Localización en Robótica Móvil

Localization In Mobile Robotics

Cezar Mihai Draghici
Departamento de Ingeniería Informática
Escuela Técnica Superior de Ingeniería Informática

Trabajo de Fin de Grado

La Laguna, 07 de Septiembre de 2014

D. **Jose Demetrio Piñeiro Vera**, con N.I.F. 43774048-B profesor Titular de Universidad adscrito al Departamento de Ingeniería Informática de la Universidad de La Laguna.

Dña. **Vanesa Muñoz Cruz**, con N.I.F. 78698687-R profesor ayudante Doctor adscrita al Departamento de Ingeniería Informática de la Universidad de La Laguna.

C E R T I F I C A N

Que la presente memoria titulada:

"Localización En Robótica Móvil."

ha sido realizada bajo su dirección por D. Cezar Mihai Draghici con NIE X5307677-J.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 7 de septiembre de 2014

Agradecimientos

El trabajo que aquí se presenta es el resultado de muchas horas de dedicación y duro esfuerzo, es el reflejo del lugar al que todo estudiante pretende llegar cuando inicia su andadura universitaria. Pero llegar a este punto no es tan importante como las experiencias vividas y, sobretodo, las personas que contribuyeron durante el camino. Me gustaría expresar mi profundo agradecimiento a todas ellas.

A mis tutores de proyecto, sin sus directrices y ayuda este trabajo no sería una realidad. Han sido sin duda una fuente de inspiración e ilusión, contagiándome de optimismo y ganas en todo momento.

A mis padres por su apoyo y confianza. Son los pequeños detalles los que marcan la diferencia y todos estos años han estado repletos de ellos.

A mi amigo Filipp por acompañarme en las noches de insomnio durante la carrera y animarme siempre que pudo.

En general a mis compañeros y amigos de facultad que me han acompañado durante estos años de facultad. Por sus palabras de ánimo, por las noches que hemos pasado sin dormir haciendo prácticas y compartiendo ideas, por todos los momentos y experiencias compartidas, a todos ellos, gracias.

Resumen

Este proyecto abarca varios temas relacionados con la robótica móvil. Comienza describiendo varios simuladores de robótica móvil, tratando las diferentes funcionalidades que ofrecen cada uno de ellos y analizándolos desde un punto de vista crítico. Prosigue con la elección de un simulador para el desarrollo de varias técnicas comúnmente utilizadas en robótica móvil. Finalmente se implementan estas técnicas en dicho simulador. Las técnicas consideradas se encuadran dentro de la planificación de rutas y la localización en presencia de incertidumbre.

Abstract

This project covers some topics related with mobile robotics. It starts describing some mobile robotic simulators, trying different functionalities and features that each of them offers and critically analyzing them. It continues with the selection of the best simulator for developing various common techniques used in mobile robotics. Finally, several of those mobile robotics techniques are implemented in the selected simulator. These techniques fall within the scope of path planning and localization in environments with uncertainty.

Índice General

Índice de figuras

Capítulo 1. Introducción

La robótica móvil surge de la necesidad de automatizar tareas tediosas como podrían ser la limpieza de una casa, la conducción, el transporte de material pesado, etc. Esta área de la robótica es muy reciente por lo que no se han realizado grandes avances, esto se debe en parte a la enorme complejidad que conlleva el desarrollo de algoritmos que resuelvan estas tareas tan difíciles con generalidades. Esta rama de la robótica se basa en otras áreas como el control, programación, inteligencia artificial, visión artificial, entre otras y sirve como base para el avance en diversos campos de la industria. Así este campo de la investigación está desarrollándose en todo momento, quedando aún mucho por recorrer.
En este proyecto se desean implementar varias técnicas relacionadas con la robótica móvil. Pero para lograr desarrollar estas técnicas, en primera instancia se debe estudiar en profundidad varios simuladores y decidir cuál de ellos ofrecerá mejores funcionalidades para lograr este fin. Una vez elegido el simulador se implementarán varias técnicas de robótica.
En la robótica móvil se podrían destacar cinco grandes áreas o temas. Estas son: medición, modelación, percepción, planificación y acción. A continuación se explicará con mayor detalle en qué consisten estas áreas.

1.1 Medición

La medición se centra principalmente en interpretar los datos percibidos a través de los sensores. Con estos datos el robot puede sensar su entorno e identificar objetos. Cuando nos enfrentamos al problema de sensar el entorno, se pueden utilizar los datos de uno o varios sensores. También se están desarrollando técnicas de sensado que combinan la información de varios sensores como un único sensor, como por ejemplo combinar una cámara con un sensor láser y obtener como información el patrón del objeto y la distancia de éste.

1.2 Modelado

En este apartado se puede diferenciar la modelización que un robot hace de su entorno, de los objetos de su entorno y de los caminos que detecta. El área de la modelización de su entorno se dedica estudiar la capacidad del robot para crear un mapa de su entorno el principal problema en este área es el comúnmente conocido como problema de SLAM. La modelización de objetos se encarga de reconocer patrones para objetos en concreto o crear un mapa con todo detalle de un objeto en concreto. La modelización de los caminos sirve principalmente para detectar las mejores rutas a seguir por el robot.

1.3 Percepción

La percepción se podría considerar como la capacidad del robot para localizarse en su entorno o la capacidad para detectar una posible situación de colisión con otros objetos. Esta capacidad para detectar las situaciones de colisión con otros objetos a veces entra dentro del área de la planificación, pero esto no es así siempre.

1.4 Planificación

La planificación se dedica principalmente a crear unos objetivos a largo plazo para el robot. También se encarga de descomponer la tarea u objetivo principal en un conjunto de subtareas a realizar a corto plazo. Por ejemplo, una tarea a largo plazo podría ser alcanzar un punto en el mapa, para realizar esta tarea es necesaria la creación de una ruta desde la posición actual hasta la posición final y finalmente dividir esto en subtareas que podrían ser los diferentes puntos que debe atravesar el robot para llegar a dicho punto final.

1.5 Acción

Esta área se encarga principalmente de la navegación del robot por el mundo. A la hora de navegar el robot debe evitar colisionar con los obstáculos que pueden surgir de manera no planificada en ciertos entornos. La navegación a su vez se divide en varios campos en función del tipo de robot, por ejemplo la navegación es distinta si el robot posee patas o ruedas.

Capítulo 2. Simuladores

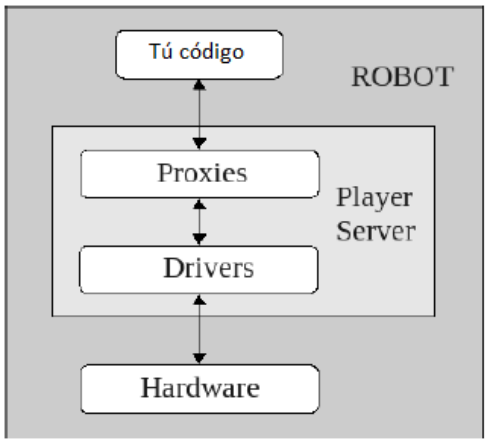
Un simulador de robótica se suele usar para crear aplicaciones empotradas para robots sin depender físicamente de la maquina en sí, ahorrando costes y tiempo. En algunos casos, estas aplicaciones se pueden implementar en robots reales sin modificaciones. El término simulador de robótica puede referirse a diferentes tipos de aplicaciones en robots simulados. Por ejemplo, en aplicaciones de robótica móvil, simuladores de robótica basada en comportamiento dan la posibilidad a los usuarios de crear mundos simples de objetos rígidos, fuentes de luz y programar robots que interactúen con estos mundos. Los simuladores basados en comportamiento habilitan acciones que son más de origen biológico que otros simuladores que son más de origen binario, o computacional.
En este proyecto se ha tomado la decisión de estudiar dos simuladores de robótica móvil. Uno de estos simuladores es Stage/Player. Se ha decidido estudiar este simulador debido a la facilidad que ofrece a la hora de simular mundos en dos dimensiones. Otra de las razones por las que se ha decidido investigar este simulador es por el hecho de que ofrece la suficiente complejidad como para poder desarrollar proyectos de investigación. Este simulador posee una gran comunidad de usuarios, los cuales aportan nuevos códigos constantemente para facilitar el uso de este simulador. Este simulador también es recomendado por un gran número de usuarios muchos de los cuales son profesionales en el sector de la robótica móvil.
Por otro lado se ha decidido estudiar el simulador STDR el cual al contrario que Stage/Player no posee un gran número de usuarios, ya que este simulador es muy reciente, de hecho aún no está desarrollado el cien por ciento de sus funcionalidades. Aún así este simulador implementa nuevas e innovadoras funcionalidades como definir modelos de los mundos virtuales a través de una interfaz gráfica, lo cual resulta muy atractivo y llamativo.
A continuación se explicarán de manera general las funcionalidades de cada uno de estos simuladores.

2.1 Player/Stage

2.1.1 Player

Player es un servidor en red multihilo para el control de robots. Ejecutándose en un robot, Player proporciona una interfaz simple y clara para comunicarse con los sensores y actuadores del robot a través de las redes IP. El programa cliente (en nuestro caso Stage) se comunica con Player a través de un socket TCP, leyendo datos de los sensores, escribiendo

órdenes en los actuadores y configurando dispositivos. La arquitectura que sigue Player es la siguiente:



Arquitectura Player

La plataforma original de Player es la familia ActivMedia Pioneer 2, pero muchos otros robots y sensores comunes son soportados actualmente. La arquitectura modular Player hace que sea muy fácil añadir soporte a nuevo hardware, además existe una comunidad de usuarios y desarrolladores que contribuyen con nuevos drivers.
Player se puede ejecutar en la mayoría de las plataformas compatibles con POSIX, incluyendo sistemas empotrados. Los requisitos que necesita Player son:

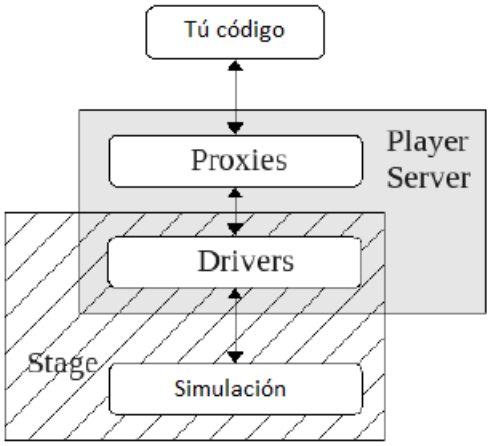
- Entorno de desarrollo POSIX.
- Pila de protocolos TCP/IP.
- Una versión reciente de GNU gcc, con soporte para C y C++.

Player hace una clara distinción entre la interfaz de programación y la estructura de control, optando por una interfaz de programación general, con la creencia de que los usuarios desarrollarán sus propias herramientas para construir sistemas de control. Es decir, está diseñado para ser independiente del lenguaje de programación y la plataforma.
Un programa cliente puede residir en cualquier máquina que tenga una conexión TCP con el robot en el que se ejecuta Player, y puede ser escrito en cualquier lenguaje que soporte sockets TCP. Actualmente existen clientes en C++, Tcl, Java, Python y más recientemente Ruby. Además, Player no impone restricciones en cuanto a cómo estructurar los programas de control del robot. Por último decir, que Player es de código abierto, bajo licencia pública GNU.

2.1.2 Stage

Stage es un simulador de robótica 2D relativamente sencillo que puede funcionar en cualquier ordenador moderno, pero que a la vez es potente y cubre las necesidades de la mayoría de los usuarios que no necesitan simular cámaras. Su principal limitación es que al

ser 2D no simula cámaras, sin embargo, gracias a esta limitación permite también que este simulador pueda funcionar en casi cualquier máquina. Es recomendable, siempre que no sea totalmente obligatorio el uso de cámaras, comenzar con Stage y hacer que el robot base su comportamiento en las lecturas del láser, sonares, blobfinders, etc. Gracias a sus bajos requisitos de hardware, muchas veces se usa Stage para simular grupos relativamente grandes de robots simultáneamente. Generalmente los robots en Stage se programan mediante la interfaz de Player. La arquitectura que forma Stage junto con Player es la siguiente:



Arquitectura Stage/Player

Stage proporciona un mundo virtual poblado de robots móviles y sensores, los cuales se ayudan de varios objetos auxiliares que le sirven para sensar y manipular el mundo.
Existen tres modos para usar Stage:

1. El programa Stage: un independiente programa de simulación de robots que carga el programa de control del robot desde una librería que se le proporciona.
2. El plugin de Stage para Player (libstageplugin): proporciona una población de robots virtuales para el popular Player y su red de sistemas de interfaces para robots.
3. Escribir tu propio simulador: la librería C++ "libstage" logra que sea fácil de crear, ejecutar y personalizar un simulador Stage desde dentro de tus propios programas.

Stage proporciona varios modelos de sensores y actuadores tales como: sonar o rangos de infrarrojo, escáneres de laser, rastreadores de manchas de colores, rastreadores fiduciales, bumpers, pinzas y robots móviles basados en odometría o localización global.
Stage fue desarrollado con un sistema multi-agente en mente, así que es bastante simple, e intenta simular cualquier dispositivo con una gran fidelidad. Este diseño está destinado a tener un compromiso útil entre los convencionales simuladores de robots de alta fidelidad y los mundos simulados comunes en investigaciones de vida artificial. Stage está diseñado para ser tan realista que brinda la posibilidad a los usuarios de usar los controladores

diseñados en robots reales. También está diseñado para ser comprendido por estudiantes y tener la suficiente complejidad como para que se pueda usar en el mundo profesional por investigadores.
Por otro lado Player proporciona varios dispositivos virtuales, incluyendo algunos sensores pre-procesados o algoritmos de sensores integrados que ayudan a construir rápidamente robustos controladores de robots. Estos son fácilmente integrables con Stage.
El nombre de Player/Stage proviene de una famosa frase de Shakespeare:

"All the world's stage,
And all the men and women merely players."

2.1.3 Descripción General del Funcionamiento

Archivos Importantes

En Player/Stage hay tres tipos de ficheros que se necesita comprender para trabajar con este simulador:

1. Los archivos .world
2. Los archivos .cfg (configuration)
3. Los archivos .inc (include)

Los archivos ".world" explican a Player/Stage qué modelos hay disponibles para colocar en el mundo. En estos archivos se describe el robot, cualquier objeto que vaya a poblar el mundo y se describe la disposición del mundo. Los archivos ".inc" siguen el mismo formato y distribución que los archivos ".world", pero estos deben ser incluidos en los ".world". Así que si existe un objeto en el mundo que se debería incluir en otros mundos, así como modelos de robots, colocando la descripción de estos en un fichero ".inc" facilitará la tarea de copiarlos. También significa que si algún día deseas cambiar la descripción de el robot, solamente se necesitará hacerlo en un único lugar y se modificará en todas las demás simulaciones. El fichero ".cfg" es lo que Player necesita para recibir toda la información sobre el robot que se vaya a usar. Este fichero le describe a Player que drivers necesita usar para interactuar con el robot. Si se está usando un robot real estos drivers se construyen en Player, alternativamente y si se desea realizar una simulación, el driver será siempre Stage (así es como Player utiliza Stage de la misma manera que usa un robot, Player piensa que es un driver de un hardware pero en realidad se está comunicando con Stage). El archivo ".cfg" le dice a Player cómo hablar con el driver, y cómo interpretar cualquier dato del driver y de esta manera se presentará en el código. Los modelos descritos en los ficheros ".world" deberían ser invocados en el fichero ".cfg" si se desea que el código sea capaz de interactuar con estos objetos (como podría ser un robot), si no se necesita que el código interactúe con los objetos entonces esto no es necesario. El

fichero ".cfg" realiza todas estas especificaciones usando interfaces y drivers, los cuales se explicarán a continuación.

Interfaces, Drivers y Dispositivos

- Los drivers son piezas de código que se comunican directamente con el hardware. Estos se construyen en Player así que es importante saber cómo se escriben estos para comprender mejor el funcionamiento de Player/Stage. Los drivers son específicos de un trozo de hardware, por tanto, el driver de un laser será diferente del driver de un sónar y también será diferente para cada modelo de un láser. Esto es igual que en el caso de las tarjetas gráficas dónde necesitas un driver diferente para cada modelo de tarjeta. Los drivers producen y leen información proveniente de las interfaces.
- Las interfaces son un conjunto de caminos hacia los drivers para enviar y recibir información desde Player. Como los drivers, las interfaces se construyen dentro de Player y se puede encontrar una gran lista de éstos en los manuales de Player. Éstos especifican la sintaxis y semántica de cómo los drivers interactúan con Player.
- Un dispositivo es un driver el cual está ligado a una interfaz a la cual Player se puede dirigir directamente. Esto se refiere a que si se trabaja en un robot real el cual puede interactuar con un dispositivo real (láser, gripper, sónar, etc), un robot simulado puede interactuar con sus simulaciones.

2.2 STDR

STDR (Simple Two Dimensional Robot Simulator) es un simulador robótico en 2D bastante sencillo. El objetivo de este simulador no es ser el más realista o el simulador con más funcionalidades. El verdadero objetivo de este simulador es poder simular un robot o un enjambre de robots de la manera más sencilla posible. Para ello se minimiza el número de acciones que el usuario necesita realizar para iniciar su experimento.
Este simulador cuenta con una potente herramienta gráfica, la cual te permite crear tus propios mapas o crear tus propios robots, a un nivel tan minucioso que te permite incluir sensores y la posibilidad de configurarlos. Por otro lado, mediante el uso de una consola también permite el uso de comandos para ejecutarlo sin necesidad del entorno gráfico.

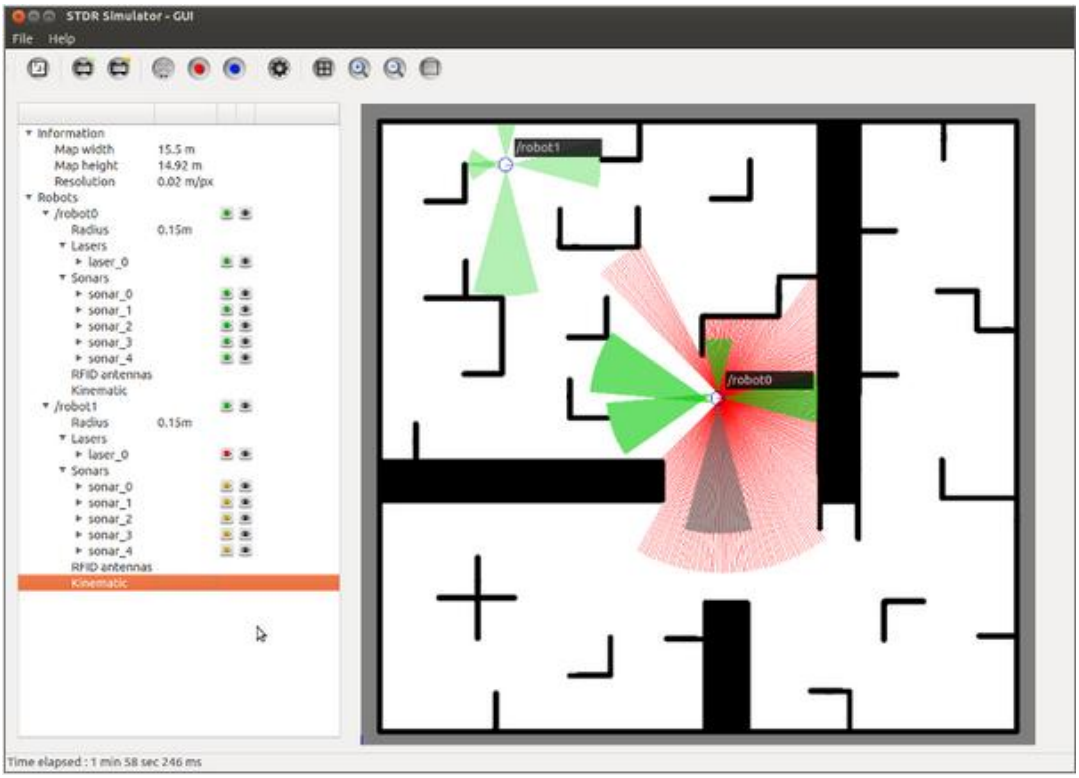


Imagen de la interfaz gráfica del simulador STDR.

Este simulador está creado de tal manera que es completamente dependiente de ROS.
ROS (Sistema Operativo Robótico) es un framework para el desarrollo de software para robots que posee todas las funcionalidades de un sistema operativo. ROS proporciona a los usuarios librerías y herramientas para facilitar el desarrollo de aplicaciones robóticas.
Todos los robots y sensores de este simulador emiten transformaciones ROS y todas las medidas son publicadas en los temas de ROS. En este sentido STDR aprovecha todas las ventajas de ROS, apuntando a un manejo sencillo del mundo y un muy buen estado artístico en el marco de los robots.
Por último decir que este simulador es una buena herramienta didáctica, útil para introducir estudiantes en el mundo de la robótica debido a su gran sencillez y a su potente herramienta gráfica. Por otro lado también puede resultar útil para realizar un prototipo de un proyecto de investigación debido a su facilidad y agilidad a la hora de recrear mundos robóticos simulados.

2.2.1 Descripción General del Funcionamiento

La mayor parte de las funcionalidades de este simulador se pueden realizar a través del entorno gráfico del mismo. Estas funcionalidades son: Crear robot, cargar o crear sensores para el robot, cargar mapas, cargar robots.

Para este simulador existen dos tipos de ficheros:

- Ficheros tipo ".yaml", en este formato se puede describir cualquier modelo para este simulador como: robots, sensores o mapas.
- Ficheros tipo ".xml", en este formato de fichero solo se puede describir los robots y los sensores, los mapas no se pueden cargar en este formato o al menos no se ha contemplado esta posibilidad.

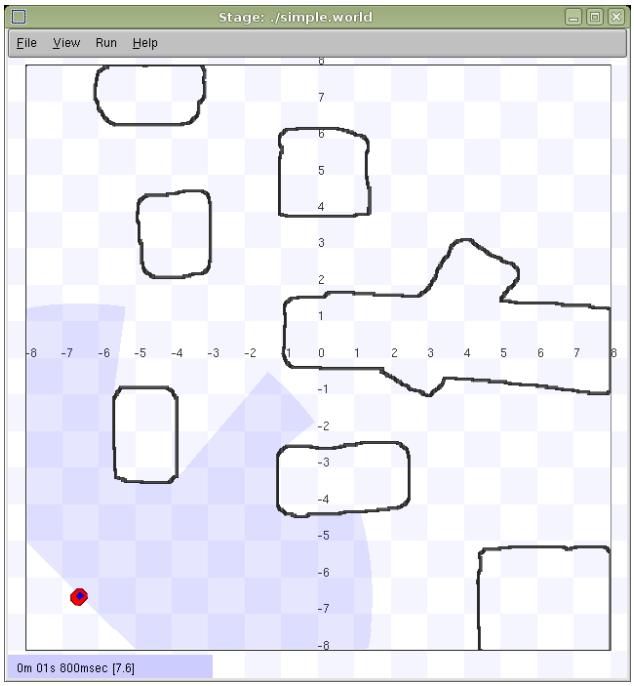
Capítulo 3. Ejemplos y Análisis de los Simuladores

En este capítulo se describirá detalladamente cómo crear un proyecto en los diferentes simuladores, desde la creación y simulación del mundo hasta el desarrollo de algún código para poder empezar a trabajar. Por último se realizará un análisis crítico de ambos simuladores y se seleccionará uno para desarrollar futuras técnicas.

3.1 Stage/Player

3.1.1 Ejecutar un Ejemplo Base

Antes de comenzar a explicar un ejemplo en profundidad se desea hablar sobre la ejecución de un ejemplo que viene acompañando a Stage se trata de el ejemplo "simple". Este ejemplo muestra un Pioneer en un mapa manejado por un algoritmo llamado "Wander". Este ejemplo se encuentra dentro Stage en la carpeta worlds. Para ejecutar este ejemplo debemos colocarnos con nuestra consola en dicha carpeta y ejecutar el comando:
player simple.cfg
Tras ejecutar el comando debería aparecer la siguiente GUI:



Ejemplo base Stage/Player

3.1.2 Elementos Básicos

Como se puede observar en el comando mencionado, cuando le decimos a Player que construya un mundo solo debemos proporcionarle el fichero ".cfg" como entrada. Este fichero de tipo ".cfg" debe indicarnos dónde encontrar los diferentes ficheros, los cual le facilitarán a Stage toda la información que necesita para simular el mundo virtual.

A continuación vamos a desarrollar un ejemplo muy completo en Stage/Player para entender el funcionamiento de este.
Vamos a definir el fichero "mundo.cfg" que tendrá las siguientes líneas de código:

```
driver
(
  name "stage"
  plugin "stageplugin"
  provides ["simulation:0" ]
  # carga el fichero nombrado en el simulador
  worldfile "mundo.world"
)
```

Aquí básicamente lo que ocurre es que el fichero de configuración le dice a Player que existe un driver que se llama "stage" en la librería "stageplugin" y esto le proporcionará a Player los datos que conforman la interfaz de simulación. Para construir la simulación Stage necesita comprobar el fichero llamado "mundo.world", el cual está guardado en la misma carpeta como indica la ruta relativa especificada.
A continuación pasaremos a explicar el fichero "mundo.world". En los ficheros ".world" se declara todos los modelos que van a existir en el mundo. Las definiciones de cada modelo deberían hacerse en los ficheros ".inc" aunque se podrían realizar en el fichero ".world" sin ningún error, sin embargo se realiza en los ficheros ".inc" por una cuestión de modularidad y buena praxis. Supongamos que nosotros creamos en este ejemplo el robot "apolo", y supongamos que en un futuro queremos volver a utilizar en otro proyecto el robot "apolo" si nosotros definimos el robot en el mismo fichero ".world", lo cual se puede hacer, para incluirlo en el nuevo proyecto deberíamos copiar las líneas de código en este. Sin embargo si lo definimos en un fichero ".inc" lo único que tendríamos que hacer es incluir en nuestro nuevo proyecto dicho fichero, y ya lo tendríamos a disposición para volver a utilizar.

Veamos a continuación como definiríamos un modelo:

```
define floorplan model
(
  color "gray30"
  boundary 1
  gui_nose 0
```

```
  gui_grid 0
  gui_move 0
  gui_outline 0
  gripper_return 0
  fiducial_return 0
  laser_return 1
)
```

En la primera línea podemos ver que se define un modelo llamado "floorplan" lo que equivaldría en español a "plano del suelo" es el nombre que se le suele poner al modelo del mapa, puede ser otro, pero este nombre es casi un convenio.

- color: Le dice a Stage/Player el color de renderizado.
- boundary: Esta línea crea una caja que rodea a este modelo (1 verdadero y 0 false), es útil en este caso ya que al crear una caja alrededor del mapa se evita que el robot pueda salir libremente del mapa, se está por tanto creando un muro que encierra el mapa que se ha cargado.
- gui_nose: Esto dice a Player/Stage hacia donde está mirando el modelo.
- gui_grid: Esto superpondrá una malla al mundo.
- gui_move: Esta opción permitirá coger y arrastrar el modelo en caso de que este a 1.
- gui_outline: esto indica si el modelo es o no esbozado. Esto podría no tener ninguna utilidad en el mapa, sin embargo en otros objetos que se planteen en el mundo pude tener interesantes utilidades.
- fiducial_return: Esto describe cómo se comportará el modelo ante los sensores fiduciales. En caso de estar a 0 el mapa no devolverá ningún valor a los sensores fiduciales.
- gripper_return: Se comporta como en el caso de los sensores fiduciales, pero en este caso es para los sensores gripper.

Estas líneas de código que se han descrito debería ir incluidas en un fichero llamado "map.inc" ya que es la descripción de una definición de un modelo, más concretamente del modelo del mapa.

El fichero "mundo.world" tendría un código como el que sigue:

```
include "map.inc"
floorplan
(
  bitmap "bitmaps/holamundo.png"
  size [12 5 1]
)
```

La primera línea sirve para instanciar el modelo que hemos definido en "map.inc" a nuestro mundo.

- bitmap: Esto sirve para cargar un bitmap, el cual pude ser de tipo .bmp, .jpg, gif o png. Las zonas negras del mapa indican al modelo las paredes, las blancas los espacios disponibles y el resto de los colores no serán renderizados.
- size: este es el tamaño en metros de la simulación. En este ejemplo es un mapa de 12m ancho x 5m largo y 1m de alto.

3.1.3 Desarrollo del Código de Simulación

Una vez visto los elementos básicos que rodean estos ficheros comencemos a definir nuestro mundo más en detalle. Primero como siempre el fichero "mundo.cfg":

```
driver (
 name "stage"
 plugin "stageplugin"
 provides ["simulation:0" ]
 worldfile "mundo.world"
)
driver (
 name "stage"
 provides ["6665:position2d:0"
        "6665:sonar:0"
        "6665:blobfinder:0"
        "6665:laser:0"]
 model "apolo1"
)
```

Lo primero es el driver que tiene que acompañar a todos nuestros proyectos Stage/Player ya que es el medio por el que se comunican ambos elementos, anteriormente se ha explicado cada uno de los elementos de este, por lo que pasemos a la parte de código nueva. En segundo lugar podemos observar la definición de nuestro robot para Player. Veamos línea a línea que es lo que ocurre:

- name "stage": esta línea le indica a Player la librería que debe usar para este modelo.
- provides [...]: esta línea le indica a Player las diferentes interfaces de Player que debe usar especificando el puerto en el que se podrán escuchar cada una y el nombre de éstas.
- model "apolo1": es el nombre que se le dará a la instancia del robot ejemplo, para este caso se ha nombrado al robot "apolo".

Una vez explicado el fichero "mundo.cfg" prosigamos con la explicación del fichero "mundo.world" el cual al ser largo se va a explicar por partes.
En primer lugar irá la cabecera que será los ficheros que vamos a incluir:

```
include "map.inc"
```

include "apolo.inc"
Aquí se incluye el fichero en el que está definido el mapa y el fichero en el que se define el robot.

A continuación se pueden definir algunos parámetros que facilita Stage, aunque no se van a exponer todos ya que están definidos en el manual de Stage, solo se van a definir unos parámetros como ejemplo. Aunque todos los parámetros ya tienen sus valores predefinidos, estos valores se pueden modificar si así se desea. Por ejemplo:
interval_sim 0.01
interval_real 0.01

- interval_sim: este parámetro permite modificar los intervalos de actualización de la ventana de simulación (en segundos). El valor por defecto es 100 milisegundos.
- interval_real: este parámetro permite modificar el tiempo real que transcurre entre cada actualización del mundo simulado (en segundos). El valor por defecto es de 100 milisegundos.
- De esto se puede deducir que con la modificación de ambos parámetros se puede modificar la velocidad de simulación.

Como se puede observar a ambos parámetros se les asigna el valor que tienen por defecto, solo sirven como ejemplo, es decir que estas líneas se pueden omitir y el resultado sería el mismo.

Una vez especificados los valores de los diferentes parámetros que deseamos modificar procedemos a la declaración de las diferentes instancias de nuestro mundo.

```
window (
  size [700 700]
  scale 35
)
```

window: es la instancia de la ventana de simulación.
size [700 700]: esta línea sirve para definir el tamaño de la ventana de simulación.
scale 35: esta línea sirve para definir cuantos pixeles será 1 metro en el mundo simulado.

```
floorplan (
  bitmap "bitmaps/custom_map.png"
  size [15 15 0.5]
)
```

Aquí se instancia el suelo y ya se ha explicado previamente.

```
apolo (
  name "apolo1"
  pose [-5 -6 0 45]
```

```
  color "green"
)
```

Aquí se instancia el robot y a continuación se explicara cada línea:

- name "apolo1": es el nombre de la instancia del robot.
- pose [-5 -6 0 45]: es la pose del robot, el primer parámetro es la coordenada x en el mundo simulado, el segundo parámetro es la coordenada y, el tercer parámetro es la coordenada z y el cuarto parámetro es la orientación en grados.
- color "green": es el color de renderizado del modelo.

A continuación mostramos el contenido del fichero "map.inc" que ya ha sido explicado anteriormente.

```
define floorplan model (
  color "gray30"
  boundary 1
  gui_nose 0
  gui_grid 0
  gui_move 0
  gui_outline 0
  gripper_return 1
  fiducial_return 1
  laser_return 1
)
```

A continuación se procederá a explicar cómo definir un modelo de un robot, en este caso del "apolo" y al ser largo se procederá por pasos. Este robot está definido en el fichero "apolo.inc" de la siguiente manera:

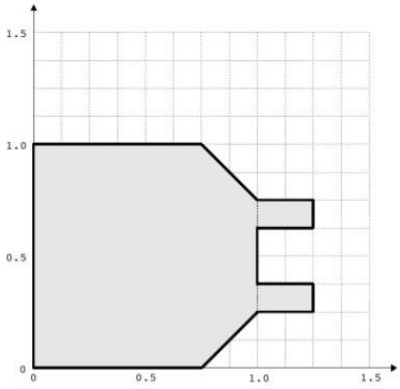
Primero la definición del modelo

```
define apolo position (
  size [0.625 0.5 0.5]
  .
  .
  .
)
```

Aquí se puede observar que se define el modelo "apolo" el cual hereda del modelo position. Este modelo simula la odometría del robot. Pero esto no es lo más importante de este modelo, lo más importante es que le proporciona un cuerpo al modelo lo cual le permite colisionar con los distintos objetos del mundo. Además este modelo utiliza la interfaz "position2d" de Player, que le comunica a éste dónde se encuentra actualmente el robot en el mundo. También hay una línea de código la cual define la escala del robot, esto

es, el ancho del robot multiplicado por el primer parámetro, el largo del robot por el segundo y el alto del robot por el tercero.

A continuación se definirá el aspecto del robot el cual quedará de la siguiente manera:



Coordenadas de creación del robot Stage/Player

Esto se definirá con el siguiente código:

```
 block (
   points 6
   point[5] [0 0]
   point[4] [0 1]
   point[3] [0.75 1]
   point[2] [1 0.75]
   point[1] [1 0.25]
   point[0] [0.75 0]
   z [0 1] )

 block (
   points 4
   point[3] [1 0.75]
   point[2] [1.25 0.75]
   point[1] [1.25 0.625]
   point[0] [1 0.625]
   z [0 0.5] )
 block (
   points 4
```

```
point[3] [1 0.375]
point[2] [1.25 0.375]
point[1] [1.25 0.25]
point[0] [1 0.25]
z [0 0.5] )
```

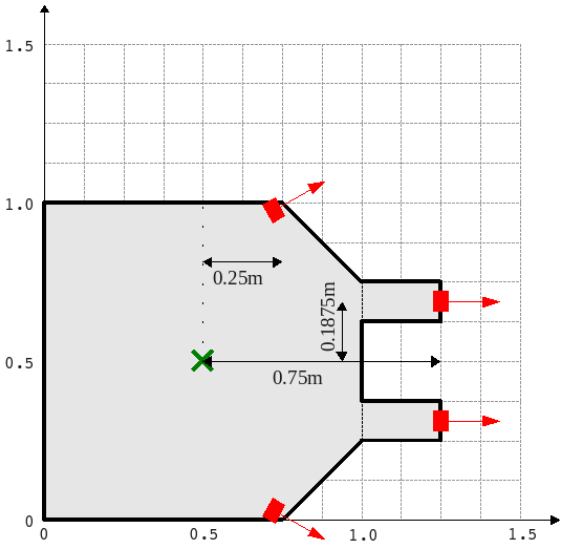
El primer bloque sirve para definir el cuerpo y los siguientes dos para definir los brazos.
Por tanto block sirve para definir un bloque de puntos los cuales representarán el aspecto físico de este. La siguiente línea precedida de "points" sirve para definir la cantidad de puntos que habrá. Posteriormente se define la posición de cada punto, y por último la altura para la cual se especifica altura inicio, altura final. Para un mejor funcionamiento los puntos se deben especificar en orden y en sentido contrario a las agujas del reloj.

En último lugar definiremos los parámetros del robot así como los diferentes sensores que posee:

```
drive "diff"
apolo_sonars()
apolo_eyes()
apolo_laser()
obstacle_return 1
laser_return 1
ranger_return 1
blob_return 1
```

Aquí cabe destacar la línea drive "diff" la cual le dice a Stage/Player el tipo de robot móvil que será el nombre "diff" viene dado por "differential steering model" y quiere decir que el robot será capaz de girar sin la necesidad de desplazarse del lugar. Las siguientes tres líneas sirven para instanciar los sensores del robot, que serán sónar, blobfinder y láser. El resto de los parámetros ya han sido explicados anteriormente.
Con esto terminaríamos de definir nuestro robot, sin embargo nos faltan por definir los sensores, los cuales se describirán a continuación. Por simplicidad se han incluido todos los sensores en el mismo fichero en el que se ha definido el robot, sin embargo lo correcto sería definir cada uno de ellos en su propio fichero por la modularidad, razón ya explicada al principio del capítulo.

Comencemos con la definición del sónar:
Para poder visualizar mejor la definición de los sonares se ha preparado una imagen en la que se puede observar claramente:



Coordenadas sónares Stage/Player

A continuación el código asociado:

```
define apolo_sonars ranger (
  scount 4
  spose[0] [ 0.75 0.1875 0 ]
  spose[1] [ 0.75 -0.1875 0 ]
  spose[2] [ 0.25 0.5 30]
  spose[3] [ 0.25 -0.5 -30]
  sview [0.3 2.0 10]
  ssize [0.01 0.05]
)
```

- define apolo_sonars ranger: esta línea sirve para definir el sónar y cómo podemos observar hereda del modelo base ranger el cual enlaza con la interfaz ranger de Player.
- scount 4: está línea define la cantidad de sónares a 4.
- spose[count][x, y yaw]: en las líneas posteriores se define las poses de cada sónar de la siguiente manera [x, y, Orientación] y estos serán relativos al centro del robot.
- sview[0.3 2.0 10]: esta línea define el campo de visión de todos los sónares de la siguiente manera [rango mínimo, rango máximo, ángulo en grado].
- ssize [0.01 0.05]: como su nombre indica define el tamaño de todos los sonares.

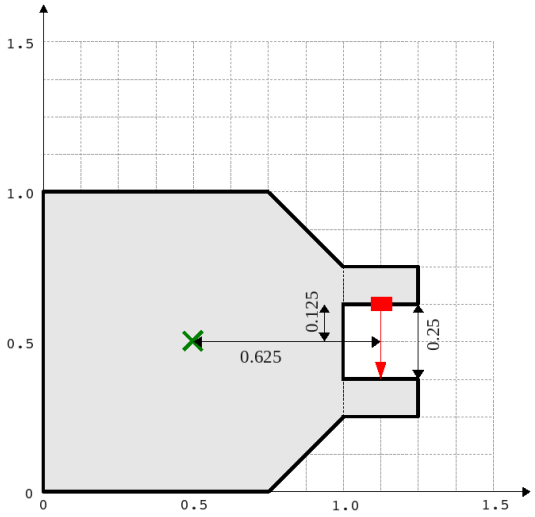
Comencemos con la descripción del sensor blobfinder:

```
define apolo_eyes blobfinder (
  colors_count 2
  colors ["orange" "DarkBlue"]
```

```
  image [160 120]
  range 5.00
  fov 1.047196667 # 60 degrees = pi/3 radians
)
```

- define apolo_eyes blobfinder: en esta línea se define el modelo que hereda de blobfinder y enlaza con la correspondiente interfaz de Player.
- colors_count 2: en esta línea se define la cantidad de colores que percibirá el sensor.
- colors ["orange" "DarkBlue"]: en esta línea se define los colores que podrá captar el sensor.
- image[160 120]: aquí se define la resolución de la cámara del sensor.
- range 5.00: será el rango máximo de visión del sensor.
- fov 1.047196667: es el ángulo de visión del sensor en radianes pi/3, o en grados 60.

Por último pasamos a la definición de los láseres, la cual se puede observar mejor con la siguiente imagen:



Coordenadas láseres Stage/Player

A continuación pasemos al código:

```
define apolo_laser laser (
  range_max 0.25
  fov 20
  samples 180
  pose [0.625 0.125 -0.975 270]
  size [0.025 0.025 0.025]
)
```

- define apolo_laser laser: en esta línea se define el láser que hereda del modelo laser.
- range_max 0.25: es el rango máximo de visión.
- fov 20: es la cantidad de rayos que posee el láser.
- sample 180: es el ángulo que cubren los rayos del láser.
- pose [0.625 0.125 -0.975 270]: es la pose del láser definida de la siguiente manera [x, y, altura, orientación en grados]. La altura es negativa debido a que se toma como referencia el punto más alto del cuerpo del robot, y como el robot tiene 1 metro de altura se puede colocar en -0.975.
- size [0.025 0.025 0.025]: es el tamaño del láser en x, y, z.

Todo este código tendría el siguiente aspecto en la simulación:

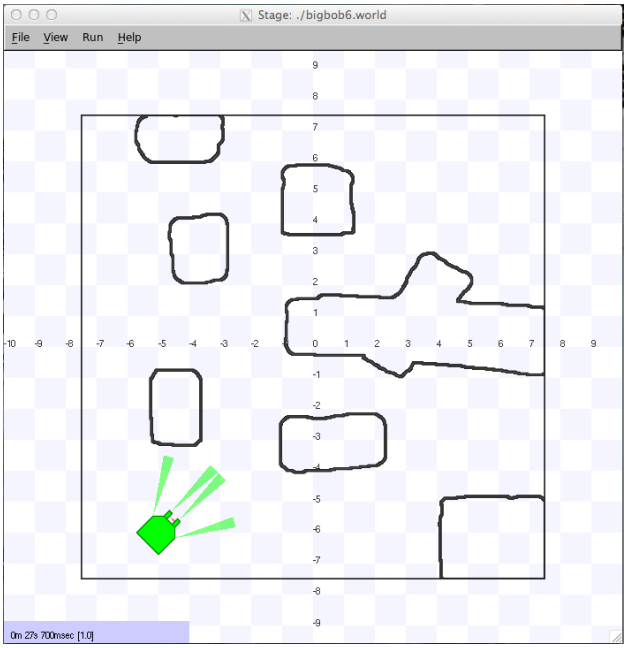


Imagen del simulador con el ejemplo desarrollado

Con esto quedan concluidos los diferentes aspectos de la simulación.

3.1.4 Desarrollo de Código de Ejecución

A continuación se describirá un pequeño código en C++ que se conecta al servidor de Player, lee los datos de los sensores y mueve el robot.

```
#include <libplayerc++/playerc++.h>
using namespace PlayerCc;
```

```
int main(int argc, char *argv[]) {
        PlayerClient cliente("localhost", 6665);

        Position2dProxy p2dp (&cliente, 0);
        LaserProxy lp(&cliente, 0);
        SonarProxy sp(&client, 0);
        BlobfinderProxy blp(&client, 0);

        int blobCount = bfp.GetCount();
        int sonarCount = sp.GetCount();
        int laserCount = lp.GetCount();

        playerc_blobfinder_blob_t blobDat;
        double dato;

        p2dp.SetMotorEnable(1);
       while (true) {
                cliente.Read();
                for(int i = 0; i < blobCount; i++) {
                        blobDat = bfp[i];
                }
                for(int i = 0; i < sonarCount; i++) {
                        dato = sp[i];
                }
                for(int i = 0; i < laserCount; i++) {
                        dato = lp[i];
                }
                p2dp.SetSpeed(1, 0.1);
        }
        return 0;
}
```

Se procederá a describir línea a línea el código mostrado anteriormente.

#include <libplayerc++/playerc++.h>

Es la librería necesaria para conectarse a Player.

using namespace PlayerCc;

Se utiliza el espacio de nombres de PlayerCc por simplicidad.

PlayerClient cliente("localhost", 6665);

Esta línea sirve para conectarse con el servidor de Player y crear el cliente.

Position2dProxy p2dp (&cliente, 0);

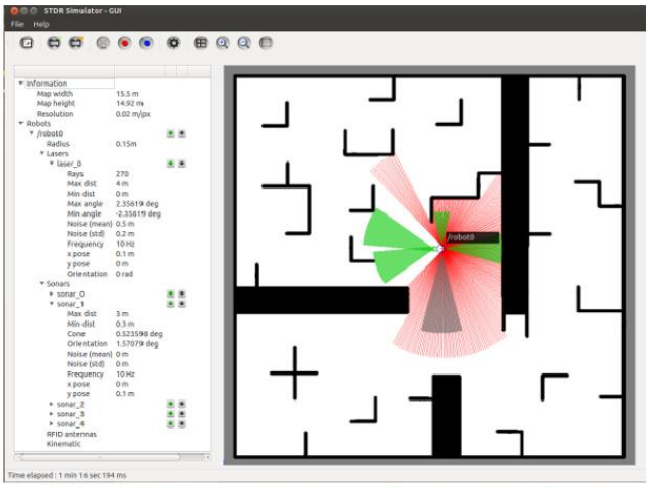
LaserProxy lp(&cliente, 0);
SonarProxy sp(&client, 0);
BlobfinderProxy blp(&client, 0);
Estas líneas sirven para obtener cada uno de los proxys para las diferentes interfaces de Player que se emplea.
int blobCount = bfp.GetCount();
int sonarCount = sp.GetCount();
int laserCount = lp.GetCount();
En estas líneas se obtiene la cantidad de sensores de cada uno de los tipos que se tienen declarados.
playerc_blobfinder_blob_t blobDat;
Se declara el tipo de dato en el que se almacena una lectura del blobfinder.
p2dp.SetMotorEnable(1);
Iniciamos los motores del robot.
while (true) {
Bucle infinito donde se seguirá al robot.
cliente.Read();
Refrezca los datos de los sensores.
for(int i = 0; i < blobCount; i++) {
	blobDat = bfp[i];
}
Bucle para obtener los datos del blobfinger.
for(int i = 0; i < sonarCount; i++) {
	dato = sp[i];
}
Bucle para obtener los datos del sonar.
for(int i = 0; i < laserCount; i++) {
	dato = lp[i];
}
Bucle para obtener los datos del laser.
p2dp.SetSpeed(1, 0.1);
Esta línea modifica la velocidad del robot a 1 m/s y la velocidad de giro a +0.1 rad/s

3.2 STDR

El simulador STDR permite crear modelos de manera muy sencilla a través de su potente Interfaz Gráfica, a un nivel muy detallado. Por otro lado este simulador también permite definir modelos en sus propios ficheros con una sintaxis determinada. Ambos casos se explicarán a continuación.

3.2.1 Interfaz Gráfica

Como ya se ha mencionado la mayoría de las funcionalidades de este simulador están disponibles su GUI. No hay mejor manera para entender una interfaz que verla.



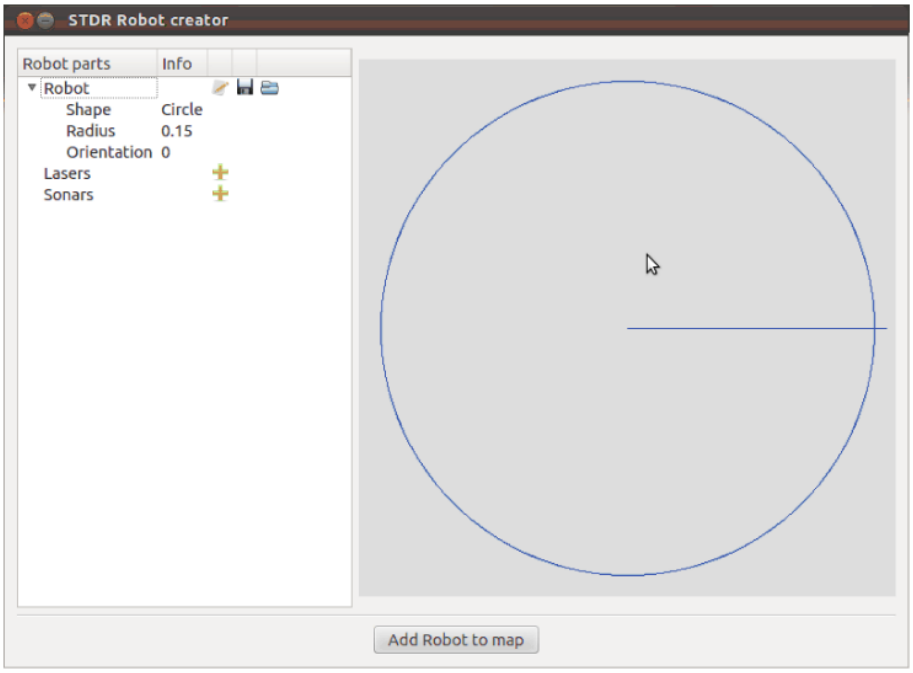
La barra de heramientas se encuentra en la parte superior de la GUI

Posee 11 botones los cuales son:

-  Cargar mapa
-  Añadir robot al mapa
-  Crear robot
-  Añadir etiqueta RFID - Aún no implementada
-  Añadir fuente termal - Aún no implementada
-  Añadir fuente de CO2 - Aún no implementada
-  Propiedades - Aún no implementada
-  Habilitar / Deshabilitar malla
-  Acercarse / alejarse
-  Ajustar mapa a la ventana

Interfaz STDR

Para cargar un mapa hay que pulsar el botón para cargar un mapa y seleccionar un archivo ".yaml". Para cargar un robot, pulsar el botón cargar un robot y seleccionar un archivo ".xml" que tenga el formato adecuado. Para crear un robot, hay que pulsar el botón para crear un robot y aparecerá la siguiente interfaz:



Interfaz de creación de robot en STDR

Podemos observar que existen opciones para modificar el nombre del robot, su forma, su radio (tamaño), y su orientación. Por otro lado se tiene la posibilidad de añadir sensores tales como láseres o sónares. Una vez que se añada un láser o un sónar, se permite configurar todas las características de dicho sensor y también se ofrece la opción para cargar una configuración de sensor.

Una vez creado o cargado el robot, se debe hacer clic en la zona del mapa donde se desea que aparezca el robot. Tras añadir el robot al mapa si se pulsa clic derecho sobre él aparecerán las siguientes opciones:

- Borrar robot: esta opción elimina el robot del mapa.
- Mover robot: esta opción permite volver a situar el robot en una nueva posición.
- Ver proximidades del robot: esta opción permite ver los puntos con los que colisiona cada uno de los sensores.
- Seguir robot: esta opción bloquea la cámara sobre el robot en concreto.

3.2.2 Sintaxis de los Ficheros

En los ficheros se puede definir modelos de: un robot, un sensor o un mapa. Los modelos de los robots y los sensores se pueden definir en ficheros con la extensión ".xml" o

".yaml", sin embargo el mapa solamente se puede definir en ficheros con la extensión ".yaml". A continuación se explicará cada uno de los formatos para los diferentes modelos.

Formato del Fichero de un Mapa

Como ya se ha comentado este fichero solo se puede definir con la extensión ".yaml" por lo que veamos un código típico de dicho formato, para ello se mostrará el contenido del fichero "mapa.yaml":

```
image: mapa.png
resolution: 0.05
origin: [0.0, 0.0, 0.0]
occupied_thresh: 0.6
free_thresh: 0.3
negate: 0
```

La definición de este fichero es muy sencilla, especificamos una imagen que servirá como mapa, la resolución y las coordenadas origen donde se situará el mapa.

Formato del Fichero de un Sensor

Los ficheros de un sensor pueden definirse tanto en formato xml como en formato yaml, por lo que veamos a continuación el contenido de un fichero "sensor.xml":

```
<laser>
 <laser_specifications>
  <max_angle>1.570795</max_angle>
  <min_angle>-1.570795</min_angle>
  <max_range>4.0</max_range>
  <min_range>0.1</min_range>
  <num_rays>20</num_rays>
  <frequency>10</frequency>
  <pose>
   <x>0</x>
   <y>0</y>
   <theta>0</theta>
  </pose>
   <noise>
    <noise_specifications>
     <noise_mean>0</noise_mean>
     <noise_std>0.08</noise_std>
    </noise_specifications>
   </noise>
 </laser_specifications>
</laser>
```

Como se puede observar en la definición de este fichero nos permite definir cada uno de los valores para los distintos parámetros que posee un láser, como el ángulo, el rango, el numero de rayos, la pose, el ruido, entre otros.
Ahora veamos cómo sería el contenido de un fichero "laser.yaml":

```
laser:
 laser_specifications:
  max_angle: 1.570795
  min_angle: -1.570795
  max_range: 4
  min_range: 0.1
  num_rays: 20
  noise:
   noise_specifications:
    noise_mean: 0
    noise_std: 0.08
  frequency: 10
  pose:
   x: 0
   y: 0
   theta: 0
```

Este fichero es equivalente al fichero anterior "laser.xml" lo único que difiere es el formato.

Formato del Fichero de un Robot

Los ficheros de un robot pueden definirse tanto en formato xml como en formato yaml, por tanto, veamos cómo se definiría el contenido de un fichero "robot.xml":

```
<robot>
 <robot_specifications>
  <footprint>
   <radius> 0.5 </radius>
  </footprint>
  <initial_pose>
   <x> 0 </x>
   <y> 0 </y>
   <theta> 0 </theta>
  </initial_pose>
  <laser>
   <filename> laser.xml </filename>
   <laser_specifications>
    <pose>
     <theta> 1.57 </theta>
```

```
      </pose>
    </laser_specifications>
  </laser>
  <laser>
    <filename> laser.xml </filename>
    <laser_specifications>
      <pose>
        <theta> -1.57 </theta>
      </pose>
    </laser_specifications>
  </laser>
 </robot_specifications>
<robot>
```

En este fichero se observa lo mismo que en el caso del láser, se define el modelo que es uno de tipo robot, luego se define sus parámetros como el tamaño la pose, etc. Sin embargo, en este modelo se observa algo nuevo y son las instancias de dos láseres en el robot. Para instanciar los láseres en el robot se debe especificar la ruta del fichero donde está definido el láser y los parámetros que se quieren modificar.

Por último se verá el formato del fichero "robot.yaml":

```
robot:
 robot_specifications:
   - footprint:
       footprint_specifications:
         radius: 0.15
   - initial_pose:
       theta: 0
       x: 0
       y: 0
   - laser:
       filename: laser.yaml
       laser_specifications:
         theta: 1.57
   - laser:
       filename: laser.yaml
       laser_specifications:
         theta: -1.57
```

Este fichero es equivalente al fichero anterior "robot.xml" por lo que dará el mismo resultado.

Estos tipos de ficheros se pueden definir de manera manual escribiendo el código directamente. Por otro lado, también se pueden definir a través de la interfaz gráfica que ofrece este simulador y guardarlos para usos posteriores.

3.3 Análisis de los Simuladores

En este apartado se analizarán las ventajas y desventajas de cada simulador, las posibilidades que ofrecen cada uno de ellos y se concluirá con la elección de uno para su posterior uso.

3.3.1 Análisis de Funcionalidades

Se comenzará analizando cada una de las funcionalidades de los simuladores y comparándolas para ver cuál de los simuladores resulta más útil para el uso que se le va a dar.

Creación de Modelos

Podemos observar que la creación de modelos en Stage/Player es muy compleja y nos permite definir detalles muy concretos. Por otro lado el simulador STDR permite definir modelos bastante complejos de una manera muy sencilla e intuitiva. Por ello a la hora de definir un modelo concreto, se tardará más tiempo en definir dicho modelo en el entorno Player/Stage que en el entorno de STDR. Sin embargo, en Player/Stage se creará un modelo virtual más parecido al modelo real. En el entorno en el que se va a usar el simulador, la definición de un modelo detallado no es un punto crítico por lo que se puntuará menos este aspecto en la elección de nuestro simulador.

Sensores

En el aspecto de los sensores, Stage/Player ofrece un amplio repertorio de éstos, por otro lado STDR ofrece muy pocos sensores. Player/Stage ofrece los siguientes sensores: láser, sónar, blobfinder, cámara, fiducial, odometría e incluso ofrece la posibilidad de añadir actuadores como pinzas para recoger objetos. Sin embargo, STDR solo ofrece láseres y sónares. Quizás en un principio esto no afecte mucho, pero a medida que el proyecto crece y se quieren añadir nuevas funcionalidades y ofrecer una mayor robustez al robot, esto puede suponer un serio problema y una limitación si no se dispone de los sensores adecuados.

Implementación de Algoritmos

STDR aún no permite incluir sus propios algoritmos de una manera sencilla, para que los ejecute el robot. Para poder ejecutar un algoritmo personalizado en STDR es necesario incluirlo en las librerías de ROS, por lo que STDR es completamente dependiente de ROS. Llegados a este punto podríamos considerar a STDR como una herramienta de simulación para ROS y no como un simulador independiente. Esto se debe en principal medida a que STDR es una herramienta nueva, posiblemente en sus versiones posteriores adopten una nueva técnica, propia para la simulación de algoritmos personalizados. Por otro lado

Stage/Player actúa como un servidor, por lo que nosotros podemos crear nuestro propio cliente de conexión en prácticamente cualquier lenguaje de programación que nos permita el manejo de TCP/IP. Stage/Player es un simulador independiente de ROS, y aunque se puede encontrar también integrado en ROS, sin embargo, se puede usar de manera individual sin ninguna implicación.

3.3.2 Ventajas e Inconvenientes de Stage/Player

Ventajas

- Reduce el tiempo de preparación de los mundos simulados ya que ofrece muchos modelos ya preparados para agilizar el proceso.
- Es sencillo de usar, basta con tener los conocimientos de un estudiante(como es mi caso) para aprender a desenvolverse con el simulador.
- Es complejo en cuanto a sus prestaciones, ya que se utiliza en proyectos de investigación por profesionales del sector.
- Tiene unos requisitos de hardware muy bajos.
- Es software con licencia pública GNU, por lo que es gratuito.
- Posee una gran comunidad de usuarios que aportan códigos que facilita el trabajo con este simulador. Por lo que fomenta el trabajo en equipo y la solidaridad de las personas.

Inconvenientes

- Es muy difícil encontrar documentación para este simulador además de que la mayoría está en inglés y muchas veces te obliga acudir al código fuente para comprender algunas funcionalidades.
- No se puede instalar en Windows. Aunque algunos usuarios afirman haberlo logrado.
- La instalación puede ser larga y tediosa.

3.3.3 Ventajas e Inconvenientes de STDR

Ventajas

- La elaboración de sus componentes simulados se puede hacer a través de un entorno gráfico.
- Tiene un potente entorno gráfico y es muy didáctico.
- Posee buenos tutoriales en su página oficial que explican claramente cómo utilizarlo.
- Instalación sencilla.

- Es software con licencia pública GNU, por lo que es gratuito.

Inconvenientes

- Depende de Ros.
- Está poco optimizado y es muy costoso computacionalmente.
- No está completamente desarrollado, hay funcionalidades de la interfaz gráfica que aún no están desarrolladas.
- El simulador tiene errores, durante las pruebas del simulador que se han realizado se ha quedado colgado el software en alguna ocasión.
- No es muy potente a la hora de desarrollar simulaciones complejas.

3.3.4 Problemas Encontrados en Player/Stage

Instalación de Stage/Player

Al iniciar el proyecto se decidió instalar Player/Stage en Windows ya que se habían encontrado tutoriales en la red de cómo hacerlo. Sin embargo, esto trajo innumerables problemas y finalmente se optó por instalar Ubuntu 14. Aparentemente en Ubuntu era más fácil, no obstante, no resulto ser tan sencillo como se creía. Se encontraron muchos errores de dependencias los cuales finalmente tras mucho esfuerzo no se lograron corregir. La casualidad llevó a probar instalar el simulador en un portátil que tenía instalado Ubuntu 12.04 y al seguir los pasos descritos en el apartado de la Instalación de Player/Stage no se encontró ningún error de dependencias y se logró instalar sin ninguna dificultad. Por lo que antes de comenzar la instalación es recomendable buscar los requisitos de instalación así como la versión de Ubuntu con la que funciona mejor, aunque en teoría Player/Stage es compatible con la versión 14 de Ubuntu en la práctica se ha demostrado que esto no es del todo cierto, a pesar de que se han instalado todos los paquetes que se requerían.

Falta de Documentación

A la hora de trabajar con Player/Stage se ha observado el problema de que hay muy poca información referente a este simulador, por lo que se han tenido muchas dificultades a la hora de empezar a trabajar con él. La mayoría de la información se ha obtenido del código fuente del simulador y de las pruebas realizadas con éste.

3.3.5 Problemas Encontrados en el Simulador STDR

Cargar Mapas

A la hora de cargar mapas a través de la interfaz gráfica se ha detectado el problema de que, el simulador, se queda "colgado" y se necesita "matar" el proceso manualmente por línea de consola. Quizás, esto se deba a que el simulador se encuentra en sus fases iniciales

y aún posee algunos problemas y errores, que posiblemente los desarrolladores solucionarán en un futuro.

Lentitud

Al usar este simulador se comprueba que está mal optimizado, debido al enorme gasto de CPU que se ha observado a la hora de probar el simulador. Esto ha derivado en una lentitud general del PC, provocando un gran retraso en la respuesta del PC ante cualquier acción.

3.3.6 Elección de Simulador

Debido en gran medida a la facilidad que ofrece Stage/Player para incorporar algoritmos personalizados en los robots y a la independencia de ROS se ha optado por utilizar este simulador. Por otro lado Stage/Player también ofrece un repertorio de sensores más amplio y esto podría ofrecer una gran ventaja futura. También está el hecho de que Stage/Player cuenta con un mayor número de usuarios, muchos de los cuales son profesionales del sector que utilizan este simulador para fines de investigación. Para concluir este capítulo se desea resaltar el enorme potencial que posee el simulador STDR, se puede observar que es un simulador muy reciente que aún se encuentra en fase de desarrollo, sin embargo, su potente interfaz gráfica le ofrece una gran oportunidad para superar al resto de simuladores.

Capítulo 4. Técnicas de Robótica Móvil Utilizados

4.1 A*

El algoritmo A* se clasifica dentro de los algoritmos de búsquedas en grafos. Este algoritmo fue presentado por primera vez en 1968 por Peter E. Hart, Nils J. Nilsson y Bertram Raphael. Este algoritmo encuentra el camino de menor coste desde un nodo origen hasta un nodo destino, siempre y cuando se den unas determinadas condiciones.

Para entender el funcionamiento de este algoritmo vamos a considerar primero un algoritmo de búsqueda en grafos sencillo que solo toma en consideración la función de costes. Este algoritmo comienza en un nodo origen y empieza a explorar todos los nodos adyacentes almacenando en cada uno de ellos el coste necesario para llegar hasta dicha posición. De tal manera que realizará una búsqueda en amplitud de todos los nodos hasta dar con el nodo destino. Esto puede resultar bastante ineficiente debido al número de nodos que se explora. Por otro lado el algoritmo a* no amplía todos los nodos adyacentes, sino que amplía el nodo adyacente de menor coste en función de la suma del coste de dicho nodo más el coste de la función heurística en el nodo. Por tanto la exploración de este algoritmo es más eficiente, sin embargo, la eficiencia de este algoritmo depende directamente de lo eficaz que es la función heurística. Así que para lograr un buen resultado en esta técnica se debe realizar una buena función heurística. Una técnica para crear una buena función heurística es tener almacenado en cada nodo la distancia en línea recta desde dicho nodo hasta el nodo destino.

4.1.1 Aplicación de A* en el Proyecto

En un principio se consideró una matriz del tamaño de la imagen, es decir, cada posición de la matriz tenía el mismo tamaño que un pixel de la imagen. En cada posición de la matriz se almacenaba un valor entero 1 si en esa posición el robot colisionaría y 0 en otro caso. Al realizar las pruebas sobre este algoritmo se generaba una ruta mínima entre los dos puntos, sin embargo, al seguir la ruta el robot colisionaba con los objetos del mapa. Esta colisión se debe al hecho de que al seguir la ruta, el robot poseía un tamaño mayor que la ruta en sí. Por tanto aunque la ruta generada pasaba solamente por zonas alcanzables por el robot, los laterales de éste rozaban objetos colisionables.

Para evitar este problema se tomo la decisión de generar una matriz que albergaba en cada posición un mayor número de pixeles. Tras varias pruebas el tamaño que

mejor resultados ofrecía eran los espacios de 10x10 pixeles. De tal manera que cada posición de la matriz poseía un total de 100 pixeles. En caso de que alguno de los 100 pixeles fuera una posición colisionable se consideraba dicha posición equivalente en la matriz con el valor 1 de colisión. Para que una posición de la matriz tuviera el valor 0 de alcanzable ninguno de los 100 pixeles debía ser colisionable. Una vez hecho esto el robot seguía la ruta, pero había casos en los que seguía colisionando con los objetos del mapa. Se intentó incrementar el número de pixeles que almacenaba cada posición de matriz. Sin embargo, al intentar realizar esto, el mapa comenzaba a alejarse mucho de la realidad.

Para arreglar este problema se modificó la función de elección de la ampliación de nodos de la técnica A*. De tal modo que cuando se calculaba el coste de el nuevo nodo a ampliar, se le añadía un nuevo coste. El nuevo coste solo se añadía en el caso de que alguna de las posiciones adyacentes, al nodo que se quería explorar, había un muro o un objeto colisionable. Al aplicar este nuevo requisito la ruta generada se alejaba una posición de las colisiones. Esto se puede observar en la siguiente imagen:

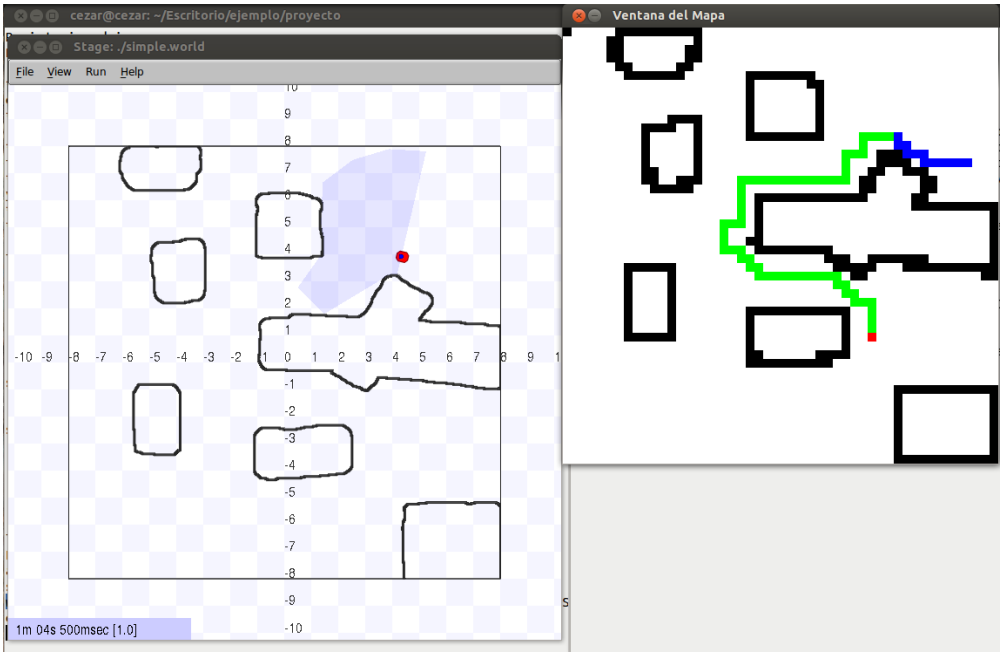


Imagen de ejecución del Planificador de Rutas

En la ventana de la izquierda se puede observar la simulación, y en la pantalla de la derecha se puede observar el mapa generado según las condiciones antes descritas y la ruta que genera el algoritmo A* junto con los nuevos requisitos implementados. En azul se puede observar la parte de la ruta que el robot ya ha recorrido, en verde se puede ver la ruta que ha calculado para recorrer y en rojo el nodo destino.

4.2 Seguimiento de Ruta del Robot

Una vez generada una ruta el robot debe recorrer dicha ruta, para ello se ha desarrollado el siguiente algoritmo:

En primer lugar debemos conocer los datos que obtenemos del robot, más concretamente del filtro de partículas. El filtro de partículas nos facilita la posición en la x e y del robot además de su orientación. La orientación devuelta está en radianes entre 0 PI radianes y 2 PI radianes. Este punto que nos ofrece el filtro de partículas lo consideraremos el punto inicial. El punto final será el punto más cercano al punto inicial de la ruta.

En segundo lugar debemos calcular el ángulo que se forma entre el punto en el que nos encontramos y el punto al que deseamos ir. Esto se calcula con la formula de la arco tangente, la cual es igual al cateto contiguo dividido entre el cateto opuesto. El cateto contiguo será la x inicial menos la x final y el cateto opuesto será la y inicial menos la y final. En el caso de el cateto opuesto se debe multiplicar por -1 para convertirlo a forma trigonométrica estándar, ya que en la imagen la y se incrementa al avanzar hacia abajo, sin embargo en forma trigonométrica estándar esto debe ser al revés.

Una vez calculado el ángulo a través de la formula mencionada se debe convertir a valores entre 0 PI radianes y 2 PI radianes. Para ello se deben considerar los siguientes casos:

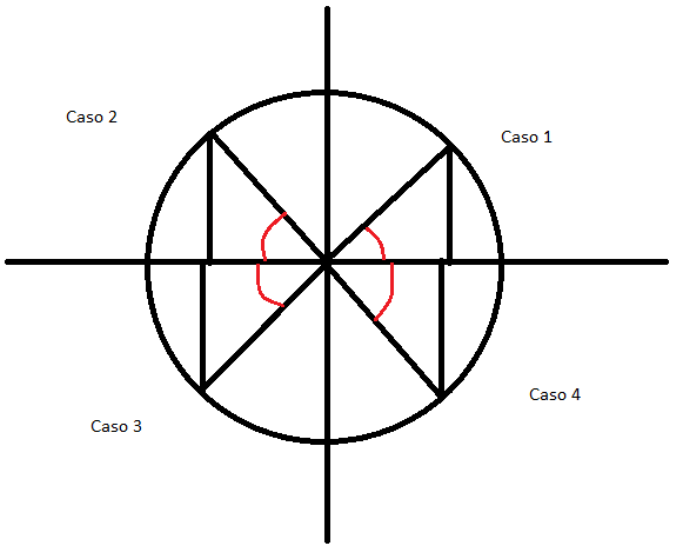


Imagen de los distintos casos de Orientación

- Caso 1: en este caso el ángulo que se genera se deja como estaba.

- Caso 2: en este caso el ángulo se debe convertir de la siguiente manera: PI + ángulo generado. Se debe tener en cuenta que el ángulo generado será negativo, debido a que el cateto contiguo es negativo.
- Caso 3: en este caso el ángulo se debe convertir de la siguiente manera: PI + ángulo generado. En este caso el ángulo generado será positivo ya que tanto el cateto contiguo como el cateto opuesto son negativos.
- Caso 4: en este caso el ángulo se debe convertir de la siguiente manera: 2 * PI + ángulo generado. En este caso el ángulo generado será negativo debido a que el cateto opuesto es negativo.

Una vez obtenido este ángulo correctamente, llamémoslo ángulo deseado y la orientación del robot ángulo actual, debemos aplicar el siguiente pseudocódigo:

```
diff <-- angulo deseado - angulo actual
if |diff| > PI then
  diff <-- |diff| - (2 * PI)
endif
return diff
```

El valor que devuelve se lo podemos proporcionar directamente como parámetro al Player y este modificará la orientación del robot hacia el punto deseado.
Para concluir este apartado se debe concretar que el robot no se puede desplazar y girar al mismo tiempo. Por lo que el robot se debe detener y modificar su orientación hasta que el ángulo "diff" sea menor que una constante llamada "Tolerancia", y en este caso es cuando el robot avanza.

4.3 Técnicas de Localización

La localización de un robot en un mapa, es un problema bien conocido en robótica y estudiado. A la hora de localizar un robot en un mapa se podrían utilizar las técnicas de odometría para saber en qué pose se encuentra un robot. Esta técnica es muy útil y precisa en brazos robóticos, sin embargo, en robótica móvil esto plantea un problema, y es que la odometría acumula un pequeño error cuando el robot se desplaza por el mundo y con el tiempo este error se va incrementando hasta que el robot cree que se encuentra en una posición que nada tiene que ver con la realidad. Una de las principales causas de este error es el hecho de que las ruedas pueden resbalar por el suelo, hecho que la odometría no contempla. Por tanto, en robótica móvil se deben emplear nuevas técnicas, ya que las técnicas tradicionales de odometría no sirven. Las técnicas de localización en robótica móvil más utilizadas en la actualidad son: Los Filtros de Kalman, y Los Filtros de Partículas. Estas técnicas tratan la incertidumbre de el mundo real a través de medios probabilísticos y usando como base las reglas de Bayes.

4.3.1 Filtros de Kalman

El filtro de Kalman se inventó en 1960, en un documento en el que Kalman describe una solución recursiva para el problema del filtrado lineal de datos discretos. La derivación de Kalman fue dentro de un amplio contexto de modelos de espacio de estados, en donde el núcleo es la estimación por medio de mínimos cuadrados recursivos. Desde ese momento, debido en gran parte al avance en el cálculo digital, el filtro de Kalman ha sido objeto de una extensiva investigación y aplicación, particularmente en el área de la navegación autónoma y asistida, en rastreo de misiles y en economía.

El filtro es un procedimiento matemático que opera por medio de un mecanismo de predicción y corrección. En esencia este algoritmo pronostica el nuevo estado a partir de su estimación previa añadiendo un término de corrección proporcional al error de predicción, de tal forma que este último es minimizado estadísticamente.

El proceso de estimación completo es el siguiente: El modelo es formulado en estado-espacio y para un conjunto inicial de parámetros dados, los errores de predicción del modelo son generados por el filtro. Estos son utilizados para evaluar recursivamente la función de verosimilitud hasta maximizarla.

4.3.2 Filtros de Partículas

En el caso de las técnicas de muestreo, como pueden ser los algoritmos de filtros de partículas, se basan en asumir aleatoriamente la existencia de valores para algunos nodos y luego usa restos valores para inferir en la cantidad de los otros nodos. Luego, se mantienen estadísticas de los valores que estos nodos toman, y finalmente estas estadísticas dan la respuesta. A estos métodos se les llama técnicas de Monte Carlo.

El uso de los métodos de Monte Carlo como herramienta de investigación proviene del trabajo realizado en el desarrollo de la bomba atómica durante la segunda guerra mundial en el Laboratorio Nacional de los Álamos en EE.UU. Fue en 1930 cuando Enrico Fermi y Stanislaw Ulam desarrollaron las ideas básicas del método, más tarde, a principios de 1947 Jhon Von Neumann envió una carta a Richtmyer a Los Álamos en la que expuso de modo exhaustivo tal vez el primer informe por escrito del método de Monte Carlo. Una de las primeras aplicaciones de este método a un problema determinista fue llevada a cabo en 1948 por Enrico Fermi, Ulam y Von Neumann cuando consideraron los valores singulares de la ecuación de Schrödinger.

El problema fundamental al que se enfrenta es encontrar el valor esperado de alguna función con respecto a una distribución de probabilidad. Aquí, la variable puede estar formada por variables discretas, continuas, o alguna combinación de ambas.

La idea principal de este algoritmo se basa en representar la función de densidad de probabilidad de la posición del robot mediante un conjunto finito de partículas, donde la variable representa la ubicación del robot y la función su probabilidad asociada, calculada mediante un modelo de observación.

Inicialmente, el conjunto de muestras se distribuye uniformemente de forma aleatoria por todo el espacio de posiciones posible. Las muestras se generan en cada instante, a partir de las muestras del instante anterior, junto con las mediciones del entorno y las acciones realizadas por el robot siguiendo tres pasos recursivamente:

- **Predicción**: Se desplazan las muestras según la acción ejecutada y se estima la nueva posición de las muestras. Para hallar el nuevo conjunto de partículas hay que tener en cuenta el conjunto de partículas del instante anterior y la acción realizada.
- **Actualización de Observaciones**: Se calculan las probabilidades *a posteriori* de cada muestra, dada la última observación sensorial. En este apartado se calcula la probabilidad de cada partícula en función de la verosimilitud de las observaciones realizadas desde las partículas generadas en el apartado de predicción con la observación real del robot.
- **Remuestreo**: Se genera un nuevo conjunto de muestras con distribución proporcional a la verosimilitud de las muestras anteriores. Se construye el nuevo conjunto de muestras, remuestreando con sustitución el conjunto actual de muestras de forma que se escoja cada muestra con probabilidad proporcional a la verosimilitud de la misma. Finalmente se normalizan las probabilidades de las muestras.

4.3.3 Elección de Técnica

La técnica que se ha decidido utilizar es el filtro de partículas y esto se debe a varias razones:

- El filtro de partículas es más sencillo de implementar que el filtro de Kalman.
- Aunque el filtro de partículas tiene una complejidad exponencial frente a la complejidad polinomial del filtro de Kalman, esto no se nota demasiado en un entorno 2D, dónde solo tenemos 3 variables. Sin embargo si nos encontráramos en un entorno 3D dónde tenemos 6 variables, sería seriamente recomendable usar el filtro de Kalman, pero en nuestro caso no es necesario.
- El filtro de partículas suele dar mejores resultados que el filtro de Kalman en casos en los que se dispone de menos información.

4.4 Filtro de Partículas

En este apartado se mostrará la aplicación práctica del filtro de partículas y la unión con el resto de las técnicas. Como ya se ha explicado previamente el filtro de partículas sirve para localizar el robot en el mapa cuando los sensores no son fiables, ni el movimiento del robot. Esta técnica se divide en varios apartados que se expondrán a continuación:

- Inicialización de las partículas.

- Mover partículas.
- Cálculo de los pesos de las partículas.
- Remuestreo de las partículas.

A continuación se explicará detalladamente cómo funciona cada uno de estos apartados.

4.4.1 Inicialización de las Partículas

Antes de explicar la inicialización de las partículas se debe tomar la decisión del número de partículas que se va a utilizar. Se ha decidido usar como número total de partículas el alto más el ancho del mapa en el que se colocarán. Como el mapa que se usa posee un tamaño de 500x500 se usarán 1000 partículas.

Para la inicialización de las partículas se propusieron dos técnicas. La primera es bien extendida y trata de colocar las partículas de manera aleatoria en el mapa, es decir, la pose de las partículas (x, y, orientación) se declara de manera aleatoria dentro de las restricciones de cada una. La segunda técnica trata de utilizar la pose inicial del robot que ofrece la odometría, ya que es el único dato 100% fiable que nos ofrecen los sensores. Al iniciar Stage/Player el robot se inicia con una pose en el mapa la cual es conocida por la clase Position2dProxy. Sin embargo esta técnica posee una restricción y es que si iniciamos el cliente y mueve el robot, este movimiento genera un error en la odometría. Si luego se apaga el cliente y se vuelve a iniciar la pose que ofrecerá la odometría ya no es de fiar. El único valor de fiar es el que ofrece la odometría a la hora de iniciar el simulador. Pero como no se sabrá en que caso se encontrará el robot se va a suponer la primera técnica propuesta.

4.4.2 Mover Partículas

Una vez inicializado entraremos en un bucle infinito que terminará cuando el cliente se detenga. El primer paso que se realizará en este bucle será desplazar las partículas. La clase partícula tendrá una función que las desplazará y su funcionamiento será el siguiente:

Para que está función tenga éxito debe conocer los siguientes datos, velocidad de giro, velocidad de desplazamiento y tiempo transcurrido desde la última vez que se han desplazado.

Por simplicidad se ha supuesto que el robot si gira no se desplaza y si se desplaza no gira.

Primero se calcula la distancia que se supone que ha tenido que recorrer y se le aplica ruido, luego la orientación en la que se supone que debería estar y se le aplica ruido.

En caso de solo girar se conoce el nuevo ángulo en el que debería estar por lo que se le aplica ruido. El ruido es un valor aleatorio de una gaussiana, que posee como parámetro los datos a los que se le desea aplicar ruido.

4.4.3 Calcular Peso de las Partículas

Para calcular el peso de las cada partícula estas poseen una función llamada calcular_peso la cual recibe como parámetro las lecturas del laser. El laser posee 10 rayos por lo que recibirá un array de tamaño 10 en el que está almacenada la distancia que ha medido cada rayo. Ahora se deben obtener las medidas del sensado de la partícula, para ello se invoca a la función llamada sensado, la cual calcula las medidas de los 10 rayos del laser para la partícula en concreto. Una vez obtenidos estos valores se resta cada medida de la partícula con su correspondiente medida del robot para calcular el error. El error calculado para cada rayo se introduce en una gaussiana de parámetro error y ruido, y se calcula el error con ruido de la medida. Cada error con ruido de cada medida se va acumulando en una variable que representará el error total y este será el valor que devolverá la función que calcula el peso de una partícula en concreto.

4.4.4 Remuestreo de las Partículas

La función de remuestreo lo que hace es eliminar las partículas menos probables y incrementar el número de las partículas más probables. Para ello necesita calcular el peso de las partículas previamente. Esta función se realiza de la siguiente manera:

Primero normaliza los pesos de las partículas mediante la división de cada partícula por la suma de todos los pesos de las partículas. La suma de todos los pesos está optimizada de tal manera que se calcula a medida que se calcula el peso de cada partícula. Esto es, en la clase FiltroParticulas se encuentran un atributo, peso_total, el cual se inicializa a 0 cuando se calcula los pesos y los valores de éste se modificarán a medida que se calcule el peso de cada partícula.

En segundo lugar se realiza el siguiente pseudocódigo:

```
index = 0
b = 0
do i = 0 to 1000
 b = b + random * peso_maximo
 encontrado = false
 while no encontrado do
  index = index % 1000
  if peso[index] < b then
   b = b - w[index]
   index = index + 1
  end if
  else
   añadir partícula al nuevo array de partículas
   encontrado = true
  end else
 end while
```

end do

Este es el pseudocódigo que realiza el remuestreo, y lo que hace es:

Primero se inicia un bucle para las 1000 partículas, en cada iteración se añade una nueva partícula al nuevo array de partículas.

Dentro del bucle se genera primero un valor aleatorio entre 0 y el peso máximo de una partícula. Para entender mejor lo que ocurre en el bucle se explicará con un ejemplo:

Supongamos que el valor aleatorio que ha generado b es 0.1 (lo cual es un valor muy alto pero se usa para mayor simplicidad), si el peso de la partícula que se está observando en este momento es de 0.07, esta partícula no se añade al nuevo array, se incrementa el índice y el valor de b es ahora 0.03. Ahora supongamos que la siguiente partícula que se mira es 0.05, como b tiene un valor inferior que es 0.03 esta partícula se añade al nuevo array. Ahora supongamos que el nuevo valor aleatorio es 0.01, sumado al valor anterior es 0.04, lo cual sigue siendo menor que el peso de la partícula que estamos mirando, por lo que se vuelve a añadir esta partícula al nuevo array y se vuelve a generar un valor aleatorio, y así sucesivamente hasta generar las nuevas 1000 partículas.

4.5 Unión de las Técnicas

Al juntar estas técnicas nuestro robot funcionará de la siguiente manera:

1. Se le facilitará un punto final al que desea llegar.
2. El filtro de partículas generará una posición en la que cree que está, que será la partícula más probable.
3. Se invocará al algoritmo A* con los parámetros, punto inicial y punto final. Donde punto final es el punto al que hemos decidido que se dirigirá nuestro robot y el punto inicial es el punto que nos ha ofrecido el filtro de partículas. Este algoritmo solo devolverá el siguiente punto de la ruta que está más cercano al actual. Cada vez que el punto actual alcance el punto de la ruta o se aleje demasiado del punto de la ruta, la ruta se vuelve a calcular.

Esto se repetirá hasta que nuestro robot haya alcanzado la posición final planteada.

Capítulo 5. Conclusiones y Trabajos Futuros

Para concluir este proyecto me gustaría destacar que los objetivos iníciales propuestos eran muy ambiciosos quizás estaban muy lejos de la realidad. Sin embargo me encuentro profundamente orgulloso de lo que he logrado realizar en este proyecto y de todas las nuevas técnicas que he aprendido a utilizar. Antes de este proyecto mis conocimientos referentes a la robótica estaban muy limitados, esto también se debe al hecho de que en el Grado de ingeniería informática no se explica profundamente este tema. Aunque he de reconocer que la asignatura óptativa de Robótica Computacional me ha ofrecido unas buenas bases teóricas para poder introducirme en este maravilloso mundo de la robótica. Con la realización de este proyecto he logrado empezar en el mundo de la robótica e iniciarme con una base solida en este campo. Me gustaría seguir estudiando este campo en un futuro y ampliar este proyecto desarrollando un SLAM completamente funcional, quizás en un mundo 3D donde el nivel de dificultad es mayor, pero espero que cuando lo intente tenga muchos más conocimientos en el tema.

Lo que se ha conseguido en este proyecto es planificar una ruta, darle la capacidad al robot para que siga dicha ruta y localizar el robot en el mapa, logros de los cuales estoy orgulloso, aunque me hubiera gustado implementar más técnicas.

En este proyecto se deseaba desarrollar algún tipo de inteligencia colectiva o de enjambre, sin embargo, no se pudo llegar a abordar este tema. En un futuro se podría implementar la idea inicial de un SLAM distribuido, donde cada robot crea su mapa y sus propios "landmarks" y lo comparte con los demás generando así un mapa del entorno todos juntos. Esto no es una idea descabellada de hecho en la actualidad se está investigando el tema del SLAM distribuido, espero algún día poder trabajar en este área y aportar nuevas e innovadoras técnicas.

Capítulo 6. Summary and Conclusions

To conclude this project I would like to highlight that the initial objectives was very ambitious and maybe they was far from reality. Therefore, I find myself deeply proud to the things that I achieved in this project and the all new techniques which I learned. Before starting this project my knowledge referred to robotic science was quite limited, this is because in the career of informatics engineer is not explained deeply. Although I must recognize that the Computational Robotic subject offers me a good theoretical bases which make me able to introduce myself in the wonderful robotic world. With the development of this project I achieved to get a more solid bases in this field. I would like to continue learning this field in the future and expand this project developing a full functional SLAM problem, maybe in a 3d world where the difficult level is higher, but I expect that when I will try it I will have much more knowledge in this topic.

What has been achieved in this project is to plan a route, make robot able to follow that route and localize the robot in the map, I am very proud of those achievements, but I would like to implement more techniques.

In this project it was desired some kind of swarm intelligence, however, has not been reached to address this topic. In the future could be implemented the initial idea of distributed SLAM, where any robot create his individual map and his individual "landmarks" and share them with the rest of the robots, making together a lonely map of their environment. This is not a crazy idea, in fact today there are researching in process of this topic of the distributed SLAM, I hope that one day I will work on this area and I will perform news techniques.

Capítulo 7. Presupuesto

Este proyecto no supone un gran coste económico, debido a que el software empleado ha sido bajo licencia pública GNU, por lo que es gratuito. El desarrollo del proyecto tampoco supone un gasto económico en equipos especializados para este fin. Por lo que el valor económico de este proyecto lo suponen las horas empleadas en su desarrollo, teniendo en cuenta que para un recién titulado en España el sueldo medio es de unos 1100€ (con suerte) lo que equivale a unos 6,875€/h y estimando que han empleado unas 320 horas en el desarrollo de este proyecto el presupuesto sería de unos 2200€.

Bibliografía

[1] Jenifer Owen (2013), *How to use Player/Stage 2nd Edition (Player 3.0.2 & Stage 3.2.2)*.

[2] Kevin (2012), *How to install Player/Stage on Ubuntu 12.04 LTS 32bit (Player 3.0.2 & Stage 3.2.2).*

[3] Azucena Fuentes Ríos, Localización y Modelado Simultáneos en ROS Para la Plataforma Robótica Manfred. *Octubre 2012*.

[4] Richard Vaughan. Massively Multiple Robot Simulation in Stage. Swarm Intelligence (2008).

[5] Richard T. Vaughan and Brian P. Gerkey, On device abstractions for portable, reusable robot code. Proceeding of the IEEE/RSJ International Conference on Intelligent Robot System (IROS 2003).

[6] A.R.D.E. Tutorial: Player/Stage/Gazebo (1 mar 2012).

[7] Stage Official Web Side User Guide.

[8] Player Official Web Side User Guide.

[9] STDR Official Web Side.

[10] Rasko Pjesivac, Player-Stage Tutorial.

[11] Tatiana Macchiavello, robótica (2008).