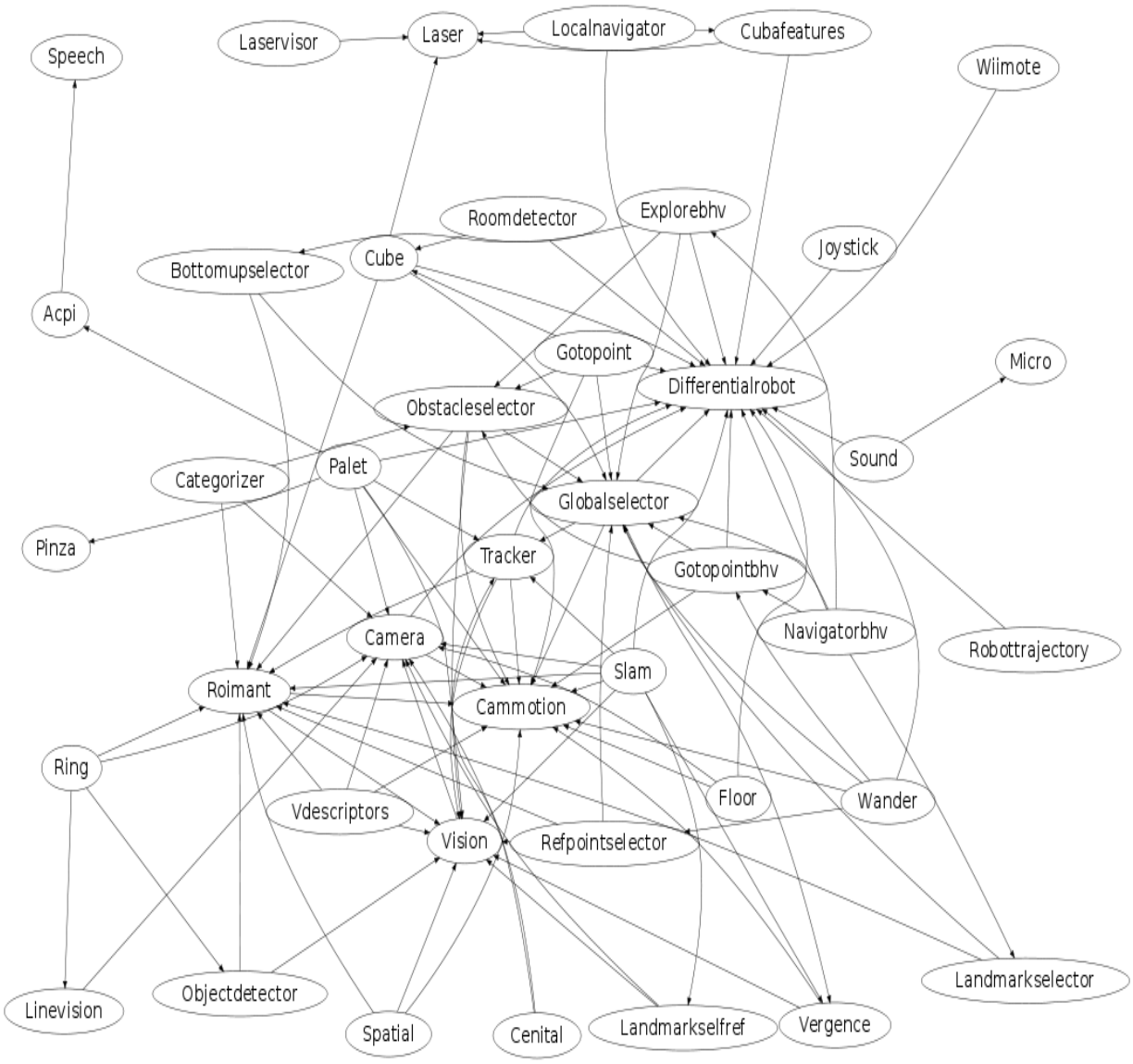

Figura 16: Grafo de componentes de RoboComp

Dispone de componentes para captura y visualización de vídeo, control del robot RobEx, detección y mantenimiento de regiones de interés (ROI), lectura y visualización de láser, lectura de joystick y navegación, entre otros muchos. Además, dispone de un generador automático de componentes nuevos; de un programa gráfico de manipulación y control de componentes, el managerComp; y hasta de un componente encargado de grabar los datos producidos por los sensores para su posterior reproducción, el replayComp.

Todos los componentes, así como clases auxiliares que se desarrollen a lo largo del proyecto se encontrarán inmersos dentro de este repositorio por lo que resulta esencial conocer su estructura.

2.4 Entorno de desarrollo

A la luz de lo comentado anteriormente, es fácil suponer que para desarrollar una aplicación basada en esta arquitectura, es necesario utilizar un conjunto de herramientas más o menos complejas y librerías que soporten las tareas exigidas a las capacidades del robot. La naturaleza de los problemas, eminentemente en tiempo real, a su vez, exige que el tiempo de proceso sea mínimo y siempre se persigue, por filosofía del laboratorio y en la medida de lo posible, un mínimo consumo de recursos.

En las siguientes páginas se describe el software utilizado.

2.4.1 IPP/FrameWave

IPP y FrameWave son dos bibliotecas para procesar señales de una y dos dimensiones. La elección de estas bibliotecas frente a otras de similar objeto se tomó por su gran eficiencia. Ambas bibliotecas son competencia directa, de hecho, mientras IPP es desarrollada por Intel, FrameWave es desarrollada por AMD. La diferencia en el rendimiento respecto a sus competidores radica en que hacen uso de las instrucciones SIMD que aportan las extensiones de x86, 3DNow!, MMX y SSE y derivadas que ambos fabricantes incluyen en sus procesadores.

Ambas bibliotecas ofrecen APIs muy similares, pero se diferencian en la licencia bajo la que se distribuyen: mientras FrameWave se distribuye bajo la licencia libre Apache 2.0, IPP es software privativo. A pesar de que es gratuito para el uso personal bajo GNU/Linux, hay que pagar la licencia si se desea usar comercialmente, y su código fuente no es público.

La eficiencia del software desarrollado, no sólo en este proyecto sino en el resto del software se debe en gran parte a la de estas bibliotecas. De ellas se han usado multitud de funciones en otros componentes relacionadas con el tratamiento de imágenes. En este trabajo se ha utilizado para el análisis de la señal en el dominio de la frecuencias.

2.4.2 Qt4

Qt4 es un framework de desarrollo cuyo objeto principal es la creación de interfaces gráficas a nivel de usuario. A pesar de que sea este su principal uso, engloba una gran cantidad de funcionalidades distintas: interfaz multiplataforma con el sistema operativo (sistema de ficheros, procesos, hilos entre otras cosas), comunicación por red mediante sockets, interfaz con OpenGL, conexiones SQL, renderizado de HTML, etc. y módulos de reproducción y multimedia.

Atendiendo al lenguaje de programación, Qt utiliza el estándar de C++, al que añade un pre-procesador que genera el código en C++ necesario para implementar sus propios módulos de extensión; pero existen multitud de bindings que hacen posible su uso desde otros lenguajes como Java, C#, Python, Perl y otros muchos.

Inicialmente fue desarrollada por Trolltech, una empresa noruega, bajo una licencia privativa. Después de varios cambios en la política de licencias pasó a distribuirse bajo una doble licencia GPL/QPL (esta última privativa).

Finalmente, tras la compra de Trolltech por parte de Nokia, Qt fue licenciada bajo LGPL en 2009, eliminando así cualquier debate sobre la licencia de la biblioteca.

2.4.3 Qt4 designer

Qt 4 designer es una herramienta de la empresa Trolltecha para diseñar y construir interfaces gráficos de usuario desde los componentes Qt. Permite diseñar y construir widgets y dialogs de una forma sencilla y eficaz.

Una característica esencial de este diseñador es que permite aprovechar las señales y slots de Qt lo que facilita la tarea de conexión del interfaz con el código interno de la aplicación.

2.4.4 Ice

Ice es el middleware del que se habló en la sección anterior que permite crear componentes software que pueden funcionar de forma distribuida sobre plataformas heterogéneas, de forma que puedan comunicarse entre si y llevar a cabo un trabajo conjunto.

Es usado por RoboComp y, por tanto, por los componentes desarrollados en el proyecto de fin de carrera. Es el principal producto de la empresa estadounidense ZeroC, y se distribuye bajo una doble licencia GPL+ privativa

para habilitar que las empresas desarrollen software privativo con Ice, a cambio del pago de una licencia. Las principales características de Ice frente a otras tecnologías similares es su rapidez, baja latencia y escalabilidad.

Uno de los problemas a resolver a la hora de crear componentes software es la creación de un lenguaje de definición de interfaces. Ice usa Slice, un lenguaje que se creó específicamente para este propósito.

2.4.5 CMake

CMake es una aplicación que permite generar automáticamente ficheros Makefile y ficheros de proyecto de varios IDE como KDevelop o Eclipse entre otros. CMake permite delegar la creación de ficheros Makefile, consiguiendo un resultado multiplataforma y robusto, sin llegar a perder el control del proceso de compilación.

CMake es una iniciativa libre que nació como respuesta a la ausencia de una alternativa suficientemente buena durante el desarrollo de una librería llamada ITK. Si bien dispone de soporte para Qt – cuyo objetivo de diseño más importante es hacer la programación multiplataforma más fácil, abstrayendo la funcionalidad de bajo nivel en las capas inferiores de sistemas operativos- y algunas otras extensiones, es muy simple hacer nuevas extensiones.

2.4.6 KDevelop

Es un entorno de desarrollo integrado para sistemas GNU/Linux y otros sistemas Unix, publicado bajo licencia GPL, orientado al uso bajo el entorno gráfico KDE, aunque también funciona con otros entornos, como Gnome.

El mismo nombre alude a su perfil: KDevelop - KDE Development Environment (Entorno de Desarrollo para KDE). A diferencia de muchas otras interfaces de desarrollo, KDevelop no cuenta con un compilador propio, por lo que depende de gcc para producir código binario.

Para la elaboración de este proyecto, hemos utilizado el lenguaje C++ y el paradigma de programación orientada a componentes, con el objetivo de obtener la meta de cualquier modelo de programación estructurada convencional, que no es otro que el de establecer una serie de normas de desarrollo que aseguren y faciliten la mantenibilidad y reusabilidad del código.

En la programación orientada componentes es importante conocer los conceptos de *objeto* y *clase*.

• Objeto: entidad que tiene unos atributos (propiedades) y unos métodos para operar con ellos. Es una representación real de la clase y no es un dato simple, sino que puede contener en su interior cierto número de atributos bien estructurados.

• Clase: definición de un tipo de objetos; especifica los atributos y métodos del mismo.

2.4.7 OpenSSH

OpenSSH es una implementación libre del protocolo SSH (Secure SHell) que ofrece tanto la parte del servidor como la del cliente. Esta herramienta es importante en la puesta en marcha de proyecto porque, junto a managerComp, nos permite ejecutar rápida y remotamente los componentes que se deseen. Además, gracias a que permite autenticación basada en llaves, se puede trabajar de forma segura sin necesidad de escribir la contraseña cada vez que se quiere cambiar el estado de algún componente.

OpenSSH es una iniciativa de los desarrolladores de OpenBSD. Se distribuye con distintas licencias (dependiendo de la pieza), pero a pesar de esto, todas ellas son libres, GPL o BSD.

2.4.8 managerComp

La aplicación managerComp permite visualizar, tanto gráficamente como en una lista, el estado de los componentes configurados en tiempo real. A pesar de estar integrado en RoboComp, se detalla independientemente por no ser un componente propiamente dicho.

Además de la visualización del estado de los componentes, también nos permite "arrancarlos" y "pararlos".

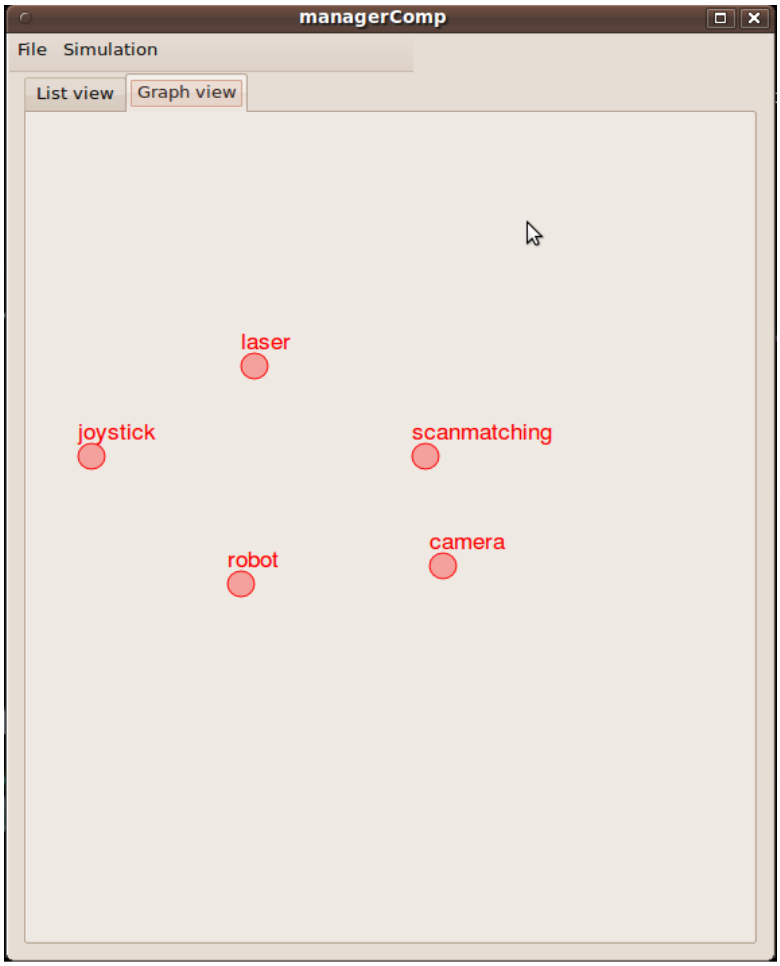

Figura 17: Herramienta managerComp

2.4.9 GNU/Linux

El sistema operativo usado durante el desarrollo y los experimentos ha sido la plataforma GNU/Linux. Gracias a él disponemos de un entorno de desarrollo gratuito y libre, herramientas de compilación, depuración, y una gran cantidad de bibliotecas complementarias.

Si bien la familiarización con todas estas herramientas y con el entorno Ubuntu requiere un período de adaptación, puede decirse que es bastante intuitivo y que trabaja muy bien bajo condiciones de gran consumo de recursos. Finalmente, cabe decir que se ha utilizado la distribución Ubuntu por razones obvias de facilidad de obtención y múltiples funcionalidades.

Capítulo 3:
Descripción del sistema

En este tercer capítulo, vamos a explicar de una forma descriptiva en qué consiste la localización basada en Scan Matching aplicada a un robot; es lo que se conoce como *Scan Matching Self-Localization*.

Definamos, primero, estos conceptos: *Self-Localization* podría traducirse en español como "auto-localización", es decir, la capacidad del robot de conocer con la mayor precisión posible la posición global en la que se encuentra dentro de un espacio determinado, en un momento dado. Esta capacidad es, también, una necesidad del robot, ya que éste necesita saber dónde está situado en cada instante para poder moverse en su entorno y ser capaz de desplazarse hasta una posición determinada, salvando obstáculos y eligiendo la trayectoria adecuada.

Scan Matching haría referencia al método utilizado para obtener la posición del robot; y éste consiste en la comparación de dos scans, uno actual y otro inmediatamente anterior, el de referencia. Llamaremos scans a cada uno de los barridos que hace el láser, con un determinado rango angular de la medida (que en el caso del láser URG-04LX es de 240º y, en el caso del UMT-30LX, es de 270º) y con un determinado número de puntos tomados, que depende de su resolución (para el URG-04LX son 632 puntos adquiridos y para el UMT-30LX serán 878 puntos).

El término "*scan matching"* equivale en español a "equiparación de scans o correspondencia entre scans o análisis". Con lo que, comparando las variaciones que percibe el sensor láser en la geometría de su entorno (en las distancias hasta cada uno de los puntos del scan), el robot es capaz de saber cuánto se ha desplazado y en qué dirección, es decir, es capaz de saber la trayectoria que ha realizado; conociendo de esta forma la nueva posición en la que se encuentra.

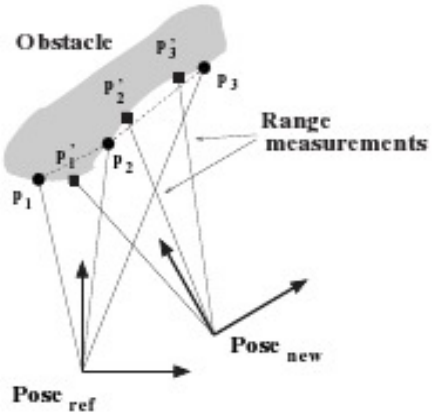

Figura 18: *Scan Matching* - Posición actual y de referencia.

La imagen muestra el concepto de *scan matching*, objeto de estudio en el presente proyecto. Los puntos del scan actual son comparados con los de referencia, de forma que se busque el giro y la translación que haga máxima la superposición de estos puntos.

3.1 IDENTIFICACIÓN DEL PROBLEMA. ESPECIFICACIÓN DETALLADA DEL PLAN.

Como ya hemos comentado con anterioridad, el problema al que vamos a enfrentarnos y que trataremos de solucionar es la falta de precisión de la odometría a la hora de estimar la posición en la que se encuentra el robot en cada momento. Sabemos que esta precisión empeora notablemente en los desplazamientos largos, ya que a lo largo del tiempo existe una inevitable acumulación de errores. En concreto, sabemos que la odometría funciona casi a la perfección en desplazamientos rectilíneos; pero cuando se efectúan trayectorias sinuosas con giros considerables, la acumulación de errores de orientación causa errores notables en la estimación de la posición.

Los robots móviles usan la odometría para la estimación de su posición de forma relativa a su localización inicial; y, al ir aumentando éstos errores proporcionalmente con la distancia recorrida por el robot, la precisión de la odometría baja notablemente.

Con la localización basada en Scan Matching se va a tratar de mejorar la precisión en desplazamientos largos y con giros y, sobretodo, evitar las imprecisiones que introducen los errores no sistemáticos como el patinaje de las ruedas debido a: suelos resbaladizos, sobre-aceleración, o, por ejemplo, cuando no hay ningún punto de contacto con el suelo, como cuando pasamos las ruedas por encima de un cable.

Lo que vamos a hacer es crear un componente, al que hemos denominado scanmatchingComp, y en el que queremos implementar los planteamientos de "s*can matching self-localization*" de Jens-Steffen Gutmann, profesor del Departamento de Informática de la Universidad de Freiburg, Alemania [2].

En sus planteamientos trata de comparar varios algoritmos existentes en la literatura del área y combinarlos con un método propuesto por él mismo para robótica de interiores.

Este componente tomará los datos del láser para un instante "t" actual y un instante "t-1", inmediatamente anterior, que se obtiene almacenando los puntos de medida del scan actual, pasando éste a ser el scan de referencia en el siguiente barrido. A continuación, adaptaremos estos datos, que son obtenidos directamente del láser, a una estructura tipo Scan con la forma del algoritmo de Gutmann, para poder realizar el scan matching.

Seguidamente, nuestra función principal llamará a las funciones que implementarán los tres algoritmos que vamos a comentar más adelante: en la sección 3.3, un algoritmo basado en *Iterative Dual Correspondence* (*IDC algorithm*), [2]; en el punto 3.4 un método basado en los desarrollos de I. J. Cox (*COX algorithm*), [3]; y, tercero y último, en la sección 3.5 el planteamiento de Gutmann, *Combined scan matcher*, [2].

Finalmente, realizaremos una función que nos dibujará, en tiempo real, las trayectorias seguidas por el robot a lo largo de los desplazamientos en su entorno. Todos estos resultados, junto a otros parámetros obtenidos internamente y la visión de las cámaras, se mostrarán por pantalla en una interfaz gráfica que hemos modificado mediante la herramienta de diseño y construcción Qt4 designer.

Todo este proceso va a ser detallado paso a paso en el siguiente capítulo, Diseño e Implementación del sistema.

Se va a desarrollar una estructura de componentes, diseñada como solución al problema. Además de especificar el nuevo componente que añadiremos al repositorio de RoboComp, también se van a comentar las principales características y funcionalidades de los componentes preexistentes que hemos utilizado en nuestro proyecto.

Gracias al uso de la programación orientada a componentes, se puede conseguir tener una visión global del sistema bastante fiel a la realidad, sólo con ver el grafo de componentes.

En la siguiente imagen, puede observarse el grafo que integra los cinco componentes en los que se basará nuestro sistema y que se van a detallar en el siguiente capítulo (*differentialrobotComp*, *laserComp*, *cameraComp*, *joystickComp* y *scanmatchingComp)* :

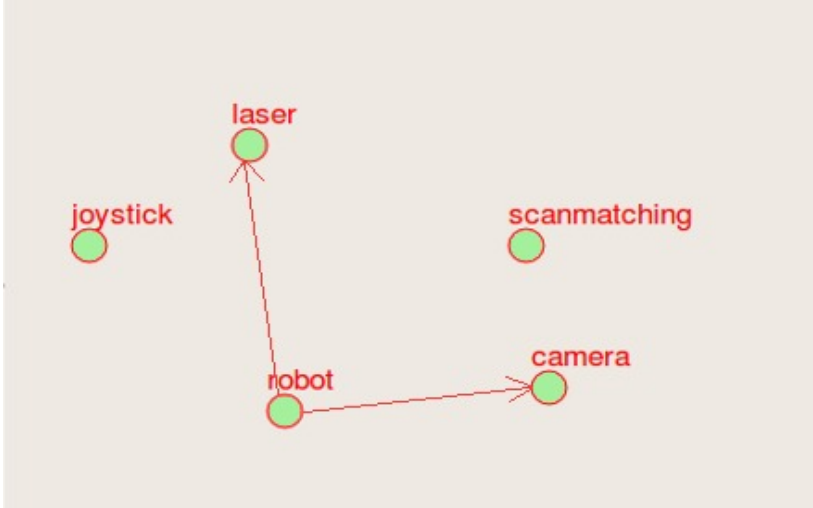

Figura 19: Grafo de Componentes.

La dirección de los enlaces hace referencia a la dependencia existente para el funcionamiento, no al sentido en el que se establece la comunicación.

3.2 LOCALIZACIÓN BASADA EN SCAN MATCHING

Debido a los objetivos de nuestro proyecto, un requisito indispensable para *self-localization* es tener la habilidad de hacer frente a entornos reales como una oficina, un almacén o un museo, que no tienen por qué ser espacios perfectamente ortogonales; así como ambientes curvilíneos, de forma tan fiable como se haría en un entorno poligonal de paredes rectas.

Además, queremos que los enfoques de nuestro sistema sean lo suficientemente robustos para hacer frente a ligeras modificaciones que puedan surgir en el día a día del entorno por el que se mueve robot. También deberá ser lo suficientemente rápido para que pueda ser usado de forma online en el propio sistema del robot.

Para cumplir con estos requisitos, ha habido que combinar los algoritmos de Gutmann con la programación orientada a componentes mediante lenguaje C++. Éste es un lenguaje de programación de alto nivel, multipropósito y con acceso a programación de bajo nivel, que posee una serie de propiedades difíciles de encontrar en otros lenguajes de alto nivel:

- Posibilidad de redefinir los operadores (sobrecarga de operadores).
- Identificación de tipos en tiempo de ejecución (RTTI).

Por ello, está considerado por muchos como el lenguaje más potente, debido a que permite trabajar tanto a alto como a bajo nivel.

Para los constructores de sistemas expertos, hay dos tipos de conocimiento: conocimiento acerca del problema particular o declarativo y conocimiento acerca de cómo obtener más conocimiento a partir del que ya tenemos o conocimiento procedural.

Para el conocimiento declarativo existen técnicas como los *Frames*

(marcos) que fueron los padres de lo que hoy conocemos como *Programación Orientada a Objetos*.

El conocimiento procedural también es llamado mecanismo de inferencia y requiere además de un método de búsqueda que permita tomar decisiones, como por ejemplo, seleccionar la regla a aplicar del conjunto total de posibles reglas. Para ello, los investigadores esperan poder resolver con una única y correcta teoría todos los problemas de inferencia y representación de conocimientos.

Los esfuerzos de la otra rama de la Inteligencia Artificial, el conocimiento declarativo, se orientan a simular los elementos de más bajo nivel que componen o intervienen en los procesos inteligentes. Ellos pretenden que, de esa combinación, emerja espontáneamente el comportamiento inteligente.

Esto adquiere una gran importancia cuando estamos trabajando en un entorno dinámico y ante situaciones imprevistas.

El robot móvil sobre el que desarrollaremos nuestro proyecto se trata de un pseudo-vehículo industrial con tres ruedas, del que ya conocemos sus características técnicas: es la estructura de hardware libre RobEx.

Para que el robot pueda realizar diferentes tareas, éste va a tener que desplazarse por el entorno; con lo que el sistema que vamos a desarrollar requiere el conocimiento de la posición global del robot con la mayor precisión posible.

Si la estimación de la posición relativa del robot con respecto a la inicial sólo se basa en la odometría, se irán acumulando errores de la misma sin que exista un límite superior; por lo que se hace necesaria la localización basada en scan matching: *Scan Matching Self-Localization*.

La mayoría de los planteamientos que usan scans de un láser para la "auto-localización" pueden clasificarse en dos categorías: los planteamientos que usan un mapa geométrico del entorno y los que equiparan un par de scans.

a) Planteamientos que utilizan un mapa geométrico:

Estos planteamientos comparan las características de un scan con un mapa global de características del entorno. Además, el mapa ha de ser determinado a priori o construido mientras se explora el entorno.

b) Planteamientos basados en Scan Matching:

Aquí, la posición del robot es estimada equiparando un par de scans . El primer scan sirve como referencia, es el "ref_scan", y tiene que ser almacenado previamente. El segundo scan, al que llamamos "current_scan" (*scan actual*) es comparado con el de referencia y su posición se determina relativamente a la del scan de referencia.

3.2.1 Elección del tipo de planteamiento.

Hay que tener en cuenta ciertos problemas a la hora de elegir el tipo de planteamiento mediante el que se encauzará nuestro proyecto:

La construcción a priori de un mapa exacto del entorno en el que se va a mover el robot es una tarea complicada. Nosotros no queremos tener que medir todas las distancias a mano y luego tener que utilizar una herramienta tipo CAD para construir el mapa. Pretendemos que sea el robot el que se cree su propio mapa mediante el uso del sensor láser.

Este tipo de planteamientos que permiten la construcción del mapa mientras se explora el entorno, generalmente empiezan con un mapa vacío. Cada intervalo de tiempo, se toma un nuevo scan y el robot estima su posición comparando el scan actual con el mapa construido hasta el momento y se actualiza el mapa integrando el nuevo scan en el mapa.

Esto funciona bastante bien hasta que la trayectoria del robot contiene un lazo. En este caso, el problema surge cuando el robot cierra el lazo: para la estimación de la posición dos partes del mapa pueden ser utilizadas. La primera parte consiste en los scans que han sido tomados más recientemente, mientras que la segunda parte consiste en los scans han sido tomados hace algún tiempo.

Debido a esto, puede haber saltos de posiciones en este área. Para evitar estos saltos de posición se necesita una corrección de todas las posiciones del scan a lo largo del recorrido del robot; pero esto también implicará una actualización geométrica del mapa, que puede resultar muy complicada y que, además, no está contemplada en la mayoría de planteamientos.

Para el desarrollo de nuestro sistema vamos a utilizar algoritmos basados en *scan matching* por las siguientes razones: scan matching no necesita de un mapa geométrico previamente concebido, existen algoritmos de comparación como el IDC (Iterative Dual Correspondence – algoritmo que explicaremos detalladamente más adelante-) que trabajan bastante bien en entornos arbitrarios que no tienen que ser necesariamente poligonales, y además, scan matching constituye la base actual en robótica para la estimación de la posición global.

Concretamente vamos a trabajar con tres diferentes algoritmos de comparación: algoritmo de emparejamiento mediante la asignación de puntos a segmentos de línea (COX algorithm – también será detallado más adelante-), algoritmo de emparejamiento mediante asignación punto a punto (IDC algorithm) y una combinación de ambos que trabaja de forma eficiente en entornos poligonales como en no poligonales (COMBINED scan matcher).

Lo que vamos a desarrollar es una estructura de componentes que van a interactuar entre sí, y en la que se va a integrar el ya mencionado scanmatchingComp, en el que hemos implementado los planteamientos de scan matching de Jens-Steffen Gutmann.

3.3 IDC Algorithm

El concepto de scan matching del algoritmo *idc* se basa en las asignaciones punto a punto (*point-to-point assignments*) entre un par de scans, que no tienen por qué basarse en características geométricas. Esto permite el uso de este algoritmo incluso en entornos que no son poligonales.

El algoritmo IDC – cuyas siglas hacen referencia a Iterative Dual Correspondence [2]- está basado, como bien indica su nombre, en una correspondencia punto a punto (dual) para calcular la alineación de los scans. El problema de conseguir una buena correspondencia es resuelto mediante dos estrategias: la regla del punto más cercano (*closest point rule*) y la regla del intervalo de correspondencia (*matching range rule*), [4]. Asimismo, existe también una fórmula para calcular la matriz de covarianza del error que pueda darse en el scan matching.

El algoritmo *idc* originario parecía ser algo lento, así que al que nosotros hemos implementado ya le habían sido añadidos unos filtros para reducir el número de puntos del scan.

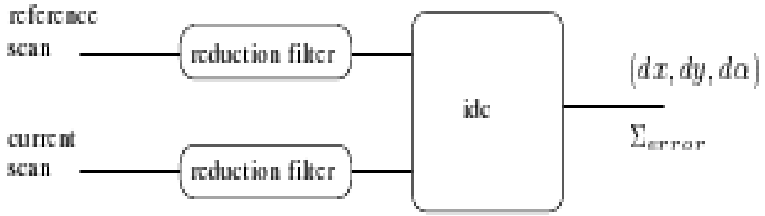

Figura 20: Extended IDC algorithm.

Este tipo de filtro sustituye nubes de puntos del scan por su centro de gravedad. De esta manera, se consigue el efecto de tener una distribución de los puntos del scan mucho más suave. También se reduce considerablemente el número de puntos del scan, sin que haya una pérdida de información demasiado notable. Es este número de puntos del scan el que será responsable del tiempo de ejecución del algoritmo *idc*.

3.4 COX Algorithm

La idea del scan matching mediante asignación de puntos a segmentos de línea (*assigning points to line segments*) es originada por Ingemar J. Cox, científico del Instituto de Investigación NEC de Princeton, New Jersey; que, desde su Departamento de Investigación Robótica, se dedica a desarrollar modelos para la estimación de la posición en navegación de robots móviles.

En este algoritmo, los puntos del scan actual son comparados frente a un modelo a priori que consta de los segmentos de línea. Es evidente que este planteamiento funciona casi en exclusividad en entornos que son principalmente poligonales.

Una pequeña e intuitiva modificación de este algoritmo consiste en extraer segmentos de línea del scan de referencia y usarlos como un modelo a priori. Esto hace que el algoritmo Cox sea adecuado para ser usado en *Scan Matching Self-Localization*.

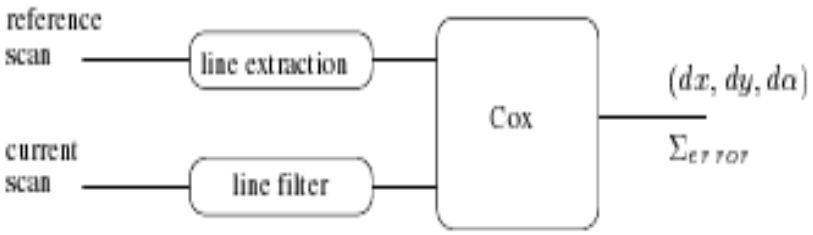

Figura 21: Extended Cox algorithm.

El algoritmo que hemos implementado en nuestro sistema hace algunas modificaciones más:

- Primero, el scan actual es filtrado mediante una línea de filtrado; esto es, los puntos del scan que no están en un segmento de línea son desechados. De esta forma, se reduce la cantidad de falsas asignaciones de puntos a segmentos de línea en situaciones en las que el entorno no es totalmente poligonal.
- En segundo lugar, un umbral fuertemente codificado ($d_{max} = 2000mm$) es usado para eliminar asignaciones que tienen una distancia mayor que d_{max}.
- Y, tercero, las asignaciones son ordenadas por sus distancias (las menores, primero). Sólo el primer 80 por ciento de las asignaciones se usarán para el matching, el resto son tratadas como fuera de rango y se desecharán.

3.5 COMBINED Algorithm

El algoritmo Combined, como bien indica su nombre, es una combinación del algoritmo Cox (desarrollado para entornos poligonales) con el algoritmo

IDC (que trabaja muy bien incluso en ambientes no poligonales) para construir un nuevo sistema que se desenvuelva bien en situaciones con estructuras poligonales y en las que sean entornos no poligonales.

Si hay suficientes puntos del scan que se encuentran en segmentos de línea entonces nuestro algoritmo Combined parece ser más rápido, ya que en este caso la implementación Cox es más efectiva. En otros casos, es el algoritmo *idc* el que se aprovechará de forma más eficiente.

Antes de realizar la correspondencia entre dos scans, se aplica un filtro llamado "projection filter" que elimina los puntos del scan, en ambos scans, que no son visibles desde la otra posición del scan. De esta manera, las asignaciones punto a segmento de línea o punto a punto falsas se minimizan.

Después de la exploración, el conjunto de scans es transferido y se calculan las posiciones de todos los scans al tiempo que se genera un mapa coherente de los scans. Además, el algoritmo construye una matriz de la estimación de la posición relativa para cada par de scans que tienen suficiente correspondencia. De esta manera, el algoritmo estima las posiciones del scan minimizando la función de covarianza de error.

Para la auto-localización, el robot simplemente toma un nuevo scan, usa la última posición estimada y busca el scan de referencia para equiparar ambos scans con el c*ombined scan matcher*.

3.5.1 Cambios en el entorno

Los algoritmos de emparejamiento de scans mencionados anteriormente permiten reconocer pequeñas modificaciones en el entorno del robot, por ejemplo, una puerta abierta o cerrada, armarios en una pared, etc. Sin embargo,

si los cambios son demasiado significativos, el algoritmo producirá resultados pobres. En este caso, los valores de los elementos de la matriz de covarianza de error serán grandes e indicarán una mala correspondencia.

Examinando la matriz de covarianza de error calculada para el algoritmo, es posible detectar situaciones en las que ha habido muchos cambios en el entorno, y, parte de los scans previamente almacenados, deberán ser sustituidos por unos actuales que concuerden mejor con la realidad.

3.6 PLAN DE ACCIÓN

Para cualquier proyecto que se desee realizar es muy importante elaborar un guión acerca de qué fases hay que llevar a cabo y en qué orden han de realizarse.

El problema que nos atañe ahora es, por tanto, la creación de un plan de desarrollo de nuestro sistema. Las tareas que han de llevarse a cabo se irán implementando de una forma escalonada.

En general, a la hora de elaborar un plan de acción han de tenerse en cuenta tres cuestiones principales: las tareas o procesos que deberán realizarse en el plan, el orden en el que se han de llevar a cabo dichas tareas y la necesidad o no de establecer puntos de retorno entre las diferentes etapas del plan. A medida que avanza un plan, a veces, es necesaria la reevaluación de parámetros o situaciones que se dieron por ciertas en una etapa anterior; más aún cuando se trabaja en un entorno real, el cual suele variar de una forma poco predecible.

Para el diseño del plan, vamos a crearnos un diagrama de bloques en el que se muestren las principales etapas a llevar a cabo siguiendo un orden de ejecución; el diagrama es el que se muestra en la siguiente figura:

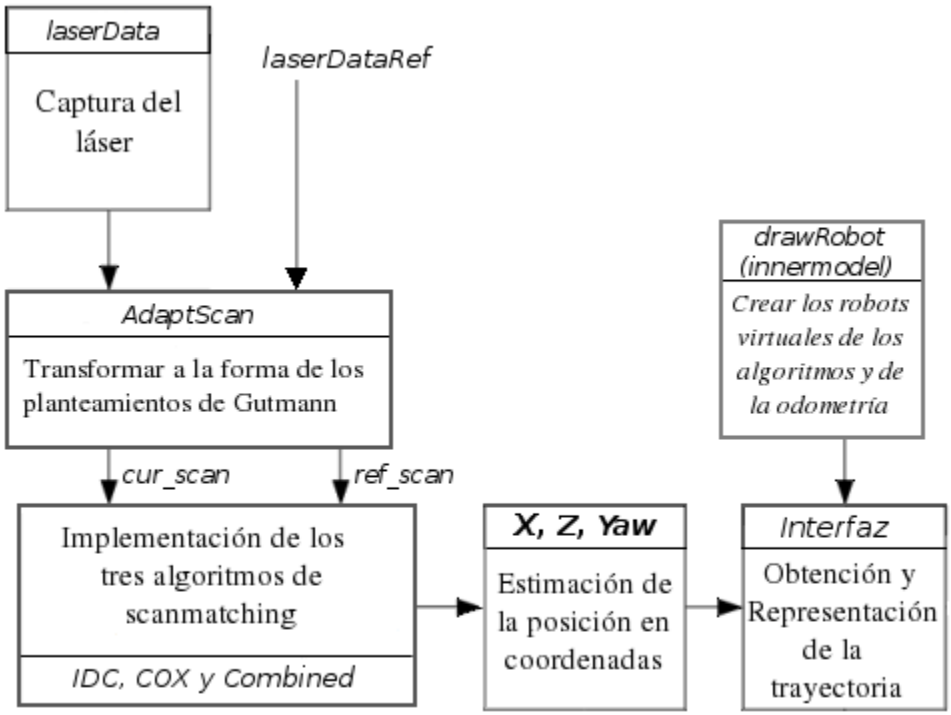

Figura 22: Diagrama de bloques de las diferentes etapas

En el siguiente capítulo se presentan y explican de una forma más detallada cada una de las fases que se van a realizar en la implementación de nuestro sistema.

Capítulo 4:
Diseño e Implementación del sistema

En este capítulo se tratará de reflejar el trabajo llevado a cabo para la consecución de nuestro sistema de estimación de la posición de un robot mediante correspondencia entre scans, ya denominado anteriormente *Scan Matching Self-Localization*.

Para la implementación de esta aplicación, hemos utilizado el lenguaje de programación C++ y el paradigma de la programación orientada a componentes. Con esto hemos conseguido una estructura sencilla, fácil de mantener y usar; proporcionándole, de esta forma, flexibilidad ante modificaciones y un alto nivel de reusabilidad.

4.1 PUNTO DE PARTIDA

Como se ha comentado previamente, el proyecto parte de una serie de componentes ya desarrollados, que interactuarán con nuestro componente para conformar el sistema que hemos implementado. En este apartado se van a comentar estos componentes y el componente genérico a partir del cual hemos elaborado nuestro scanmatchingComp, éste es el basicmonitorComp.

Todos ellos se encuentran en continua revisión y ampliación; puede dirigirse a "http://robocomp.wiki.sourceforge.net" para obtener la última versión disponible o ampliar la información.

4.1.1 basicmonitorComp

El componente basicmonitorComp constituye los cimientos sobre los que hemos construido nuestro scanmatchingComp. Es un componente muy básico que está preparado para permitir la captura instantánea del láser y de las cámaras y, al que hemos ido añadiendo funciones y completando su interfaz gráfica, para conseguir la implementación de nuestro sistema.

En la interfaz de este componente se reserva un espacio en el que se representan los ejes del que posteriormente será nuestro "*mundo*", el entorno que rodea al robot.

Todos los componentes del sistema van a trabajar con visión estereoscópica, que aunque para nuestro proyecto es meramente informativa, en la interfaz gráfica se nos muestra en tiempo real lo que está viendo el robot.

4.1.2 cameraComp

El componente cameraComp es otro de los primeros componentes de RoboComp. Su principal función es capturar vídeo y atender a las peticiones de imágenes que otros componentes realicen. Además, ha de adjuntar a las imágenes la configuración de la posición que tenía la torreta y el estado de la odometría cuando la imagen fue tomada. Tiene capacidad tanto para mandar imágenes de una cámara, como de dos a la vez.

Para evitar problemas de sincronización, cuando trabaja con dos cámaras, puede devolver ambas imágenes simultáneamente en una única llamada. Esta característica es básica para trabajar con visión estereoscópica.

Otra de las características interesantes de cameraComp es que permite trabajar con distintos tipos de cámaras: v4l2 (Video For Linux v2), IEEE 1394 (FireWire), o mediante una canalización Unix hacia Mplayer (lo que además de extender aun más el abanico de cámaras soportadas, permite la captura de imágenes desde ficheros de vídeo y recientemente desde el simulador de robots Gazebo).

4.1.3 joystickComp

El componente joystickComp es también un componente ya previamente creado que, aunque de relativa sencillez, es de vital importancia para el desarrollo de nuestro proyecto, ya que nos permite controlar los desplazamientos del robot a nuestro antojo.

Es el que nos ha permitido recorrer las trayectorias que se han llevado a cabo en la realización de los experimentos.

4.1.4 laserComp

El componente laserComp nos permite establecer la conexión con el láser y que éste nos devuelva los datos referentes a las mediciones que realiza en cada barrido: la distancia a cada punto del scan y el ángulo correspondiente. Para la auto-localización del robot, nuestro sistema de coordenadas nos muestra tres datos: las coordenadas del punto en el plano (X , Z) y el desvío angular, *Yaw*, con respeto a la posición inicial.

4.1.5 differentialrobotComp

El componente differentialrobotComp es el que controla la base robótica que, con una determinada velocidad de rotación de sus dos ruedas motrices, mueve al robot, dirigiéndolo con una tercera rueda de giro libre.

Mediante las dependencias atribuidas en nuestro *.xml* hacemos que, al arrancar este componente, se arranquen también el laserComp y el cameraComp.

4.2 CREACIÓN DE UN COMPONENTE GENÉRICO

Antes de proceder con la especificación de los componentes construidos se mostrará como crear un componente genérico. Cuando se incluye un nuevo componente en RoboComp se usa generalmente una herramienta que lo genera automáticamente. El generador de código también incluye el código necesario para crear una conexión a los componentes que se le especifiquen.

A los componentes que se generen automáticamente se le pueden añadir después todas las clases que sean necesarias. Además, muchas veces se desea añadir en el fichero de configuración algún parámetro específico y seguramente, modificar el fichero de interfaz por defecto (que no ofrece ningún método). De no usar el generador automático de código, la creación de componentes, incluso pequeños, sería bastante tediosa y susceptible a errores, ya que es necesario integrar el código del middleware Ice y el código propio del componente.

Por tanto, la creación del nuevo componente se llevará a cabo utilizando esta herramienta, siguiendo la estructura básica definida mediante esta creación.

Esta estructura es la que se observa en la figura siguiente:

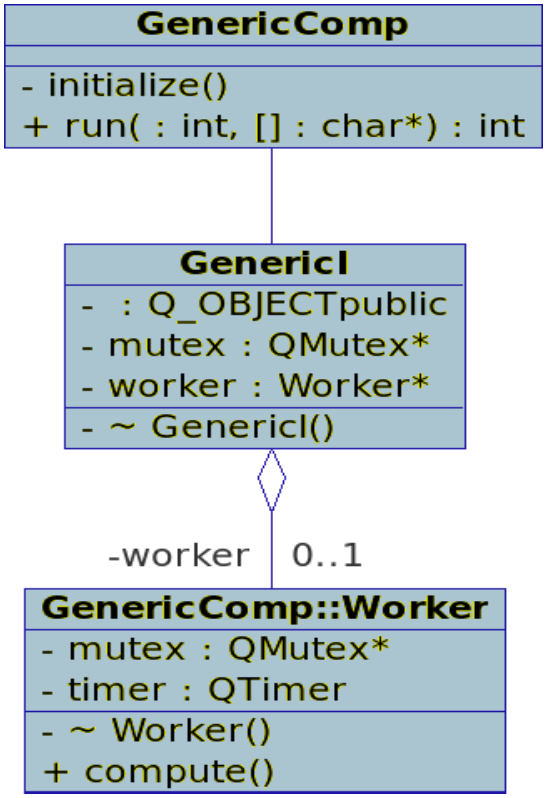

Figura 23: Diagrama de Clases de un componente genérico.

Como se puede observar en la figura, además del código de programa principal, el componente plantilla consta de tres clases: GenericComp, GenericI y Worker.

Cuando se usa el generador de código, éste modifica los nombres de las clases GenericComp y GenericI en función del nombre del componente, cambiando *Generic* por lo que corresponda.

Cada componente consta de, al menos, dos hilos de ejecución, siendo estos el hilo principal y un hilo separado para responder a las llamadas Ice. El último de estos dos hilos se crea cuando se instancia una interfaz Ice (a la clase creada

se le tiene que pasar como parámetro la clase que se hace cargo de las llamadas). El hilo principal de ejecución se encarga de la parte activa del componente, de hacer las llamadas Ice necesarias si las hay, y los cálculos asociados al comportamiento del componente.

Para facilitar la comprensión del código la clase GenericComp delega en Worker todo el trabajo y GenericI queda como una clase que, de forma transparente y asíncrona, responde a las llamadas remotas.

Las clases Worker y GenericI, que como ya hemos visto se encargan de la parte interna y externa respectivamente, se ejecutan en exclusión mutua gracias a un mutex que comparten. El bloqueo del mutex se realiza cada vez que GenericI recibe una llamada o que Worker va a cambiar alguna parte de su estado que pueda modificar las respuestas de GenericI a sus invocaciones remotas.

Como se puede ver en el diagrama, GenericI tiene acceso a la clase Worker. Dicha condición, que sólo se da en ese sentido, es necesaria porque generalmente toda la información relativa al estado del componente se guarda dentro de Worker.

4.2.1 Archivo *.xml*

Para poder arrancar los componentes y trabajar con ellos desde el managerComp de una forma cómoda e intuitiva, es necesario crear un archivo *.xml* que nos permita proceder desde la herramienta mangerComp y no como se hacía años atrás desde la terminal del P.C.

XML es un método para introducir datos estructurados en un fichero de texto. Cuando pensamos en "datos estructurados" pensamos en hojas de cálculo, parámetros de configuración, dibujos técnicos, etc. XML consiste en una serie

de reglas o pautas para planificar formatos de texto para tales datos, de manera que produzcan archivos que sean fácilmente generados y leídos por un ordenador; evitando problemas comunes como la falta de soporte o la dependencia de una determinada plataforma.

Nuestro archivo *.xml* (*figura* 24) establecerá los nodos que vamos a utilizar, los puertos a los que han de conectarse éstos, las dependencias entre los distintos nodos y cómo arrancar y detener la ejecución de los mismos.

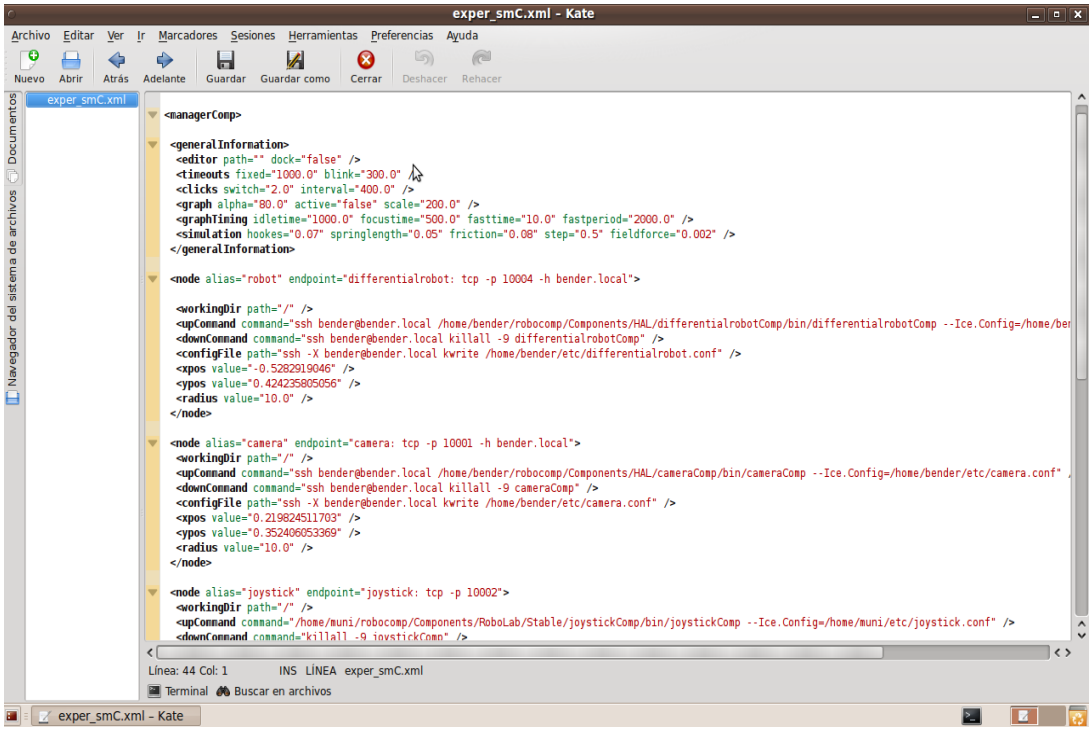

Figura 24: Archivo *exper_smC.xml*

4.3 DISEÑO DEL NUEVO COMPONENTE

El diseño del componente que queríamos implementar en nuestro sistema, al que hemos denominado scanmatchingComp, se ha acometido de forma estructurada, comenzando por hacer la llamada desde la considerada como función principal (*void compute*) a las funciones que tomarán los datos del láser (*getLaser*) y de las cámaras (getCamara).

Aunque lo explicaremos con detalle más adelante, ambas funciones serán llamadas sólo si se activa el botón correspondiente en la interfaz gráfica que hemos desarrollado – *pbLaser* y *pbCamara*, respectivamente-.

Como apunte, hay que decir que la mayoría del código que vamos a explicar a lo largo del capítulo se ha desarrollado principalmente en dos archivos: el *worker.cpp* y el *worker.h* de nuestro componente.

```
void Worker::compute ( )
{


      if (pbLaser->isChecked())  {
            getLaser ( cont );
            cont++;                  }

      if (pbCamara->isChecked())
            getCamara ( );
}
```

Tabla 3. Función principal

4.3.1 Interfaz Gráfica

Vamos a comenzar explicando un poco la interfaz que muestra por pantalla nuestro componente ya que, aunque cronológicamente no ha sido la primera fase de la realización del proyecto, si que vamos a estar hablando de ella en todo este capítulo y, por ello, queremos dar una visión general de su apariencia y su funcionamiento.

Primero introduciremos la herramienta Qt4 designer que nos permite construir interfaces gráficas de usuario desde los componentes Qt. Asimismo, permite diseñar *widgets*, *buttons* y *dialogs* de una forma sencilla y eficaz.

Una característica esencial de este diseñador es que facilita la tarea de conexión del interfaz con el código interno de la aplicación.

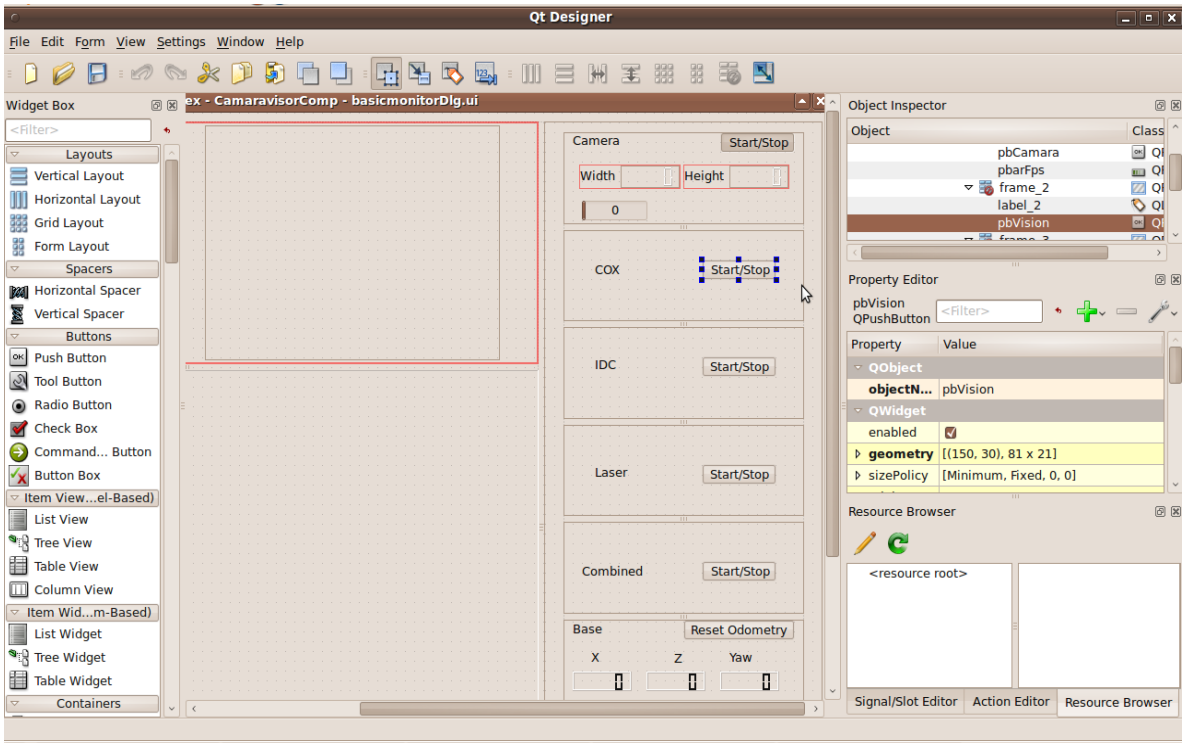

Figura 25: Diseño Interfaz mediante herramienta Qt4 designer

Seleccionando cada uno de los botones, widgets o ventanas de diálogo podemos cambiar los atributos y propiedades de los mismos a nuestro antojo y, al realizar el siguiente *cmake.*, los cambios ya se habrán aplicado al código interno del componente.

La interfaz que hemos diseñado para el componente nos permite utilizar de forma sencilla e intuitiva todas las funciones y algoritmos que integra nuestro componente.

A continuación se ofrece una imagen que nos muestra la apariencia final que tiene nuestra interfaz gráfica. Esta imagen pertenece a un instante de secuencia de un replay grabado con imagen a color, aunque para nuestros experimentos hayamos utilizado imágenes en blanco y negro por una mera cuestión de reducir el volumen de información a almacenar y transmitir:

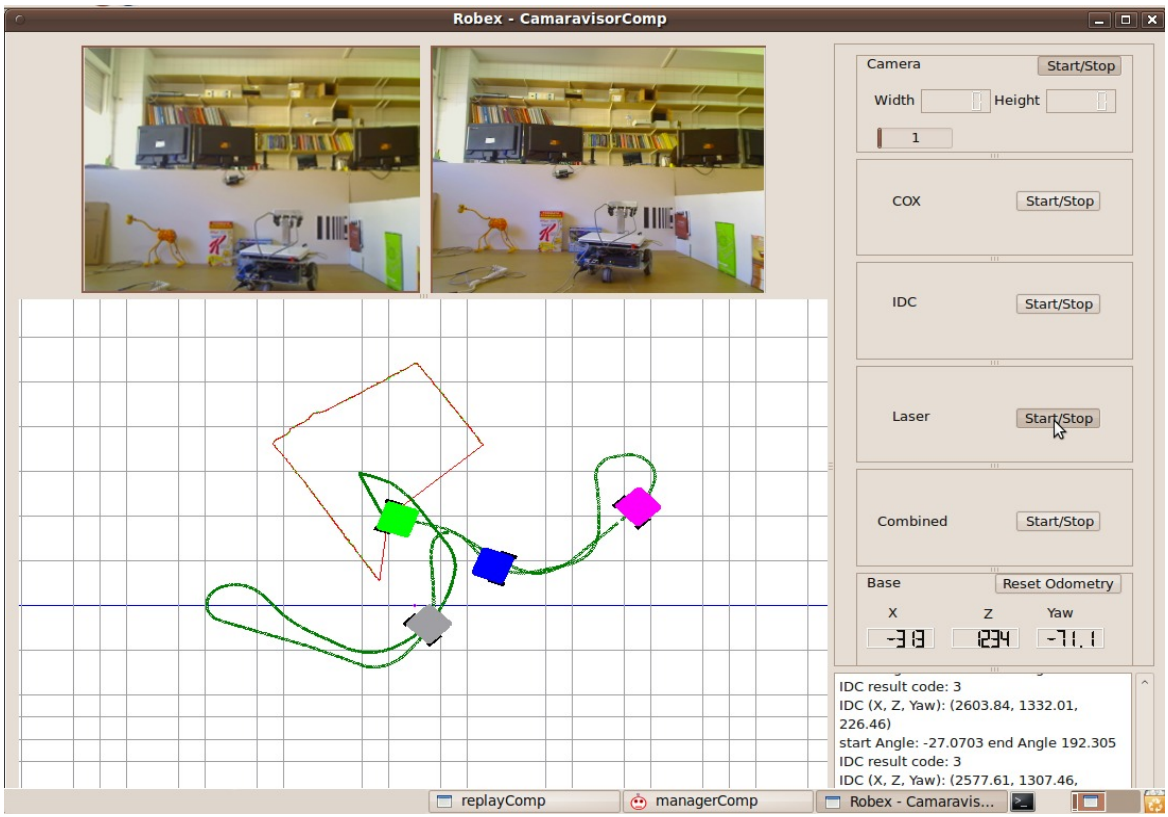

Figura 26: Interfaz Gráfica

4.3.2 Captura de Imágenes

Como ya hemos dicho anteriormente, queremos que nuestro sistema trabaje con visión estereoscópica, que aunque es meramente ilustrativa, nos parece interesante poder observar desde la interfaz gráfica lo que está viendo el robot en tiempo real.

Para ello había que tomar los datos de las cámaras – izquierda y derecha- por separado y reproducirlos en las ventanas destinadas a ello en la interfaz. Esta parte se implementa en la función *getCamara*, como puede apreciarse en las siguientes líneas de código:

```
void Worker::getCamara()
{
  try
	{
		_camara_prx->getYImage(0, imgV,  hState, bState_cam);  // en
camaraComp
		memcpy(imgI->bits(), &imgV[0], camParams.size);

		_camara_prx->getYImage(1, imgV,  hState, bState_cam);  // en
camaraComp
		memcpy(imgD->bits(), &imgV[0], camParams.size);

		qvisorI->setImage(imgD);
		qvisorI->drawCrossHair(Qt::yellow);
		qvisorD->drawCrossHair(Qt::yellow);
	}
```

Tabla 4. Función getCamara

4.3.3 Captura del Láser

En una primera fase, nos propusimos comenzar por la tarea a priori más sencilla, pero no por ello menos importante, ya que la base de nuestro proyecto sería tomar los datos del láser, para después trabajar con ellos y poder implementar los distintos algoritmos, tratando de obtener algún resultado satisfactorio.

Primero había que crear una función que tomase los valores del láser – los que ya conocemos como puntos del scan- y los almacenase; para lo cual se ha utilizado un vector que se irá rellenando. Después había que representar estos valores en el mundo (nuestro espacio gráfico reservado para dibujar el entorno del robot) y, así, poder observar gráficamente lo que el sensor láser estaba viendo a su alrededor.

Para comenzar hay que pasarle los valores del láser a nuestro vector *laserData*; estos valores serán la distancia hasta el punto en concreto del entorno y el ángulo. Posteriormente nosotros transformaremos estos dos valores en tres: un par de puntos que nos darán la posición en el espacio bidimensional (X y Z), con los que ya queda perfectamente definida la localización; y una desviación angular como dato de refuerzo (Yaw).

Seguidamente, hay que almacenar el scan actual, para que en el siguiente barrido del láser, éste pase a ser el de referencia y pueda realizarse una equiparación entre ambos scans. Lo que hacemos es pasarle al vector de referencia (*laserDataRef*) los datos del vector del scan actual (*laserData*). Todo esto puede observarse en el fragmento de código que se presenta a continuación; aunque se puede profundizar en el código, de forma más completa y rigurosa, en el anexo que se ha incluido al final:

```
 static int pantx = bState.x;
 static int pantz = bState.z;

 static int pantqx = bState.x;
 static int pantqz = bState.z;
       try
       {
       laserData = _laser_prx->getLaserData ();

        if (cont != 0)
              {
              for (unsigned int i=0; i<laserData.size(); i++)
              {
              QMat p = innerModel->laserToWorld(laserData[i].dist ,
laserData[i].angle);
              world->drawLine(QLine(pantx, pantz, p(0), p(2)), Qt::red);
              pantx = p(0);
              pantz = p(2);
       QMat q = innerModel->laserToWorld(laserDataRef[i].dist ,
laserDataRef[i].angle);
              world->drawLine(QLine(pantqx, pantqz, q(0), q(2)), Qt::green);
              pantqx = q(0);
              pantqz = q(2);
              }
        world->drawLine(QLine(pantx, pantz, bState.x, bState.z), Qt::red);
        pantx = bState.x;
        pantz = bState.z;
        world->drawLine(QLine(pantqx, pantqz, bState.x, bState.z), Qt::green);
        pantqx = bState.x;
        pantqz = bState.z;

        laserDataRef = laserData;
```

Tabla 5. Método Almacenar scan y Representar gráficamente

Veamos, a continuación, cómo se representa en la interfaz el scan actual y el de referencia de una captura del láser. La imagen se toma mientras el robot está girando bruscamente, para así ver más claramente ambos scans, ya que cada barrido del láser iguala el de referencia al actual. Puede observarse el scan de referencia pintado en color verde y el scan actual representado en color rojo:

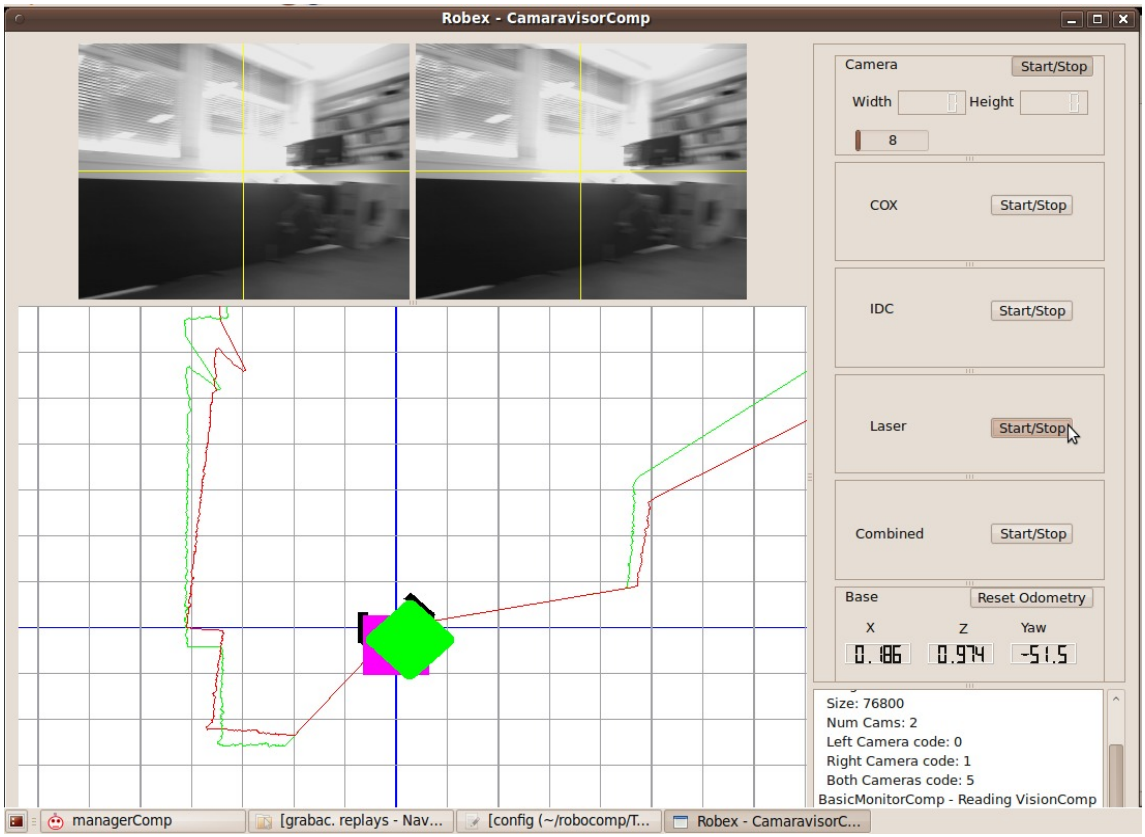

Figura 27: Scans actual y de referencia

Por otra parte, crearemos un contador para nuestra función *getLaser* (puede verse en la *Tabla 3. Función principal*), que se actualice con cada barrido del láser y nos permita así, tomar los datos de la odometría para la posición inicial sólo en el primer barrido que efectúe el sensor. A partir de este momento, el algoritmo correspondiente realizará él mismo la estimación de la posición del robot en tiempo real. La obtención de la odometría de la base es inmediata mediante la llamada a la función *innerModel->getBaseOdometry()*.

4.3.4 Adaptar la estructura scan

Los vectores *laserData* y *laserDataRef* no son de la forma "*struct scan"*, que se requiere para poder implementar los algoritmos de Jens-Steffen Gutmann cuyo funcionamiento hemos visto en el capítulo anterior. Lo que haremos es crear una función denominada *adaptScan* que transformará ambos vectores en la forma de Gutmann. La llamada a esta función se hará en *getLaser*, como puede verse en el siguiente extracto de código:

```
void Worker::getLaser(int cont)
{
try
{

   if (cont != 0)
         {

          current_scan = adaptScan(laserData);
          ref_scan = current_scan;
          }

   else // cont == 0
          {
           laserDataRef = laserData;
           ref_scan = adaptScan(laserDataRef);
```

Tabla 6. Llamada a adaptScan

La implementación en C++ de la función *adaptScan* se incluye con mayor detenimiento en el Anexo, capítulo 7.

4.3.5 Algoritmos de S*can Matching*

Como se vio en el capítulo 3, se van a implementar tres algoritmos – Cox, Combined e *idc*- para llevar a cabo la estimación de la posición mediante correspondencia entre scans.

Lo que haremos será controlar la utilización de cada uno de ellos desde la interfaz gráfica, utilizando los botones habilitados para ello. En el código la llamada de cada uno de los algoritmos se va a afrontar del siguiente modo:

```
if (cont != 0)
    {
       if (pbIDC->isChecked())
       IDCscanmatching();

       if (pbVision->isChecked())
       COXscanmatching();

       if (pbVision_2->isChecked())
       COMBINEDscanmatching();
     }

else // cont == 0
    {
      bStateIDC = bState;
      innerModelIDC->updatePropioception( bStateIDC, hState );

      bStateCOX = bState;
      innerModelCOX->updatePropioception( bStateCOX, hState );

      bStateCOMB = bState;
      innerModelCombined->updatePropioception( bStateCOMB, hState );
```

Tabla 7. Llamada diferentes algoritmos

En *contador=0*, es decir, la primera vez que activamos el láser y realiza su primer barrido, actualizamos la posición de los algoritmos con la que nos ofrece la odometría.

En la interfaz gráfica los botones que llamarán a los algoritmos pueden verse en la siguiente imagen:

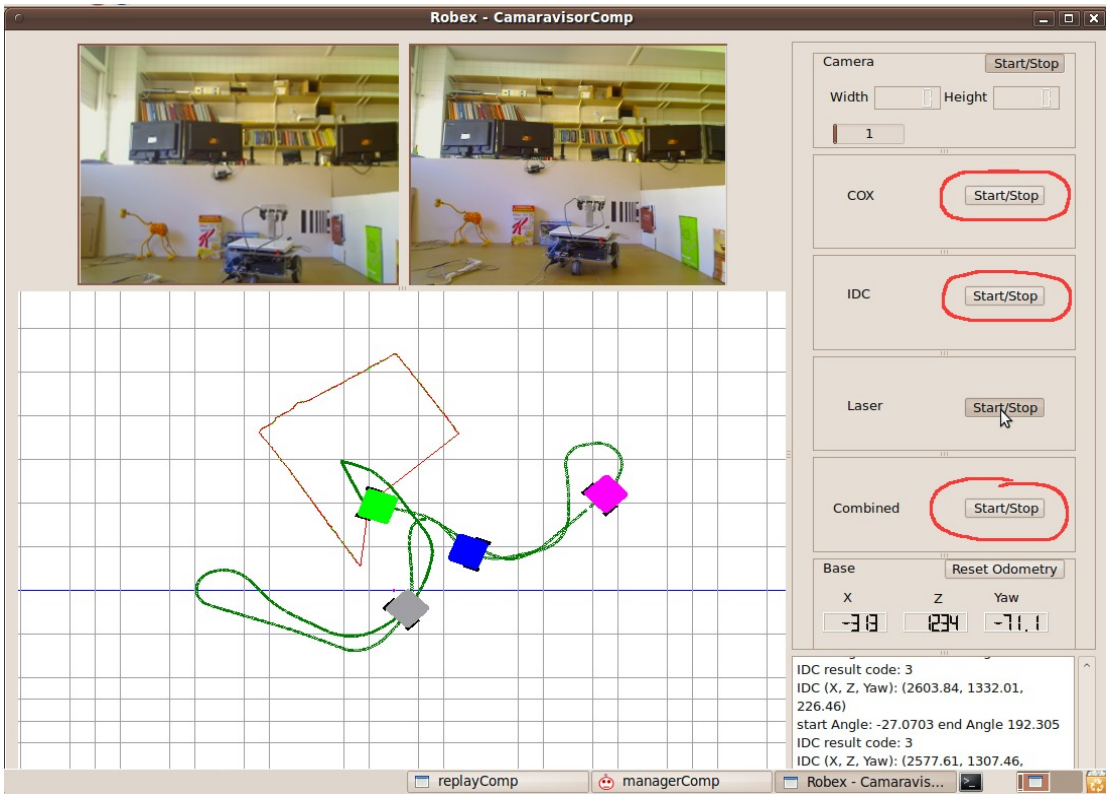

Figura 28: Botones para activar algoritmos

La implementación de los algoritmos resultó especialmente complicada desde el momento en que el sistema de coordenadas de los planteamientos de Jens-Steffen Gutmann era totalmente distinto del sistema que estábamos utilizando desde un principio dentro de la plataforma RoboComp.

Hubo que salvar dificultades como éstas y como, por ejemplo, averiguar en qué archivos y librerías se sustentaba la implementación de los algoritmos, para añadir éstos a nuestro componente. Algunos de los archivos que se han tenido que añadir, se enumeran a continuación para que se pueda apreciar el grado de complejidad y de interdependencias que existe en el sistema desarrollado:

./scan.c	./misc.c	./posegauss.c	./filter.c
./combined.c	./cox.c	./matrix.c	./idc.c
./linescan.c	./leastsquare.c	./sl.c	./romo.c

Cada una de las funciones empleadas en la implementación de los algoritmos de *scanmatching* se detalla minuciosamente en el capítulo 7, Anexo.

4.3.6 Obtención y Representación de la trayectoria

Por otro lado, también queremos que se pueda resetear la odometría cada vez que queramos, y así, poder tomar como punto inicial – coordenadas (0,0,0)- cualquier punto del entorno que se nos antoje. Para ello hemos creado un botón que nos permite hacerlo en el momento que queramos y, que lo que hace, es llamar a la función *resetOdometer*. Por defecto, cada vez que arrancamos nuestro componente, la odometría se inicializa a cero.

Esta parte que hemos comentado queda reflejada en las siguientes líneas de código, en las que como puede apreciarse el desvío angular, *lcdNumber_3*, se presentará en grados (°) en nuestra interfaz:

```
//Trying DifferentialRobotProxy
	try
	{
		_base_prx->ice_ping();
		_base_prx->resetOdometer();
		qDebug() << "DifferentialRobot is up";
	}

void Worker::compute ( )
{
	try {
	  _base_prx->getBaseState(bState);
	  innerModel->updatePropioception( bState, hState );

	  lcdNumber->display(bState.z);
	  lcdNumber_2->display(bState.x);
	  lcdNumber_3->display(bState.alfa*180/M_PI);
```

Tabla 8. Método Inicializar Odometría

Para la representación de los desplazamientos que realice el robot por su entorno, se han creado varios robots virtuales que se dibujarán en la pantalla como representación del robot RobEx y que se desplazarán por nuestro entorno virtual, nuestro mundo, siguiendo la trayectoria que le marque el algoritmo correspondiente. Esta implementación se ha llevado a cabo utilizando la clase *innermodel*, ya previamente creada, y llamando a la función *new RobotInnerModel( )*. Para la posterior representación gráfica en la interfaz, llamaremos a la función *drawRobot( )*, como puede verse en las siguientes líneas de código:

```
//inner continously updating model of robot

innerModel = new RobotInnerModel();
innerModel->init(QString::fromStdString(params.robotName));

innerModelIDC = new RobotInnerModel();
innerModelIDC->init(QString::fromStdString(params.robotName));

innerModelCOX = new RobotInnerModel();
innerModelCOX->init(QString::fromStdString(params.robotName));

innerModelCombined = new RobotInnerModel();
innerModelCombined->init(QString::fromStdString(params.robotName));

void Worker::compute( )
{
try {
  innerModel->updatePropioception( bState, hState );
  innerModelIDC->updatePropioception( bStateIDC, hState );
  innerModelCOX->updatePropioception( bStateCOX, hState );
  innerModelCombined->updatePropioception( bStateCOMB, hState );

      world->drawRobot( innerModel );
      world->drawRobot( innerModelIDC, Qt::magenta );
      world->drawRobot( innerModelCOX, Qt::blue );
      world->drawRobot( innerModelCombined, Qt::gray );
      world->drawAxis(Qt::gray, 8);
```

Tabla 9. Robots virtuales

Finalmente, una vez representados los robots virtuales en la interfaz, lo que se propuso fue que se movieran por el entorno virtual siguiendo la trayectoria que resulta de la estimación de la posición mediante cada uno de los algoritmos y mediante la odometría. De esta forma se pueden comparar resultados, ver dibujada la trayectoria que ha determinado cada uno, cómo se toman los giros, etc.

Para ello se ha desarrollado la función *drawOdometry( )*, que dibuja las trayectorias estimadas por cada uno de los métodos mediante la representación de pequeñas circunferencias una a continuación de otra. Esto lo hace la función *drawEllipse( )*, a la que se le pasan los valores de los vectores dinámicos que se han creado para almacenar la trayectoria del robot – *pathOdom*, *pathOdomIDC*, *pathOdomCOX* y *pathOdomCOMB*-. En el siguiente fragmento de código se expone brevemente este desarrollo:

```
void Worker::drawOdometry()

{
 for (int i = 0; i < n; i+=2)
   world->drawEllipse ( QPoint (pathOdom[i], pathOdom[i+1]), 15, 15,
   Qt::darkGreen );

 for (int i = 0; i < n_idc; i+=2)
   world->drawEllipse (QPoint(pathOdomIDC[i], pathOdomIDC[i+1]),15, 15,
   Qt::magenta);

 for (int i = 0; i < n_cox; i+=2)
   world->drawEllipse ( QPoint (pathOdomCOX[i], pathOdomCOX[i+1]), 15, 15,
   Qt::blue);

 for (int i = 0; i < n_comb; i+=2)
   world->drawEllipse ( QPoint (pathOdomCOMB[i], pathOdomCOMB[i+1]), 15, 15,
   Qt::darkGray );

}
```

Tabla 10. Dibujar trayectorias

Para ver la implementación de la función *drawOdometry( )* al completo, hay que dirigirse al capítulo 7, Anexo.

A priori, podía parecer que el número de tareas a llevar a cabo era pequeño, basándonos en el diagrama de bloques del capítulo anterior. Sin embargo, ya hemos visto que muchas de ellas tienen varias subdivisiones internas que equivalen a complejas funciones en el código.

No hay que olvidar que se trata de un sistema basado en componentes, lo que quiere decir que es necesario establecer una comunicación con cada uno de los componentes implicados en el proceso. Por tanto, hay que tener en cuenta posibles fallos en dicha comunicación.

Capítulo 5:
Experimentos y Resultados

En este quinto capítulo vamos a exponer las pruebas a las que se ha sometido a nuestro sistema de estimación de la posición de un robot móvil, presentando los resultados obtenidos y haciendo comparativas tanto estadísticas como gráficas de los mismos.

El objetivo inicial de este proyecto, como ya se ha expresado en capítulos anteriores, es el de estimar, de la forma más precisa posible, la localización de un robot móvil durante la navegación del mismo en su entorno, mostrando gráficamente las trayectorias calculadas por los diferentes algoritmos y la odometría en tiempo real.

A continuación, se van a detallar los experimentos que se han llevado a cabo junto con algunas imágenes de los mismos.

5.1 DESCRIPCIÓN EXPERIMENTOS

En la realización de las pruebas hemos utilizado los dos láser de que disponemos, el Hokuyo URG-04LX y el Hokuyo UMT-30LX, montados sobre los dos robots RobEx que ya hemos mostrado en capítulos anteriores – *bender* y *bender_II* respectivamente -.

Estas pruebas han sido realizadas en la sala del laboratorio de robótica de la UEX, Robolab, en el entorno con diferentes obstáculos que se mostraba en el capítulo 1º, apartado 1.3.2 (véase *figura* 7).

Se han realizado diez tests o recorridos de prueba para cada láser para poder comparar resultados y sacar algunas conclusiones. Cada test ha sido grabados mediante la herramienta replayComp – que se explicará en detalle más adelate - y se ha estimado la trayectoria seguida por el robot mediante cada uno de los tres algoritmos desarrollados a lo largo del proyecto y la odometría.

En todas las pruebas realizadas se ha efectuado un recorrido aleatorio pero de duración similar, en el que se efectúan giros rápidos, más lentos, se pasa por encima de cables, etc, simulando posibles maniobras y obstáculos que pudieran encontrarse en un entorno real. La característica distintiva de todos estos tests realizados es que todos los recorridos comienzan y finalizan en el mismo punto, marcado en el suelo con cinta adhesiva, como se muestra en las siguientes imágenes:

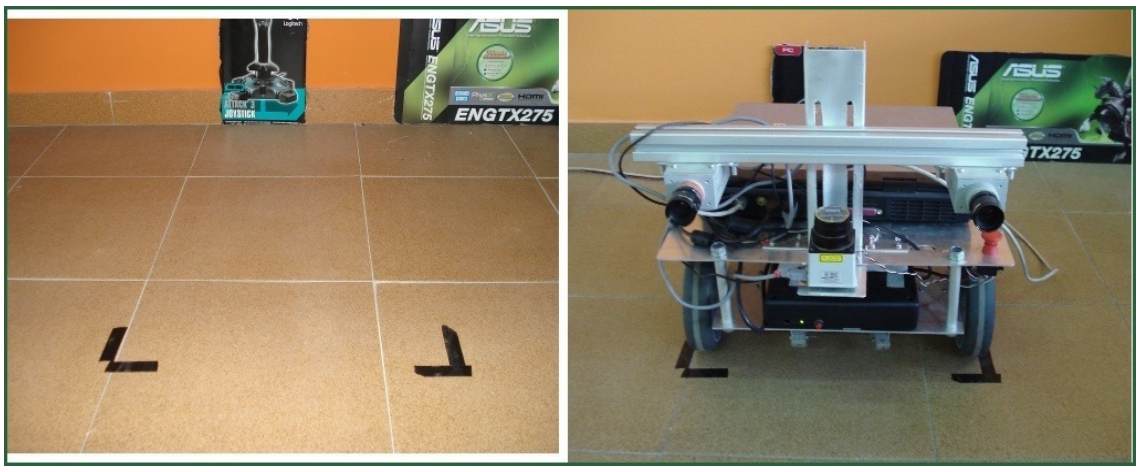

Figura 29: Marcas adhesivas (*izquierda*) y Robot situado en posición de inicio (*derecha*)

La experimentación va a consistir en realizar un recorrido partiendo de una posición inicial y regresando de nuevo a dicha posición; lo que queremos conseguir con esto es observar el error que introducen los algoritmos en la estimación de la posición y tras un seguimiento de la trayectoria del robot más o menos prolongado. Si todo transcurriera de forma exacta, las coordenadas finales estimadas por todos los algoritmos deberían ser: (X, Z, Yaw) = (0, 0, 0), al igual que en la posición inicial (*figura* 30).

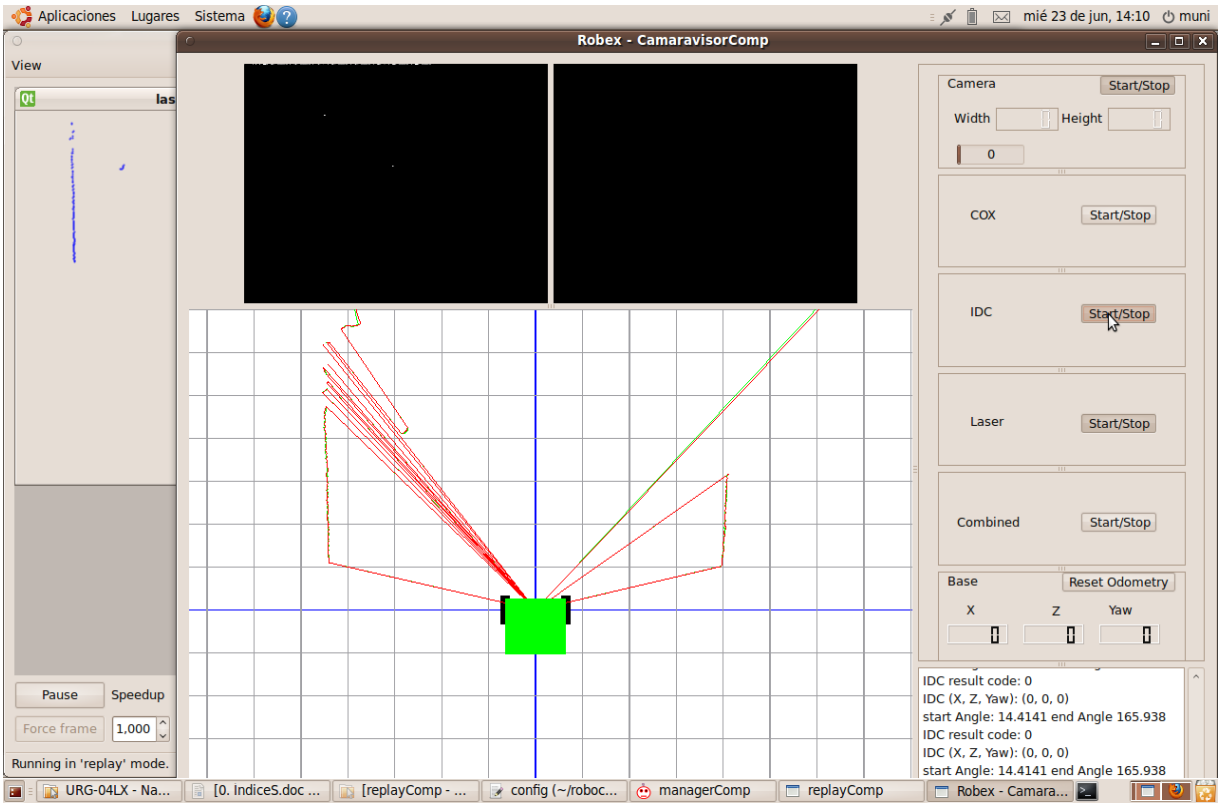

Figura 30: Coordenadas posición inicial

Pero sabemos que tal precisión es inalcanzable, ya que se van a introducir errores durante el proceso. Primero, hemos de tener en cuenta el error humano que introducimos nosotros mismos al tratar de dejar el robot en la misma

posición en la que inició su marcha. Este error ha sido medido in situ, de tal forma que pueda ser tenido en cuenta a la hora de conocer el error que van a introducir los algoritmos implementados. Cabe añadir que el error humano de aproximación a la posición de partida nunca ha excedido de los 5cm. y que el desvío angular máximo se ha calculado entorno a 5 º.

Con esta información, ahora el error que nos atañe es el que van a introducir cada uno de los tres algoritmos y la odometría.

El error que introduce cada algoritmo lo vamos a obtener como la diferencia entre la posición final medida experimentalmente (*error humano*) y la posición final que estima el algoritmo determinado. Esta posición final que nos ofrecen tanto la odometría como los algoritmos se muestra por pantalla mediante la interfaz y puede observarse en la imagen que se muestra a continuación:

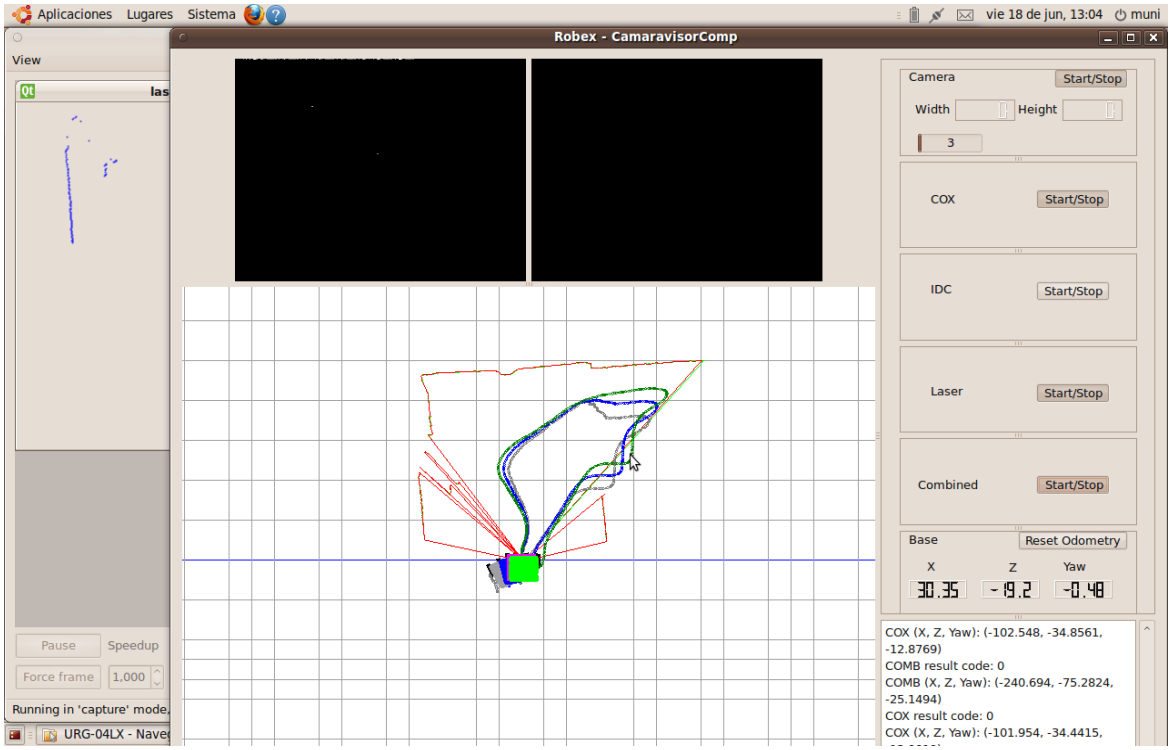

Figura 31: Coordenadas posición final

Los datos que nos muestran tanto la odometría como cada uno de los algoritmos se expresan siempre en milímetros (mm) para las coordenadas en el plano de la posición actual (X y Z) y en grados (º) para el desvío angular actual con respecto a la posición inicial (Yaw).

A continuación, procedemos a la exposición de los resultados obtenidos tras la realización de todas las pruebas con los diferentes algoritmos y una vez calculado ya el error que se introduce en cada una de ellas.

5.2 RESULTADOS OBTENIDOS

Para ofrecer una mejor visión de los resultados obtenidos vamos a hacer uso de la estadística descriptiva para poder apreciar mejor el significado de los resultados que se derivan de nuestros experimentos.

En teoría de probabilidad, la ***varianza*** de una variable aleatoria – en nuestro caso el error en la estimación de la posición, medido en milímetros - es una medida de su dispersión, definida como la media de las diferencias cuadráticas de n valores con respecto a su media aritmética (σ^2).
Está expresada en unidades distintas de las de la variable. Por ejemplo, si la variable mide una distancia en centímetros, la varianza se expresará en centímetros cuadrados.

$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \bar{x})^2$$

La ***desviación estándar*** o desviación típica (σ) se calcula como la raíz cuadrada de la varianza, constituyendo una medida estadística de dispersión expresada en las mismas unidades que la variable tratada.

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}(x_i - \overline{x})^2}$$

Utilizando estas dos medidas de dispersión vamos a elaborar unas tablas en las que se mostrará, primero, la media aritmética de los errores de cada uno de los métodos y, seguidamente, la varianza del dichos errores en el espacio. Estas tablas se muestran a continuación:

	Media del error (mm)			Varianza del error (cm^2)	
	X	Z	Yaw (°)	X	Z
Odometría	32,0	53,9	5,4	8,1	12,7
Combined	368,9	426,8	23,8	191,0	390,5
COX	169,4	214,7	14,2	57,9	126,6
IDC	115,2	74,3	6,8	178,1	41,3

Tabla 12. Errores medios y Varianza del error – Hokuyo URG-04LX

	Media error (mm)			Varianza del error (cm^2)	
	X	Z	Yaw	X	Z
Odometría	265,3	269,2	13,7	166,0	449,7
Combined	232,6	348,5	28,4	184,6	236,8
COX	217,5	291,5	21,1	197,7	331,2
IDC	198,3	211,8	11,3	126,5	347,8

Tabla 11. Errores medios y Varianza del error – Hokuyo UTM-30LX

No hemos incluido en estas tablas la varianza del error para la desviación angular (Y*aw*), ya que los grados al cuadrado no nos parecían una unidad de medida demasiado ilustrativa. En cambio, la varianza del error para las coordenadas en el espacio sí que han quedado reflejadas en la tabla, en centímetros cuadrados (cm^2) para hacernos una mejor idea del área de incertidumbre que ésta representa.

En las tablas del error medio y la varianza del error de cada algoritmo junto a la odometría (*tablas* 11 y 12) puede apreciarse que para el láser modelo UTM-30LX los resultados obtenidos para los algoritmos son bastante mejores que para el láser URG-04LX, es decir, el error es más pequeño. Esto puede ser debido a que el láser URG-04LX, al tener una distancia máxima de medida de cuatro metros, cuando un punto del scan se encuentra a una distancia superior a ésta, no toma ningún valor para este punto, no lo considera; con lo que esto afectará notablemente en la estimación de la posición.

En la siguientes tablas, se puede apreciar perfectamente como se dispersan los errores en la medida de las coordenadas (*X* y *Z*) y del ángulo (Y*aw*) para cada uno de los láser empleados, observando los valores que proporciona la desviación estándar:

	Desviación estándar (mm)		
	X	Z	Yaw (°)
Odometría	28,4	35,6	7,3
Combined	138,2	197,6	11,9
COX	76,1	112,5	6,9
IDC	133,5	64,3	6,8

Tabla 13. Desviación Estándar del error – Hokuyo URG-04LX

	Desviación estándar (mm)		
	X	Z	Yaw (°)
Odometría	128,9	212,1	11,5
Combined	135,9	153,9	15,0
COX	140,6	182,0	12,6
IDC	112,5	186,5	6,3

Tabla 14. Desviación Estándar del error – Hokuyo UTM-30LX

Hay que decir que para los robots diferenciales que hemos utilizado – plataforma RobEx -, al tener éstos una velocidad de desplazamiento bastante lenta, la odometría proporciona unos resultados bastante buenos y que, por eso, los resultados de los tres algoritmos implementados pueden parecer menos brillantes. Pero es importante saber que la odometría tiene serios problemas cuando hay irregularidades en el suelo o cuando se producen giros bruscos – como ya vimos en el capítulo primero - y que estos problemas se agravan al aumentar la velocidad de desplazamiento; mientras que los algoritmos de *scan matching* no se ven tan afectados por estas circunstancias.

También puede observarse que el algoritmo *idc* obtiene mejores resultados que los otros dos, es decir, el error medio es bastante menor. Pero si se observan detenidamente las tablas de la desviación estándar (*tablas* 13 y 14), puede apreciarse que ésta ya no tiene unos valores tan pequeños, lo que significa que hay una dispersión considerable de los valores de la variable. Esto se debe a que, al ser el *idc* un algoritmo de correspondencia punto a punto, a veces, cuando el láser percibe grandes cambios en el entorno, el algoritmo encuentra la mejor correspondencia punto a punto entre scans con una posición que no se corresponde con el desplazamiento real del robot, pero que sí produce el mejor emparejamiento posible entre scans. Esto se traduce en un salto de mayor o

menor tamaño del robot virtual en su trayectoria, que queda reflejado en la interfaz gráfica y que hace que la dispersión de los errores aumente – la desviación típica aumentará -.

Como solución hemos planteado unas restricciones, que se pueden aplicar a nivel de código, que consisten en regular y filtrar los parámetros que ofrece el algoritmo para estimar su posición. Esto es, si entre un scan y el siguiente refleja un cambio en el desplazamiento euclídeo del robot desproporcionado – del orden de varios centímetros ya lo sería, teniendo en cuenta la velocidad a la que se desplaza el robot -, entonces la variación que introduce el robot en las coordenadas de la nueva posición se toma como cero. De esta manera, se tratan de evitar estos saltos de posición que no se corresponden con la realidad, pero se aumenta la dependencia del sistema con respecto a la odometría.

En el soporte en formato digital adjuntado con la presente memoria del proyecto, pueden verse las tablas de los valores obtenidos en cada experimento en un archivo de hoja de cálculo. También se adjuntan las imágenes, así como un replay de la realización de alguna de las pruebas.

5.3 Empleo de replayComp

Como ya sabemos, replayComp es una aplicación de RoboComp que es capaz de grabar y simular la actividad de algunos de los principales componentes, encargándose de grabar los datos producidos por los sensores para su posterior reproducción.

Es extremadamente útil cuando no disponemos del hardware necesario o éste da problemas, para poder seguir trabajando con las grabaciones ya hechas.

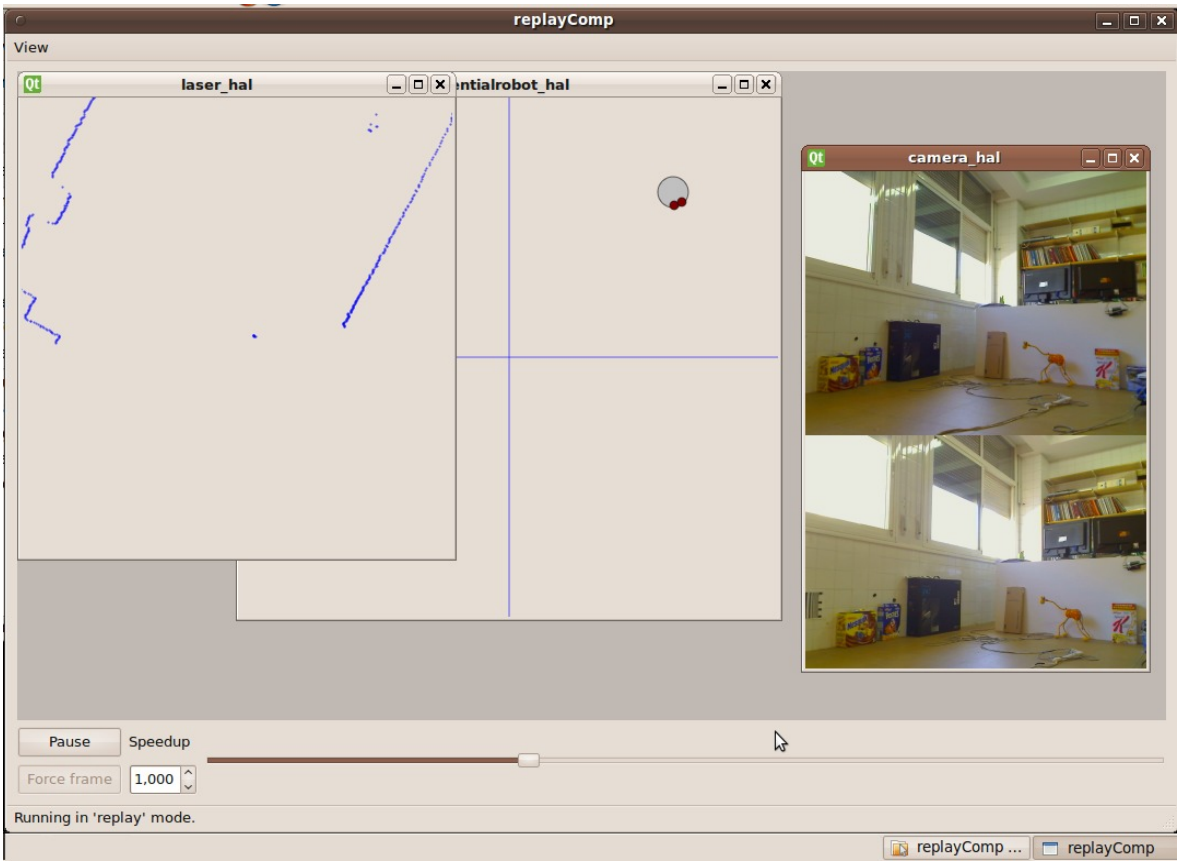

Figura 32: Aspecto gráfico de replayComp

La herramienta replayComp ha sido utilizada en la realización de los experimentos para grabar cada uno de los tests que hemos realizado con nuestro sistema; y así, poder repetir algún proceso que fuera necesario, poder utilizar los algoritmos juntos y cada uno por separado, poder repetir las pruebas todas las veces que se desee sin tener que establecer conexión con el robot; o incluso poder simular desde casa las mismas trayectorias que se han ejecutado en el laboratorio.

Capítulo 6:
Conclusiones y Líneas de trabajo futuras

Este capítulo va a estar dedicado a la exposición de las principales conclusiones extraídas del proyecto y a la presentación de posibles líneas de trabajo futuras para continuar desarrollando el sistema creado.

A continuación, comenzaremos por comentar las conclusiones que se pueden extraer tanto de los resultado como de la propia elaboración de este proyecto.

6.1 CONCLUSIONES EXTRAÍDAS

Mediante el presente Proyecto Fin de Carrera se ha presentado la implementación de un sistema que permita la estimación de la posición de un robot móvil durante la navegación, basándose en los algoritmos de *scan matching* que se han explicado en profundidad en los capítulos anteriores.

Como resultado se obtiene una herramienta, que consigue la estimación en tiempo real de la posición de un robot autónomo móvil cualquiera, mediante técnicas de correspondencia entre scans – *scan matching* - y que cuenta con un alto grado de reusabilidad y mantenibilidad, gracias a la utilización, para el desarrollo de este sistema, del lenguaje C++ y del paradigma de programación orientada a componentes.

Podemos concluir que, para la auto-localización de robots móviles lo óptimo es utilizar, durante la navegación, sistemas híbridos que utilicen tanto la odometría como algoritmos basados en *scan matching;* pudiendo, de esta forma, estimar la posición de los robots con el método más adecuado en cada situación que lo requiera; es decir, con la implementación que resulte más precisa en cada instante. Hay que añadir que, para obtener unos resultados con buena precisión, lo más adecuado es usar conjuntamente ambas estimaciones, pero manteniendo siempre una independencia entre ambas.

Lo más importante de todo lo que se ha conseguido es que se deja hecha una herramienta de estimación de la posición para robots móviles y que, de ahora en adelante, podrá ser probada en robots diferentes para ver su funcionamiento y podrá seguir siendo desarrollada para optimizar aún más su funcionamiento, por ejemplo, efectuando más restricciones o retocando internamente los algoritmos para que devuelvan parámetros más precisos.

Entre sus posibles aplicaciones, podríamos destacar su uso para el desarrollo de sistemas inteligentes de vigilancia, en las industrias para optimización del espacio y gestión automática de almacenes, o como ayuda para orientación en lugares como museos, hospitales, etc.

6.2 TRABAJOS FUTUROS

A continuación, se exponen algunas de las posibles líneas de trabajo futuras que pueden derivarse de este proyecto:

1. Optimizar la precisión del sistema, continuando con el desarrollo del mismo, por ejemplo, efectuando más restricciones en la correspondencia entre scans. Además, se podría enfocar esta mejora estudiando, por ejemplo, la velocidad de desplazamiento y de giro del robot que se utilice, para conocer los valores de máximo desplazamiento posible entre cada scan.

2. Perfeccionar la implementación interna de los algoritmos de *scan matching*, retocando sus parámetros internos para que devuelvan resultados más precisos y fiables.

3. Puede ser interesante utilizar el componente creado para este sistema, el scanmatchingComp, junto a otros componentes de localización que ya están creados e incluidos en RoboComp.

Finalmente, destacar que, con la realización de este proyecto fin de carrera, he tenido que aprender nuevos conceptos, no sólo en el ámbito de la programación orientada a componentes y la localización basada en los scans de un láser, sino también a la hora de afrontar complejos problemas que siempre tienen una solución.

Por ello, concluyo diciendo que me ha resultado muy satisfactoria esta experiencia práctica que supone la elaboración del Proyecto Fin de Carrera, que me ha formado para poder solventar los problemas que puedan surgirme, de ahora en adelante, en mi – esperemos no muy lejano- futuro profesional.

6.3 Cronología de las etapas de desarrollo del proyecto

La elaboración de este proyecto ha llevado un tiempo de desarrollo de cuatro meses y medio de atención única y exclusiva al mismo.
Todo empezó tras haber establecido las pautas del proyecto que íbamos a realizar, en enero del presente año, antes de los exámenes de las dos últimas asignaturas que me quedaban por conseguir.

A partir del 15 de febrero ya comenzó el trabajo en la sala de Robolab, empleando el primer mes en la instalación y adecuación de todos los componentes de RoboComp, así como en el perfeccionamiento de los conocimientos acerca del lenguaje de programación C++ y sobretodo, del paradigma orientado a componentes.

A mediados de marzo se comenzó con la implementación del código y el estudio de los algoritmos de *scan matching* y de cómo incluirlos en nuestro sistema, que hasta la consecución de la herramienta con interfaz gráfica que se ha presentado en la presente memoria, nos ha llevado un tiempo de dos meses y medio de trabajo, con una dedicación exclusiva a la misma.

A principios de junio se comenzó con la redacción de la memoria, a la par que se iban dando los últimos retoques al sistema y se iban realizando los experimentos; y esto, nos ha llevado le resto del tiempo hasta finales de junio que se ha concluido el proyecto fin de carrera.

Capítulo 7: Anexos

Como ya hemos comentado, en este penúltimo apartado de la memoria se pretende incluir las partes del código que se han explicado en el capítulo cuarto y que por extensión no quedaron reflejadas en el mismo.

7.1 Función *getLaser*

```
void Worker::getLaser(int cont)
{
        static int pantx = bState.x;
        static int pantz = bState.z;

        static int pantqx = bState.x;
        static int pantqz = bState.z;
        try
        {
         laserData = _laser_prx->getLaserData ();

         if (cont != 0)
           {
                for (unsigned int i=0; i<laserData.size(); i++)
                {
  QMat p = innerModel->laserToWorld( laserData[i].dist , laserData[i].angle );
                   world->drawLine(QLine(pantx, pantz, p(0), p(2)), Qt::red);
                   pantx = p(0);
                   pantz = p(2);
  QMat q = innerModel->laserToWorld(laserDataRef[i].dist,laserDataRef[i].angle);
                   world->drawLine(QLine(pantqx, pantqz, q(0), q(2)), Qt::green);
                   pantqx = q(0);
                   pantqz = q(2);
                 }
               world->drawLine(QLine(pantx, pantz, bState.x, bState.z), Qt::red);
               pantx = bState.x;
               pantz = bState.z;
```

```
            world->drawLine(QLine(pantqx, pantqz, bState.x, bState.z), Qt::green);
                pantqx = bState.x;
                pantqz = bState.z;

        // adaptScan() convert the current scan reading to Gutmann's scan format


            current_scan = adaptScan(laserData);
            laserDataRef = laserData;
            ref_scan = current_scan;

            if (pbIDC->isChecked())
            IDCscanmatching();

            if (pbVision->isChecked())
            COXscanmatching();

            if (pbVision_2->isChecked())
            COMBINEDscanmatching();
          }

          else // cont == 0
          {
            laserDataRef = laserData;
            ref_scan = adaptScan(laserDataRef);

            bStateIDC = bState;
            innerModelIDC->updatePropioception( bStateIDC, hState );
            bStateCOX = bState;
            innerModelCOX->updatePropioception( bStateCOX, hState );
            bStateCOMB = bState;
            innerModelCombined->updatePropioception( bStateCOMB, hState );

            dx=0.0, dy=0.0, da=0.0;         // Error en (X,Z,Yaw) posicion del robot
            dxx=0.0, dyy=0.0, daa=0.0;    // Acumulacion del error

            dxCOX=0.0, dyCOX=0.0, daCOX=0.0;
            dxxCOX=0.0, dyyCOX=0.0, daaCOX=0.0;

            dxCOMB=0.0, dyCOMB=0.0, daCOMB=0.0;
            dxxCOMB=0.0, dyyCOMB=0.0, daaCOMB=0.0;
          }
        }

        catch(const Ice::Exception& ex)
        {
              qDebug() << "Laser - No connection to LaserComp...";
        };
}
```

Tabla 15. Función getLaser

7.2 Función *adaptScan*

```
struct Scan *Worker::adaptScan(RoboCompLaser::TLaserData laserData)
{

   struct Scan scan_tmp, *scan = NULL;
   struct ScanPoint *sp = NULL;
   struct PoseGauss *pos = &scan_tmp.sc_Pos;
   struct Pose scanpos;

   long i;
   scan_tmp.sc_Num = laserData.size();

std::cout << "sc_Num (number of readings):" << laserData.size() << std::endl;

   scan = AllocScan(scan_tmp.sc_Num);
     if(scan == NULL)
     return(NULL);

   pos->pg_Mean.p_X  =  innerModel->getBaseZ();
   pos->pg_Mean.p_Y  = -innerModel->getBaseX();
   pos->pg_Mean.p_A  = M_PI/2 - innerModel->getBaseAngle();

   pos->pg_Sigma = Matrix3Zero;

   scan_tmp.sc_ScannerPos.p_X = 0;
   scan_tmp.sc_ScannerPos.p_Y = 0;
   scan_tmp.sc_ScannerPos.p_A = 0;

   *scan = scan_tmp;
   scan->sc_Id = scanId++;

   sp = scan->sc_Points;
   GetScanPose(scan, &scanpos);

   // read scan points
   for(i=0; i < scan_tmp.sc_Num; i++)
       {
        QMat p = innerModel->laserToBase( laserData[i].dist , laserData[i].angle );
        sp->sp_xl =  p(2);
        sp->sp_yl = -p(0);

  QMat pglobal = innerModel->laserToWorld(laserData[i].dist, laserData[i].angle);
         sp->sp_X =  pglobal(2);
         sp->sp_Y = -pglobal(0);

         sp->sp_A = M_PI/2 - laserData[i].angle;
         sp->sp_D = laserData[i].dist;

         sp->sp_Weight = 1;
```

```
        sp->sp_LineNum = -1;
        sp++;
       }

   scan->sc_Num = sp - scan->sc_Points;

   ScanRemoveOutliersFilter(scan);
   SortScanPoints(scan->sc_Points, scan->sc_Num);


   //  determine start & end angles
   if (scan->sc_Num > 0)
   {
       scan->sc_StartA =scan->sc_Points[0].sp_A;
       scan->sc_EndA =scan->sc_Points[scan->sc_Num-1].sp_A;
    }

    return(scan);
}
```

Tabla 16. Función *adaptScan*

7.3 Función *Algoritmo idc*

```
void Worker::IDCscanmatching()
{

       MoveScan(current_scan, dxx, dyy);
       RotateScan(current_scan, daa);

 textOut->append( QString("start Angle: %1 end Angle %2").arg(current_scan
 ->sc_StartA*180/M_PI).arg(current_scan->sc_EndA*180/M_PI));

              int result;
              result = slIDCScanMatch(ref_scan, current_scan, &match, &error);

              textOut->append( QString("IDC result code: %1").arg(result));

       PoseGaussTransformToGlobal(&ref_scan->sc_Pos, &match, &matchpos);

       dx=matchpos.pg_Mean.p_X - current_scan->sc_Pos.pg_Mean.p_X;
       dy=matchpos.pg_Mean.p_Y - current_scan->sc_Pos.pg_Mean.p_Y;
       da=matchpos.pg_Mean.p_A - current_scan->sc_Pos.pg_Mean.p_A;

              if ((sqrt(pow(dxx,2) + pow(dyy,2)) < 500))
              {
                dxx=dxx+dx;
```

```
                dyy=dyy+dy;
                daa=daa+da;
               }
              else
                dx = 0; dy = 0; da =0;

              MoveScan(current_scan, dx, dy);
              RotateScan(current_scan, da);

              bStateIDC.z = current_scan->sc_Pos.pg_Mean.p_X; // z = x
              bStateIDC.x = -current_scan->sc_Pos.pg_Mean.p_Y; // x = -y
              bStateIDC.alpha = M_PI/2 - current_scan->sc_Pos.pg_Mean.p_A;

              // print results in UI
        textOut->append( QString("IDC (X, Z, Yaw): (%1, %2, %3)").arg(bStateIDC.x)
        .arg(bStateIDC.z).arg(bStateIDC.alpha*180/M_PI));

              laserDataRef = laserData;
              ref_scan = current_scan;

              Odometry = innerModel->getBaseOdometry();
              ref_Odometry = innerModel->getBaseOdometryAnt();

              std::cout << " cur_odometry: (" <<  Odometry(0) << ", " <<
Odometry(1) << ", " << Odometry(2)*180/M_PI << ")" << std::endl;
              std::cout << " ref_odometry: (" <<  ref_Odometry(0) << ", " <<
ref_Odometry(1) << ", " <<  ref_Odometry(2)*180/M_PI << ")" << std::endl;

              printResult(ref_scan, current_scan, &match, error);

}
```

Tabla 17. Función *IDCscanmatching*

7.4 Función *Algoritmo COX*

```
void Worker::COXscanmatching()
{

        MoveScan(current_scan, dxxCOX, dyyCOX);
        RotateScan(current_scan, daaCOX);

        int result;
        result = slCoxScanMatch(ref_scan, current_scan, &matchCOX, &errorCOX);
        textOut->append( QString("COX result code: %1").arg(result));

  PoseGaussTransformToGlobal(&ref_scan->sc_Pos, &matchCOX, &matchposCOX);
```

```
    dxCOX=matchposCOX.pg_Mean.p_X - current_scan->sc_Pos.pg_Mean.p_X;
    dyCOX=matchposCOX.pg_Mean.p_Y - current_scan->sc_Pos.pg_Mean.p_Y;
    daCOX=matchposCOX.pg_Mean.p_A - current_scan->sc_Pos.pg_Mean.p_A;

    dxxCOX = dxxCOX+dxCOX;
    dyyCOX = dyyCOX+dyCOX;
    daaCOX = daaCOX+daCOX;

        MoveScan(current_scan, dxCOX, dyCOX);
        RotateScan(current_scan, daCOX);

        bStateCOX.z = current_scan->sc_Pos.pg_Mean.p_X; // z = x
        bStateCOX.x = -current_scan->sc_Pos.pg_Mean.p_Y; // x = -y
        bStateCOX.alpha = M_PI/2 - current_scan->sc_Pos.pg_Mean.p_A;

                // print results in UI
  textOut->append( QString("COX (X, Z, Yaw): (%1, %2, %3)").arg(bStateCOX.x)
  .arg(bStateCOX.z).arg(bStateCOX.alpha*180/M_PI));

        laserDataRef = laserData;
        ref_scan = current_scan;
        Odometry = innerModel->getBaseOdometry();
        ref_Odometry = innerModel->getBaseOdometryAnt();
}
```

Tabla 18. Función *COXscanmatching*

7.5 Función *Algoritmo Combined*

```
void Worker::COMBINEDscanmatching()
{

        MoveScan(current_scan, dxxCOMB, dyyCOMB);
        RotateScan(current_scan, daaCOMB);

        int result;
        result = slCombinedScanMatch(ref_scan, current_scan, &matchCOMB,
        &errorCOMB);
        textOut->append( QString("COMB result code: %1").arg(result));

        PoseGaussTransformToGlobal(&ref_scan->sc_Pos, &matchCOMB,
        &matchposCOMB);

  dxCOMB=matchposCOMB.pg_Mean.p_X - current_scan->sc_Pos.pg_Mean.p_X;
  dyCOMB=matchposCOMB.pg_Mean.p_Y - current_scan->sc_Pos.pg_Mean.p_Y;
```

```
 daCOMB=matchposCOMB.pg_Mean.p_A - current_scan->sc_Pos.pg_Mean.p_A;

      dxxCOMB = dxxCOMB+dxCOMB;
      dyyCOMB = dyyCOMB+dyCOMB;
      daaCOMB = daaCOMB+daCOMB;

      MoveScan(current_scan, dxCOMB, dyCOMB);
      RotateScan(current_scan, daCOMB);

      bStateCOMB.z = current_scan->sc_Pos.pg_Mean.p_X; // z = x
      bStateCOMB.x = -current_scan->sc_Pos.pg_Mean.p_Y; // x = -y
      bStateCOMB.alpha = M_PI/2 - current_scan->sc_Pos.pg_Mean.p_A;

            // print results in UI
 textOut->append( Qstring("COMB (X,Z,Yaw): (%1, %2, %3)").arg(bStateCOMB.x)
 .arg(bStateCOMB.z).arg(bStateCOMB.alpha*180/M_PI));

            laserDataRef = laserData;
            ref_scan = current_scan;
            Odometry = innerModel->getBaseOdometry();
            ref_Odometry = innerModel->getBaseOdometryAnt();
}
```

Tabla 19. Función *Combinedscanmatching*

7.6 Función *drawOdometry*

```
void Worker::drawOdometry()
{
 int n, n_idc, n_cox, n_comb;

 QMat current_p(2), previous_p(2);
 QMat current_IDC(2), previous_IDC(2);
 QMat current_COX(2), previous_COX(2);
 QMat current_COMB(2), previous_COMB(2);

 current_p(0) = bState.x;
 current_p(1) = bState.z;
 current_IDC(0) = bStateIDC.x;
 current_IDC(1) = bStateIDC.z;
 current_COX(0) = bStateCOX.x;
 current_COX(1) = bStateCOX.z;
 current_COMB(0) = bStateCOMB.x;
 current_COMB(1) = bStateCOMB.z;
```

```
 previous_p(0) = _lastbState.x;
 previous_p(1) = _lastbState.z;
 previous_IDC(0) = _lastbStateIDC.x;
 previous_IDC(1) = _lastbStateIDC.z;
 previous_COX(0) = _lastbStateCOX.x;
 previous_COX(1) = _lastbStateCOX.z;
 previous_COMB(0) = _lastbStateCOMB.x;
 previous_COMB(1) = _lastbStateCOMB.z;

 n = pathOdom.size();
 pathOdom.resize(n+2);

 n_idc = pathOdomIDC.size();
 pathOdomIDC.resize(n_idc+2);

 n_cox = pathOdomCOX.size();
 pathOdomCOX.resize(n_cox+2);

 n_comb = pathOdomCOMB.size();
 pathOdomCOMB.resize(n_comb+2);

 pathOdom[n]   = bState.x;
 pathOdom[n+1] = bState.z;

 pathOdomIDC[n_idc]   = bStateIDC.x;
 pathOdomIDC[n_idc+1] = bStateIDC.z;

 pathOdomCOX[n_cox]   = bStateCOX.x;
 pathOdomCOX[n_cox+1] = bStateCOX.z;

 pathOdomCOMB[n_comb]   = bStateCOMB.x;
 pathOdomCOMB[n_comb+1] = bStateCOMB.z;

   for (int i = 0; i < n; i+=2)
 world->drawEllipse (QPoint(pathOdom[i], pathOdom[i+1]),15,15,Qt::darkGreen);

   for (int i = 0; i < n_idc; i+=2)
 world->drawEllipse (Qpoint (pathOdomIDC[i],pathOdomIDC[i+1]), 15, 15,
 Qt::magenta);

    for (int i = 0; i < n_cox; i+=2)
 world->drawEllipse ( QPoint (pathOdomCOX[i], pathOdomCOX[i+1]), 15, 15,
 Qt::blue);

    for (int i = 0; i < n_comb; i+=2)
 world->drawEllipse ( QPoint(pathOdomCOMB[i], pathOdomCOMB[i+1]), 15,15,
 Qt::darkGray );
}
```

Tabla 20. Función *COXscanmatching*

Capítulo 8:
Bibliografía

[1] Capek, K. - " Rossum's Universal Robots ", 1921.

[2] Gutmann, J.S. y Schlegel, C. - " AMOS: Comparison of Scan Matching Approaches for Self-Localization in Indoor Environments "

[3] Cox, I. J. " Blanche- An experiment in guidance and navigation of an autonomous robot vehicle " Robotics and Automation, IEEE Transactions

[4] Lara, C. , Romero, L. y Calderón, F. " A Robust Iterative Closest Point Algorithm with Augmented Features "

[5] Pilar Bachiller Burgos - " *Percepción dinámica del entorno en un robot móvil* " Tesis doctoral, 2008.

[6] http://en.wikipedia.org/wiki/Differential_wheeled_robot

[7] http://www.hokuyo-aut.jp/02sensor/07scanner/urg_04lx.html

[8] Minguez, J. y Lamiraux, F. - " *Metric-Based Scan Matching Algorithms for Mobile Robot Displacement Estimation* "

[9] http://www.hokuyo-aut.jp/02sensor/07scanner/utm_30lx.html

[10] Joyanes, L. - " *Fundamentos de programación. Algoritmos, estructuras de datos y objetos* ". 2003.

[11] RoboComp - http://sourceforge.net/apps/mediawiki/robocomp/index.php?title=Main_Page#Components.

[12] Robolab. RobEx arena - http://robexarena.com.

[13] http://www.vitutor.com/estadistica/descriptiva/

[14] Ramón Cintas Peña - “ Técnicas visuales de reconocimiento de objetos en robots móviles ”. Proyecto Fin de Carrera 2009.

[15] Creative Commons España - http://es.creativecommons.org.

[16] Página del laboratorio de robótica de la UEX - http://robolab.unex.es/