GROSELLA

(Graphical ROS Experiment Loader and LAuncher)

Memoria del proyecto de final de carrera correspondiente a los estudios de Ingeniería Superior en Informática presentado por Miguel Angel Blanco Muñoz y dirigido por Ricardo Toledo Morales.

Bellaterra, Septiembre de 2011

El firmante, Ricardo Toledo Morales, profesor del departamento de Ciencias de la Computación de la Universidad Autónoma de Barcelona

CERTIFICA:

Que la presente memoria ha sido realizada bajo su dirección por Miguel Angel Blanco Muñoz

Bellaterra, Septiembre de 2011

Firmado:

A mis padres.

Agradecimientos

Agradezco al Dr. Ricardo Toledo Morales, Coordinador de grupo de R+D en el Centro de Visión por Computador y profesor del departamento de Ciencias de la Computación de la Universidad Autónoma de Barcelona, por brindarme la oportunidad de realizar este proyecto.

También al Dr. Yiannis Demiris director de BioART (Bioinspired Assistive Robots and Teams) en el Imperial College de Londres, por proporcionarme un lugar en su laboratorio para el desarrollo del proyecto.

Quiero agradecer especialmente a Arturo Ribes Sanz, estudiante de PhD en el Instituto de Investigación en Inteligencia Artificial de la Universidad Autónoma de Barcelona, por su supervisión del proyecto, incansable ayuda, consejos y convivencia.

Índice general

Índice de figuras

Capítulo 1

Introducción

1.1. Introducción

El mundo de la Robótica ha evolucionado mucho y muy rápido en las últimas décadas, haciéndose un hueco importante en otras disciplinas. La alta variedad de robots disponibles en el mercado hace necesario desarrollar librerías que facilitan la tarea de programación.

En los últimos años, han aparecido muchos *frameworks* para la comunicación con robots, permitiendo el uso de gran variedad de sensores y multitud de algoritmos. Estos *frameworks* o *middlewares* son Player [1], YARP [2], Orocos [3], CARMEN [4], Orca [5], MOOS [6], and Microsoft Robotics Studio [7].

Por otro lado tenemos ROS (Robot Operating System) [8], que es un meta-sistema operativo de código abierto para robots, que proporciona abstracción de *hardware*, control de dispositivos a bajo nivel, paso de mensajes entre procesos, y administración de paquetes. También proporciona herramientas y librerías para obtener, crear, escribir, y ejecutar código.

Para el desarrollo del proyecto se ha elegido ROS, por la amplia documentación y recursos disponibles, su gran comunidad y por su integración en lenguajes de programación como Python y C++. También posee fácil integración en otros

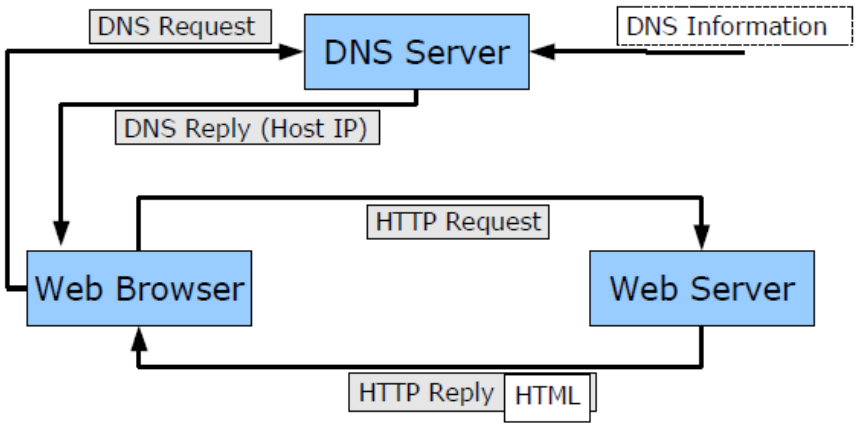

Figura 1.1: Ejemplo de comunicación en Internet

sistemas como por ejemplo, Player u Orocos. De esta manera un nodo desarrollado para ROS, puede ser utilizado por Player u Orocos. Básicamente el funcionamiento de ROS está basado en el paradigma del productor/consumidor, existen nodos que requieren datos para su uso, y otros que los generan. Para todo este intercambio de información es necesario una plataforma, ROS utiliza unos mecanismos muy conocidos en el mundo de los Sistemas Operativos, que son los *sockets* y los procesos. Por lo tanto podemos decir que ROS es una red de intercambio de información entre procesos. Podemos ver el paralelismo entre la comunicación en Internet (Figura 1.1) y la comunicación en ROS (Figura 1.2).

1.2. Motivaciones

ROS proporciona muchas herramientas para la gestión y visualización de información que producen los sensores de los robots. Crear un experimento requiere modificar parámetros del mismo para su posterior evaluación y rectificación. ROS aunque es un sistema muy potente, carece de un paquete que englobe todas las tareas necesarias para la experimentación.

La principal razón que me ha motivado a realizar este proyecto es la oportuni-

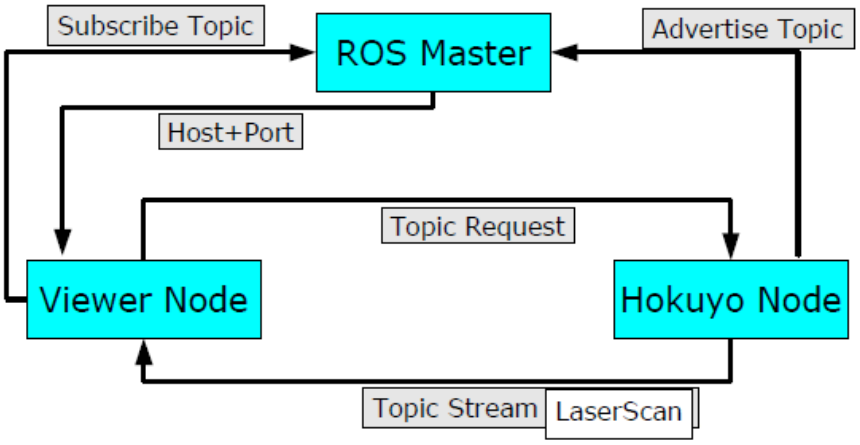

Figura 1.2: Ejemplo de comunicación en ROS

dad de suplir las carencias descritas. Además, me llena poder aportar un granito de arena a la investigación de la Visión por Computador y la Inteligencia Artificial, dado que el proyecto va a ser utilizado por Arturo Ribes Sanz en sus experimentos con robots.

Por otro lado es una meta para mi realizar un buen proyecto, que consta de la búsqueda de soluciones a problemas reales, diseñar, implementar, documentar y presentar, al igual que se podría hacer en una empresa.

1.3. Objetivos

El objetivo principal es crear una herramienta que facilite gráficamente el diseño de experimentos para su posterior uso en los robots móviles del Instituto de Investigación en Inteligencia Artificial (IIIA)-Consejo Superior de Investigaciones Científicas (CSIC) y el Imperial College of London. Será utilizada para la edición de la configuración para su posterior ejecución e interpretación visual mediante gráficas de los experimentos.

ROS [8] será la plataforma usada para la comunicación con el robot y el marco para el desarrollo del proyecto. Algunos experimentos necesitan funcionalidades

que ROS no proporciona. Por esta razón, la segunda parte del proyecto consiste en la realización de un *package* para ROS. Esta parte del proyecto será empaquetado en un *stack*, para su posterior publicación en línea para su libre uso y distribución por la comunidad ROS.org.

Así pues, el proyecto constará de dos partes, una interfaz gráfica y un nodo de ROS que aportará funcionalidades extra.

1.3.1. Interfaz Gráfica

La interfaz gráfica se desarrollará en Python, mas concretamente en wxPython, por su versatilidad y fácil uso en el ámbito de creación de interfaces gráficas de usuario. Además en el *package rx* de ROS está hecho en wxPython y resultará mas cómodo para consultar. Se hará orientado a objetos, para una posible modificación del sistema ROS en el futuro, y así poder adaptar la aplicación fácilmente.

1.3.2. Nodo de ROS

Para poder realizar ciertos experimentos en robots es necesario calcular el Optical Flow. Esta técnica proporciona el campo vectorial que indica el movimiento entre dos imágenes consecutivas. Como ROS no lo implementa, se ha decidido que la segunda parte del proyecto constará de la realización de un nodo para el cálculo del Optical Flow [32] a partir de las imágenes proporcionadas por el robot. Para visualizar su correcto funcionamiento, se hará un *plug-in* en la interfaz gráfica.

1.4. Organización de la memoria

El resto de la memoria se organiza de la siguiente manera. En el segundo capítulo, se expone todo los conceptos necesarios para la comprensión de las herramientas que se dispone para desarrollar el proyecto. El tercer capítulo, informa

al lector de la propuesta de solución, los recursos utilizados y la planificación temporal del proyecto. El cuarto capítulo, explica la especificación de roles de los módulos y una descripción de las clases implementadas. Por último en conclusiones, se comentan la reflexión general y las aplicaciones futuras.

Capítulo 2

Conocimientos previos

Para comprender la propuesta de solución descrita en el siguiente capítulo, es necesario tener unos conocimientos previos de ROS [8] y también las herramientas/librerías utilizadas en el desarrollo de la aplicación. Para el uso de la aplicación hay que instalar ROS [9], matplotlib [25], OpenCV [28] y wxPython [16].

2.1. ROS

En lo que respecta al proyecto hay que conocer dos partes importantes del sistema ROS. La primera sección, su sistema de ficheros para reconocer la organización física de los interventores del sistema, que se describen en la segunda sección. La tercera menciona las herramientas básicas de ROS. Hay que tener en cuenta que el funcionamiento de sus herramientas difiere en algunos aspectos a su API [10].

2.1.1. Sistema de ficheros ROS

Los conceptos utilizados en el sistema de ficheros de ROS son:

Packages(Paquetes) : Son la unidad principal de organización del software en

ROS. Un *package* contiene nodos (procesos), librerías de desarrollo, sets de datos, ficheros de configuración, o cualquier otro elemento que sea útil para organizarlo todo junto.

Manifests(manifest.xml) : Proporcionan meta-datos sobre un *package*, incluyendo su información de licencias y dependencias, además de la información sobre el lenguaje específico y los parámetros de compilación.

Stacks(Pilas) : Son colecciones de *packages* que proporcionan funcionalidades añadidas. Los *stacks* son también la manera de publicación de software en ROS y tiene asociados sus números de versión.

Stack Manifests(stack.xml) : Al igual que *manifests* pero orientado a *stacks*.

Message (msg) : Contiene la descripción de los mensajes, definen las estructuras de datos de los mensajes que se envían en ROS.

Service (srv) : Contiene la descripción de los servicios, definen las estructuras de datos de las peticiones y las respuestas de los servicios de ROS.

2.1.2. Grafo de comunicación ROS

El grafo de comunicación de ROS es una red punto a punto de procesos de ROS que procesan datos juntos y conforman el grafo. Estos procesos son:

Nodes : Son procesos que llevan a cabo el cómputo. ROS está diseñado modularmente a pequeña escala, un sistema de control de un robot tendrá por lo tanto muchos nodos.

Master : El ROS Master proporciona el registro de nombres y las consultas al resto del grafo. Sin el Master, los nodos no serían capaces de encontrarse el uno al otro o no podrían invocar servicios.

Messages : Los nodos se comunican entre ellos mediante el paso de mensajes. Un mensaje es simplemente una estructura de datos, que contiene campos. Es-

tos campos son tipos estándar primitivos (entero, punto flotante, booleano, etc.).

Topics : Los mensajes son encaminados vía un sistema de transporte donde publican/suscriben. Un nodo envía un mensaje publicándolo en un *topic*. El *topic* es un nombre identifica el contenido del mensaje. Un nodo que está interesado en una cierta clase de datos se suscribirá al *topic* apropiado. Pueden haber múltiples editores simultáneos y suscriptores para un solo *topic*. Un nodo puede publicar y/o suscribirse a múltiples *topics*.

Services : El paradigma de editor/suscriptor es un modelo muy flexible, pero es de muchos a muchos, y el transporte unidireccional no es apropiado para interacciones de petición/respuesta, que a menudo requieren un sistema distribuido. Estas peticiones/respuestas las gestionan los servicios, definidos por pares de estructuras de mensajes, uno de petición y otro de respuesta.

Bags : Es el formato para guardar y reproducir mensajes de ROS. Es un mecanismo de almacenamiento de datos, como los de los sensores, que aunque dificultan su recolección son necesarios para el desarrollo y puesta en practica de algoritmos.

2.1.3. Herramientas de visualización y gestión de datos en ROS

Para el desarrollo de sistemas autónomos en ROS es necesario crear interactuación entre todos los componentes descritos en las dos secciones anteriores. La visualización de esas relaciones en ROS la proporciona la herramienta *rxgraph* que muestra el grafo de comunicación. Si se quiere ver la relación que puedan tener los datos que fluyen en el sistema, ROS dispone de una herramienta para el dibujo a tiempo real de los mismos llamada *rxplot*. También existen varias herramientas por línea de comandos como *rostopic* y *rosnode* que proporcionan información de los *topics* y *nodos* que intervienen respectivamente. Para ejecutar un nodo está *rosrun* y para un conjunto de nodos *roslaunch*.

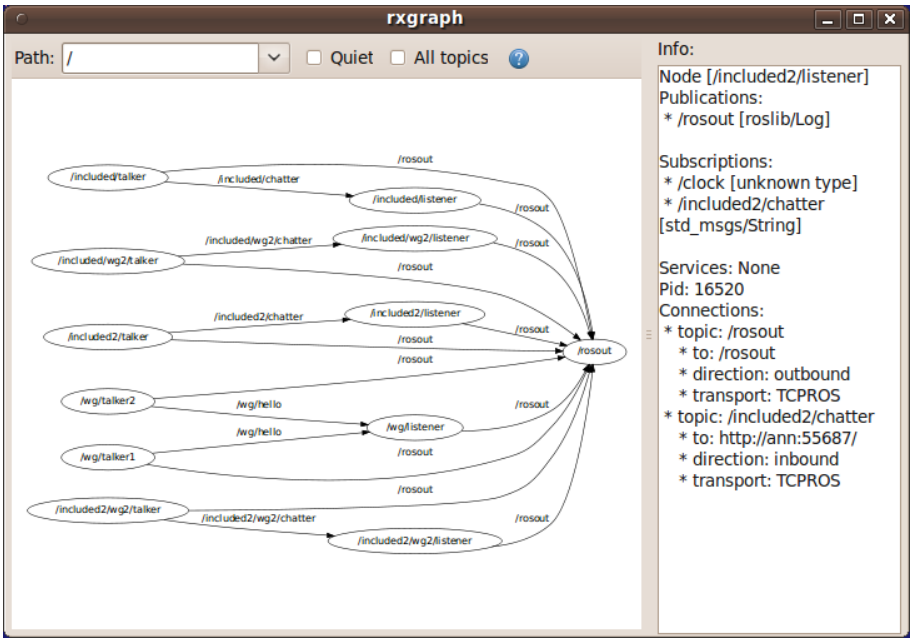

Figura 2.1: rxgraph

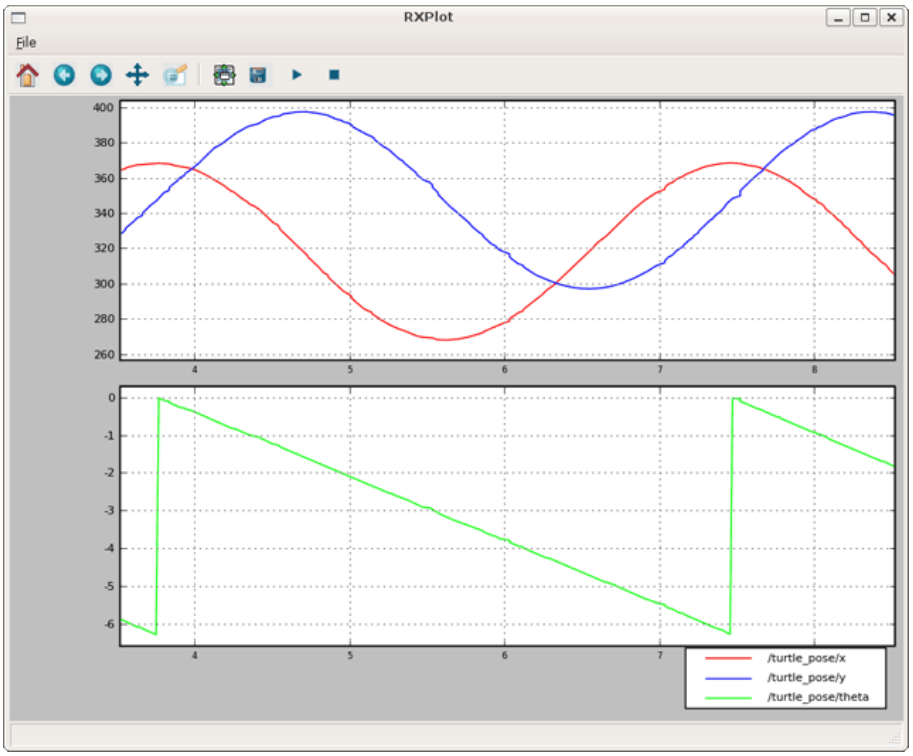

Figura 2.2: rxplot

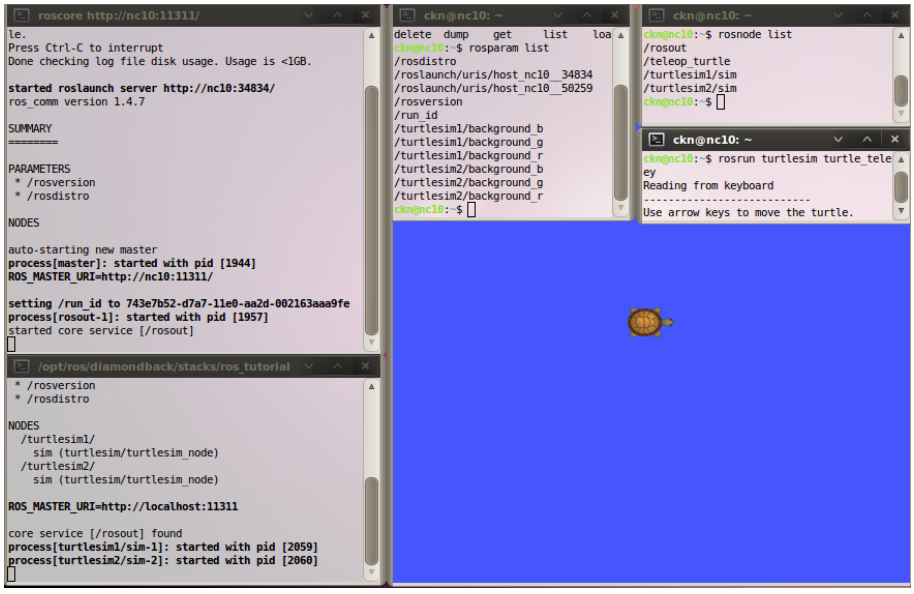

Figura 2.3: Terminales abiertas para el uso de ROS

El problema derivado de la existencia de todas estas herramientas es que resulta muy engorroso gestionar tantas *terminales* abiertas. Se puede observar en la Figura 2.3, que para lanzar una configuración sencilla del simulador *turtlesim* que posee ROS, mostrar los nodos activos y sus respectivos parámetros, acaba siendo algo complicado su gestión. Si se pretende hacer un experimento en el que intervengan muchos nodos y se quieren visualizar gráficamente los datos, acaba siendo un trabajo imposible.

2.2. wxPython

wxPython [15] es un *binding* en Python [11] [12] de la librería wxWidgets [14] para la programación de GUIs (Grafical User Interface) para múltiples plataformas, GNU/Linux, Mac OS X y Windows. Es de código abierto y posee una gran comunidad de desarrollo, además de una gran cantidad de tutoriales que facilitan su comprensión y uso.

El eje principal de una aplicación es la clase wx.Frame, dentro de ella pueden

convivir varios wx.Panel y dentro de estos es donde se alojan todos los *widgets*.

Los *widgets* son las pequeñas piezas que componen una aplicación y sirven para la interactuación entre el usuario y la lógica interna del programa. Existe una gran cantidad de *widgets* como por ejemplo árboles (TreeCtrl), cuadros de texto(TextCtrl) y botones(Button), que son los que se han usado básicamente.

Hay una gran cantidad de información sobre wxPython, aunque se han utilizado más referencias, estas son las básicas [17] [18] [19] [20].

2.3. matplotlib

Matplotlib [24] es una biblioteca para la generación de gráficos 2D a partir de datos contenidos en listas o *arrays* en el lenguaje de programación Python y su extensión matemática NumPy. Matplotlib es multiplataforma y se puede utilizar en GNU/Linux, Mac OS X y Windows.

Se pueden generar gráficos de barras, diagramas de dispersión e histogramas de una manera sencilla en pocas líneas de código. Proporciona una API [26], muy parecida a la de MATLAB. El hecho que la sintaxis sea parecida a la de MATLAB, hace mas sencilla la tarea de programación en matplotlib.

2.4. OpenCV

OpenCV [27] es una biblioteca libre de visión artificial originalmente desarrollada por Intel. OpenCV es multiplataforma, existiendo versiones para GNU/Linux, Mac OS X y Windows. Contiene más de 500 funciones en su API [29] que abarcan una gran gama de áreas en el proceso de visión, como reconocimiento de objetos (reconocimiento facial), calibración de cámaras, visión estéreo y visión robótica.

El proyecto pretende proporcionar un entorno de desarrollo fácil de utilizar y altamente eficiente. Esto se ha logrado, realizando su programación en código C y C++ optimizados, aprovechando además las capacidades que proveen los proce-

sadores multinúcleo. OpenCV puede además utilizar el sistema de primitivas de rendimiento integradas de Intel, un conjunto de rutinas de bajo nivel específicas para procesadores Intel (IPP).

Capítulo 3

Propuesta de Solución

Hemos visto que son muchas las posibilidades y herramientas que ofrece ROS para el desarrollo de sistemas autónomos con el mismo sistema. Para el uso óptimo de estas herramientas es necesaria una herramienta para englobar de una manera intuitiva todo el proceso de diseño de experimentos. Por eso se ha pensado en realizar este proyecto, un centro de control para todas las prestaciones ya comentadas.

3.1. Descripción de la propuesta

GROSELLA constará de una serie de herramientas para la visualización y ejecución de experimentos sobre el sistema ROS. Los requisitos básicos son:

Topics : Los *topics* son el motor del sistema, envían y reciben mensajes, por eso se mostrará un árbol con los *topics* activos.

Mensajes : Los *topics* contienen atributos y sus tipos, para saber que contienen, se mostrará información de los topics(publishers/subscribers).

Suscripciones : Para acceder a la información de los *topics*, se podrán hacer suscripciones a los distintos mensajes de un *topic*, en función de si son graficables o no.

Gráficas : La visualización de los datos es una buena manera de entenderlos, para ello se mostrarán gráficas de las suscripciones hechas.

Lanzadores : A partir de una lista de los nodos y sus parámetros se podrán crear y lanzar *launchers*.

3.2. Recursos

Para la realización del proyecto no será necesario ningún recurso extra, a los descritos seguidamente, con lo cual no será necesario ningún gasto. Para la implementación se dispone de un portátil propio (Samsung nc10) con Eclipse y su entorno PyDev para el desarrollo en Python.

Para poder hacer pruebas del funcionamiento, se utilizarán tanto el robot *Pioneer 2-AT* del IIIA-CSIC como el *Pioneer Peoplebot* de BioART (Imperial College of London).

Todas las librerías que se utilizarán son de código abierto, ROS [8], wxPython [15] y OpenCV [27], por lo tanto tampoco añaden ningún gasto adicional.

Como interprete para pruebas se ha usado iPython [13].

Para la edición de gráficos que contiene la memoria se ha usado GIMP. Para hacer la planificación del proyecto se ha utilizado Planner. Para el dibujo de diagramas se ha usado DIA.

La memoria se ha hecho en LaTeX [21] con su editor multiplataforma y libre TexMaker [22]. Para la comprensión del uso de LaTeX se ha utilizado el curso [23].

3.3. Planificación

En la Figura 3.1 se muestra la planificación temporal del proyecto que está pensado para realizarlo en seis meses. Se puede observar, que se tardará un mes

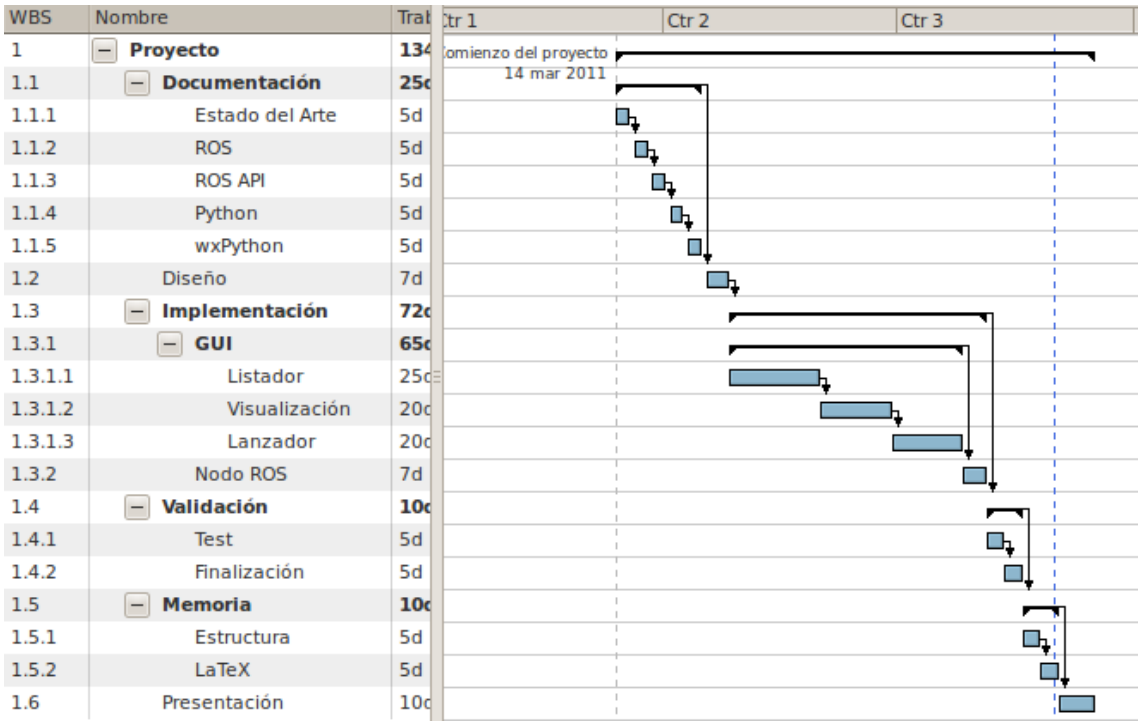

Figura 3.1: Planificación del proyecto

en la documentación, y la mitad de la planificación la ocupa la implementación. La validación, memoria y presentación se harán durante los dos últimos meses.

Capítulo 4

Desarrollo del proyecto

4.1. Diseño de la solución

4.1.1. Interfaz gráfica

La interfaz gráfica se divide en tres grandes módulos basados en la planificación. Esta división está basada en la lógica del programa y la manera de interactuación con el usuario. Estos módulos son el Visualizador, el Lanzador y el Listador. Las responsabilidades de cada módulo son las siguientes.

Lanzador : Gestiona la ejecución de nodos para un experimento. Se pueden configurar los parámetros. Se puede cargar un *launcher*, en XML con el mismo formato de ROS, usado por *roslaunch*. Se pueden crear *launchers* desde cero o añadir nodos y cambiar parámetros a uno cargado y guardar los cambios en el mismo formato. Se puede ejecutar y parar los experimentos.

Listador : Gestiona la lista de *topics* activos del experimento en ejecución. Permite seleccionar atributos de un *topic* y graficarlos. Según el tipo de datos se grafican de una manera u otra si es que se dispone de alguna. Proporciona información sobre el tipo de datos, sus atributos y si son atributos compuestos también sus derivados. Se actualiza a tiempo real para monitorizar

cambios en el sistema, como por ejemplo si se lanza algún nodo o se mata.

Visualizador : Se encarga de mantener una lista de figuras que contienen los gráficos activos. Se puede organizar en figuras, subplots y líneas. Dispone de diferentes maneras de visualizar los datos, línea de tiempo, diagrama de dispersión y para el nodo implementado, el modo *quiver* que muestra el Optical Flow. Se puede extender la visualización mediante *plug-ins* para el formato deseado. Es capaz de guardar en formato PNG una imagen de las gráficas.

4.1.2. Nodo de ROS

Como complemento a la interfaz gráfica se ha pensado en el desarrollo de un nodo que proporcione el Optical Flow de las imágenes obtenidas con la cámara del robot, dado que ROS no implementa en ninguno de sus nodos esta funcionalidad.

Para el cálculo del Optical Flow se ha utilizado la implementación que proporciona OpenCV. El algoritmo implementado es el de Lucas-Kanade [30]. Esta implementación, dadas dos imágenes consecutivas y un vector de posiciones dentro de la primera imagen, calcula las nuevas posiciones en la segunda imagen. A partir de este vector de nuevas posiciones se calcula el desplazamiento de los *pixels* en el tiempo, además de su equivalencia en coordenadas polares.

Los puntos en los que se calcula el Optical Flow estan distribuidos de manera uniforme en una rejilla de NxM puntos. Toda la información va encapsulada en un mensaje de ROS (Figura 4.1) que se publica con el *timestamp* de la imagen sobre la que se ha calculado. Como era necesario modificar varios parámetros, tanto del nodo en si como de la implementación del Optical Flow, se ha decidido usar un *package* que ROS dispone para tal efecto. Este paquete, *dynamic reconfigure*, que facilita la reconfiguración dinámica de nodos a través de una GUI. El nodo proporciona una estructura de datos a configurar a la GUI y se presentan esos parámetros al usuario. Después de la modificación de esos parámetros, *dynamic_reconfigure* le envía la nueva configuración al nodo.

```
1  Header header
2  int16 width
3  int16 height
4  float32 minx
5  float32 miny
6  float32 maxx
7  float32 maxy
8  int8[] status
9  float32[] flowx
10 float32[] flowy
11 float32[] mag
12 float32[] theta
13 float32[] errors
```

Figura 4.1: Mensaje de Optical Flow

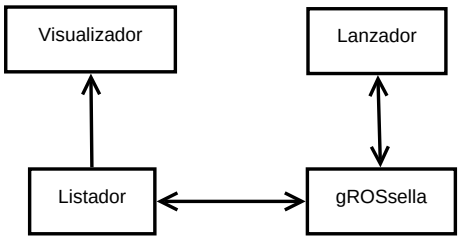

Figura 4.2: Arquitectura del sistema

Como el resultado del cálculo del Optical Flow tiene demasiado ruido se añadió un filtro pasa bajos [31] para suavizarlo. Esos parámetros se pueden modificar a tiempo real mediante el *dynamic_reconfigure* explicado anteriormente.

4.2. Implementación

4.2.1. Arquitectura del sistema

En la Figura 4.2 se muestra la arquitectura del sistema implementado. La ventaja de esta arquitectura básicamente es el desacoplamiento de los módulos, que

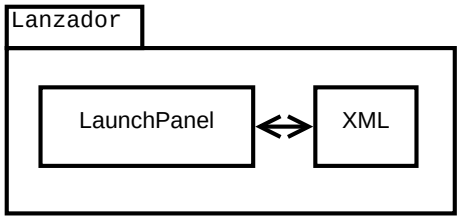

Figura 4.3: Submódulos del Lanzador

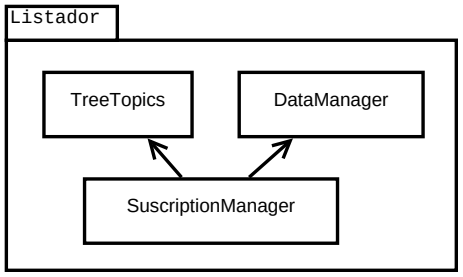

Figura 4.4: Submódulos del Listador

hace mas sencilla su implementación y estructuración de las clases. Podemos observar que gROSella, el eje principal de la aplicación, se comunica con el módulo Lanzador y el Listador. Por otro lado el módulo Listador se encarga de comunicarse con el Visualizador, de esta manera, la parte de visualización de los datos es completamente independiente del sistema, y podría ser sustituido por una implementación diferente.

4.2.2. Módulos

La división en los módulos ya descritos la podemos dividir en submódulos para repartir roles. Esta división se describe a continuación.

Lanzador

- LaunchPanel : Tiene que gestionar los nodos y parámetros del lanzador y vincularlos a la GUI.
- XML : Se encarga de exportar e importar el formato de *roslaunch* (XML).

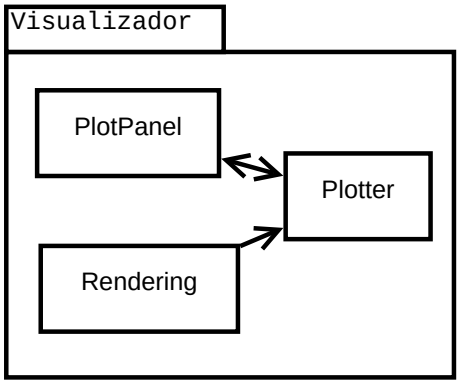

Figura 4.5: Submódulos del Visualizador

Listador

- TreeTopics : Gestiona la creación de la lista de *topics* en la GUI.
- DataManager : Gestiona las suscripciones a datos para enviárselos al módulo Visualizador.
- SubscriptionManager : Proporciona la funcionalidad básica para la gestión de suscripciones.

Visualizador

- PlotPanel: Gestiona las figuras activas.
- Plotter: Contiene las ventanas donde se visualizan los datos.
- Rendering: Son los diferentes *plug-ins* de visualización.

4.2.3. Clases

El reparto anterior de roles finalmente se divide en las clases descritas en la siguiente sección.

- grosella: Clase principal que dibuja la ventana con su menú, botones y los módulos. También gestiona el paso de información entre módulos,

Lanzador

- NodeLauncherPanel: panel que se encarga de cargar lanzadores en *NodeLauncher* y guardar los lanzadores en formato *launch* de ROS. También contiene los parámetros globales y las máquinas donde se lanzarán los nodos.
- NodeLauncher: contiene la información de un nodo y sus parámetros.
- Dict2XML: convierte un diccionario a formato *launch* en XML.

Listador

- Topics: panel que se encarga de toda la gestión del modulo listador.
- TopicTreeCtrl: se encarga de crear una estructura en árbol de los *topics* activos.
- TopicTreeItemData: derivada de TreeItemData de wxPython que la extiende para poder contener un DataConnection.
- DataConnection: contiene la información de una conexión.
- TopicArrayDataConnection: contiene la información de una conexión de la creación del árbol.
- Subscription: encapsula la información de una suscripción.
- TopicTreeSubscription: derivada de Subscription para encapsular la información de la creación del árbol de *topics*.
- DataSubscription: derivada de Subscription para encapsular los datos necesarios para poder dibujar.
- SubscriptionManager: gestiona las suscripciones.
- TopicTreeSubscriptionManager: derivada de SubscriptionManager que gestiona las suscripciones de la creación del árbol de *topics*.

- DataSubscriptionManager: derivada de SubscriptionManager que gestiona las suscripciones de los datos necesarios para poder dibujar.

Visualizador

- Plotter: encargada de hacer los *plots* en función de su tipo.
- PlotTreeCtrl: encargada de gestionar los *plots*, *subplots* y líneas activas.
- PlotTreeItemData: derivada de TreeItemData de wxPython que la extiende para poder contener la información necesaria de los plots.
- SubplotTreeItemData: derivada de TreeItemData de wxPython que la extiende para poder contener la información necesaria de los subplots.
- LineTreeItemData: derivada de TreeItemData de wxPython que la extiende para poder contener la información necesaria de las lineas.
- Buffer: contiene los datos a dibujar.
- SubplotRenderer: encapsula la información necesaria para dibujar.
- ScatterRenderer: derivada de SubplotRenderer que la extiende conteniendo la información para un dibujo de puntos.
- QuiverRenderer: derivada de SubplotRenderer que la extiende conteniendo la información para un dibujo del Optical Flow.
- SeriesRenderer: derivada de SubplotRenderer que la extiende conteniendo la información para un dibujo temporal.

4.2.4. Ventanas del programa

En la Figura 4.6 se muestra la ventana principal del programa al completo. La Figura 4.7 muestra las funcionalidades básicas del programa, que están dispuestas a modo de botones. Por orden de aparición:

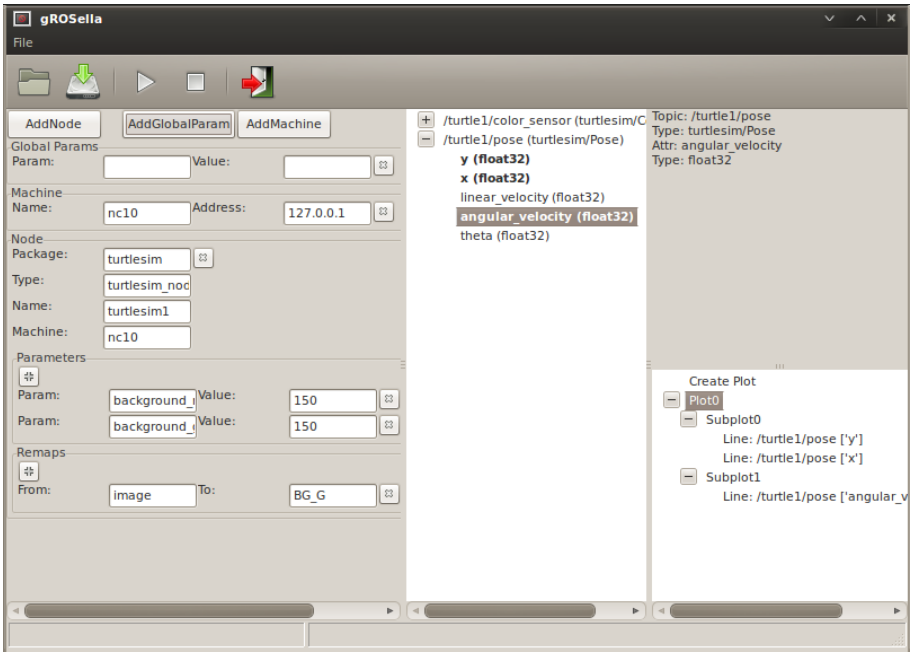

Figura 4.6: gROSella

Figura 4.7: Botones básicos

Cargar : Abre un diálogo para elegir un *launcher* definido por el usuario o del sistema y lo carga en el panel del módulo lanzador.

Guardar : Exporta a formato *launcher* toda la información contenida en el panel del módulo lanzador.

Reproducir : Lanza los nodos con sus respectivos parámetros y mapeos contenidos en panel del módulo lanzador.

Parar : Para los nodos y sus parámetros que se han lanzado.

Salir : Cierra el programa y las figuras si es que hay alguna abierta.

Figura 4.8: Botones del Lanzador

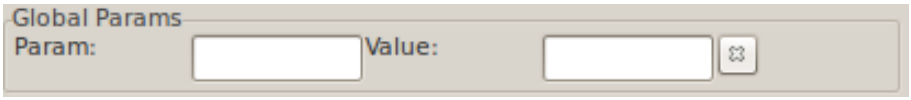

Figura 4.9: Parámetros Globales del Lanzador

La división en módulos del programa también es aplicable a la representación de la GUI. Cada módulo tiene su *frontend* y se describen a continuación.

Lanzador : Como se muestra en la Figura 4.8, consta de tres botones. AddNode que añade un nodo al panel con sus respectivos parámetros y mapeos (Figura 4.11). AddGlobalParam que añade parámetros globales al panel (Figura 4.9). AddMachine que añade la máquina en la que se quiere ejecutar el *launcher* (Figura 4.10), ya que puede ser ejecutada en una máquina diferente a donde se está ejecutando gROSella. Todo lo que se ha añadido se puede eliminar usando el botón "X".

Listador : Consta de tres partes. Árbol donde se muestran los *topics* y sus parámetros que están activos para su posterior graficado (Figura 4.12). Panel de información sobre el parámetro seleccionado (Figura 4.13). Aunque es parte del módulo Visualizador, se ha decidido colocar el árbol de las figuras activas, en esta parte de la GUI (Figura 4.14).

Visualizador : Ventana donde se grafican los parámetros de los *topics* seleccionados. Consta de unos botones de herramientas para la navegación por el gráfico y para poder guardarlo en una imagen (Figura 4.15). En las Figuras

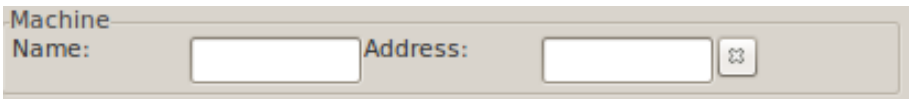

Figura 4.10: Máquinas del Lanzador

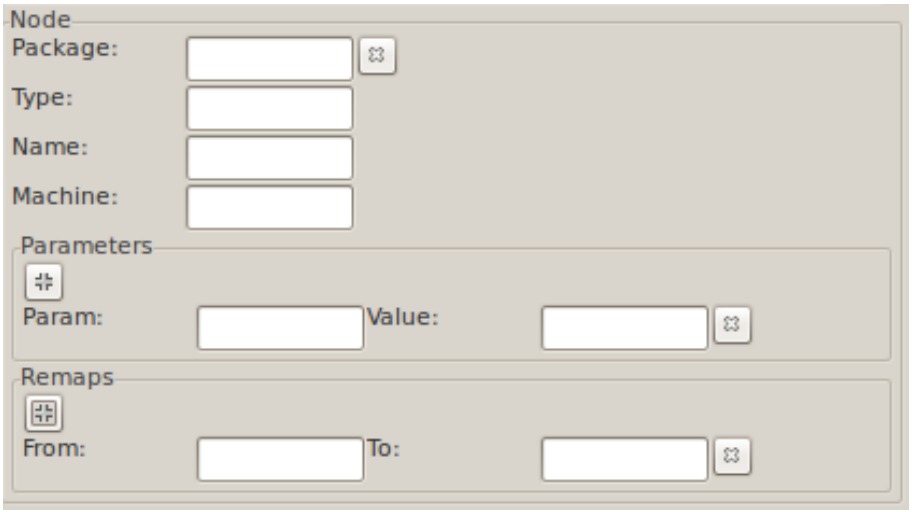

Figura 4.11: Nodos del Lanzador

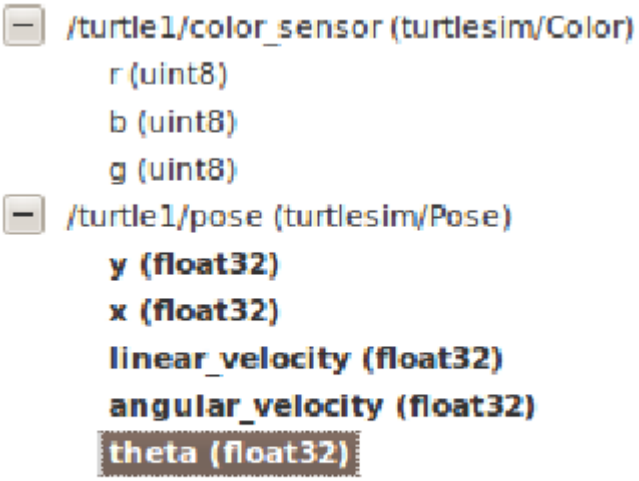

Figura 4.12: Árbol del Listador

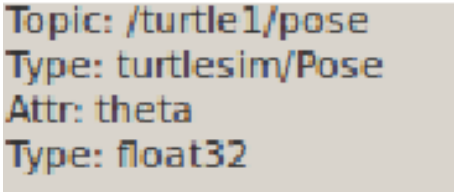

Figura 4.13: Información del Listador

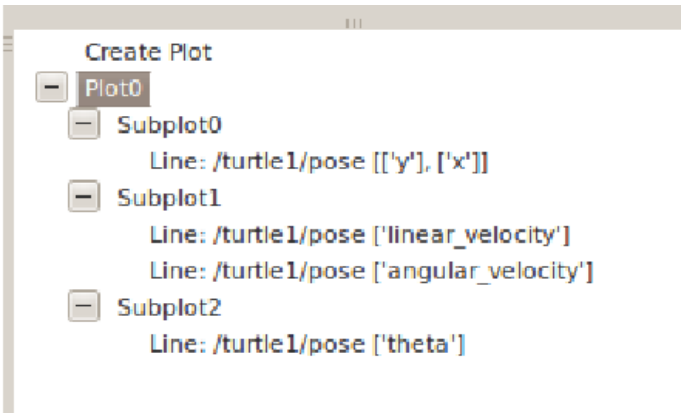

Figura 4.14: Árbol de figuras

Figura 4.15: Botones del Visualizador

4.16, 4.17, 4.18 y 4.19 se muestran los distintos tipos de graficado que se han desarrollado.

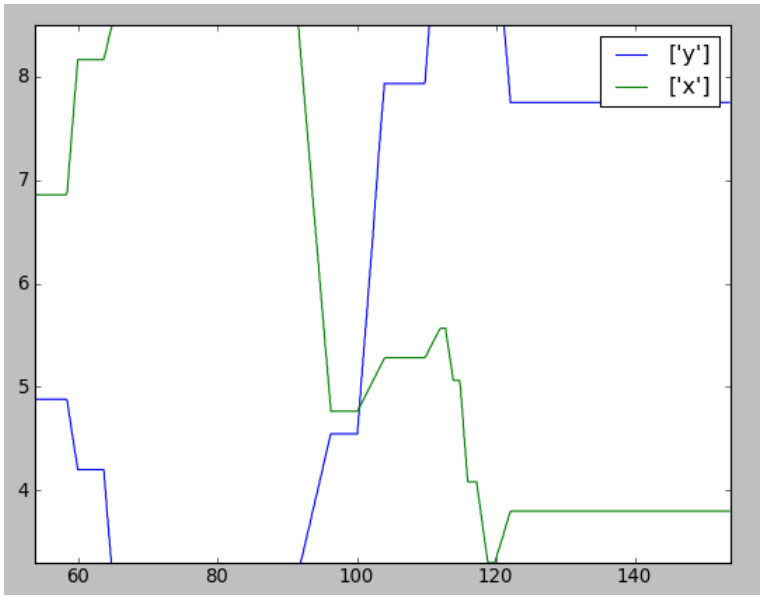

Figura 4.16: Series Plot

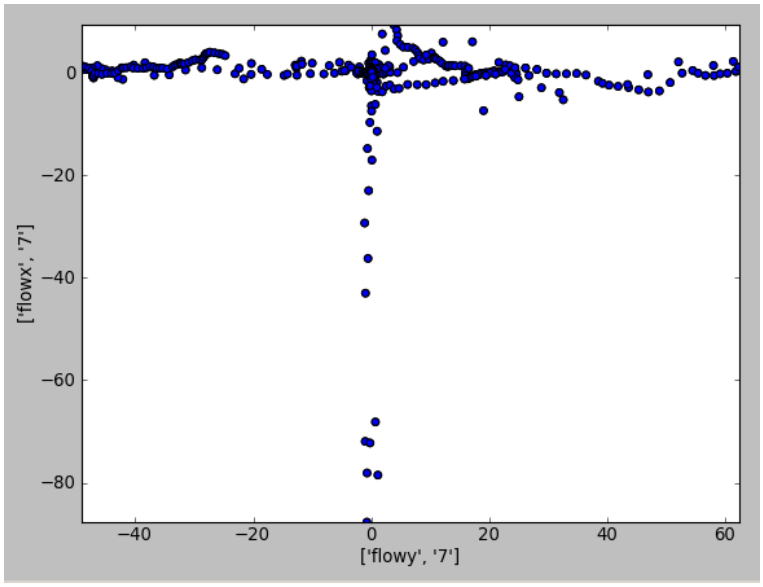

Figura 4.17: Scatter Plot

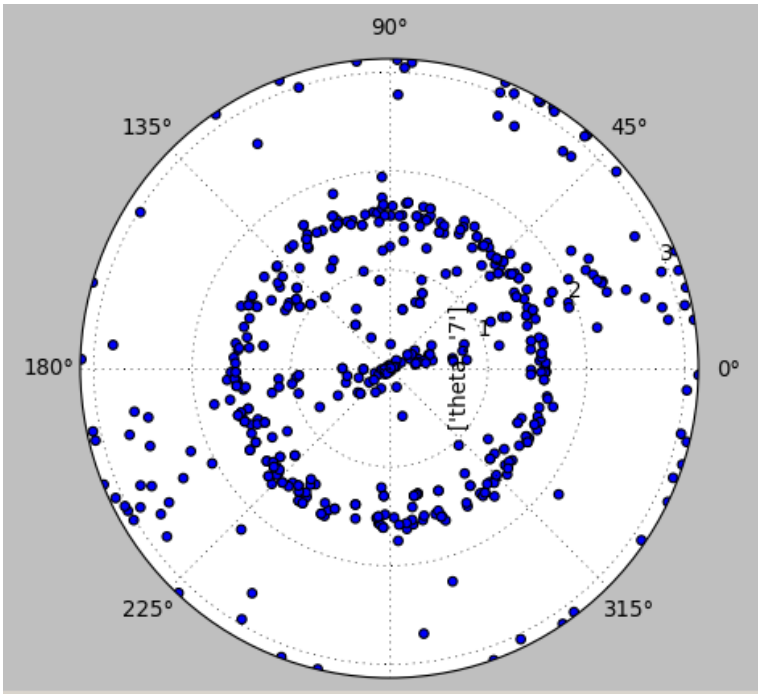

Figura 4.18: Scatter Polar Plot

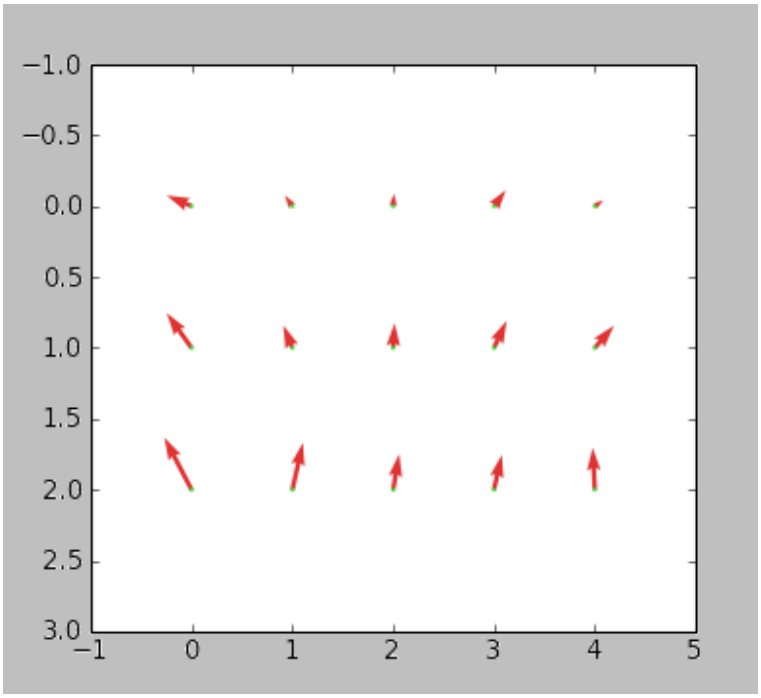

Figura 4.19: Quiver Plot

Capítulo 5

Conclusiones

5.1. Conclusiones

La finalidad de la aplicación era crear una herramienta que facilitase el proceso de experimentación con ROS, además de una ayuda gráfica de los datos que fluyen por el sistema. Finalmente el proyecto ha sido realizado y finalizado cumpliendo con los requisitos iniciales.

El proceso de diseño ha sufrido cambios durante el desarrollo, aunque los que ha habido han sido para dividir mas los roles del programa. A la planificación también se le han aplicado cambios sobre el orden de desarrollo de los módulos junto con el nodo implementado y la redacción de la memoria.

El desarrollo ha sido más difícil de lo esperado, dado que la API de ROS difiere bastante del uso por consola de sus herramientas. Esa mínima relación directa ha repercutido en el desarrollo, ya que se diseñó sobre el funcionamiento de ROS y al final hubo que hacer muchos cambios para poder adaptarlo a la API.

El hecho que hubiera tanta información sobre wxPython ha hecho que el desarrollo de la parte gráfica fuera más sencillo de lo esperado. De la misma manera, matplotlib tenía muchos ejemplos para la realización de gráficas de todo tipo, que han sido de gran ayuda.

5.2. Trabajo futuro

Como ampliación al trabajo hecho, se podrían añadir los siguientes requisitos:

Roscore : Para hacer posible el funcionamiento de ROS hay que lanzar el comando *roscore* que hace de centro de comunicaciones. Se podría lanzar automáticamente al iniciar la aplicación o añadir un botón para ello.

Bags : Como se ha explicado en la introducción, todos los datos se pueden guardar en *bags*. Sería interesante poder cargar y guardar *bags* desde el programa.

Colores : Al añadir una línea a una gráfica, adquiere un color nuevo por defecto. Se podría añadir la posibilidad de elegir colores de las líneas para conseguir unas gráficas más acordes con las necesitadas por el usuario.

Plug-ins : La aplicación se ha diseñado para una posible extensión de los tipos de visualización. De esta manera se podrían añadir visualizaciones de otros tipos de datos que no contempla.

Listas dinámicas : En vez de tener que escribir el nombre del nodo y sus parámetros, sería más sencillo mediante un desplegable elegir el nodo y dinámicamente obtener el desplegable de los parámetros.

Añadir launchers : La aplicación carga un *launchers* y se le pueden añadir/modificar nodos/parámetros/mapeos. Sería interesante añadirle una funcionalidad para poder cargar un *launcher* y añadirle otro sin eliminar la información del primero.

Bibliografía

[1] The Player Project.
<http://playerstage.sourceforge.net/>

[2] Yet Another Robot Platform.
<http://eris.liralab.it/yarp/>

[3] The Orocos Project.
<http://www.orocos.org/>

[4] CARMEN, Carnegie Mellon Robot Navigation Toolkit.
<http://carmen.sourceforge.net/>

[5] Orca.
<http://orca-robotics.sourceforge.net/>

[6] The MOOS Cross Platform Software for Robotics Research.
<http://www.robots.ox.ac.uk/ mobile/MOOS/wiki/pmwiki.php>

[7] Microsoft Robotics Developer Studio.
<http://msdn.microsoft.com/en-us/robotics/default.aspx>

[8] ROS (Robot Operating System).
<http://www.ros.org/wiki/>

[9] Instalación de ROS.
<http://www.ros.org/wiki/ROS/Installation>

[10] ROS API.
<http://www.ros.org/doc/api/>

[11] Python Programming Language.
<http://www.python.org/>

[12] Python Information and Examples.
<http://www.secnetix.de/olli/Python/>

[13] IPython: Productive Interactive Computing.
<http://ipython.org/>

[14] wxWidgets.
<http://www.wxwidgets.org/>

[15] wxPython.
<http://www.wxpython.org/>

[16] Instalación de wxPython.
<http://wiki.wxpython.org/InstallingOnUbuntuOrDebian>

[17] API wxPython.
<http://www.wxpython.org/docs/api/>

[18] wiki wxPython.
<http://wiki.wxpython.org/>

[19] The wxPython tutoriales.
<http://zetcode.com/wxpython/>

[20] wxPython Documentation.
<http://www.wxpython.org/onlinedocs.php>

[21] LaTeX Project.
<http://www.latex-project.org/>

[22] TexMaker.
<http://www.xm1math.net/texmaker/>

[23] Edición Avanzada de Textos Científicos con LaTeX y Gestión Bibliográfica.
<http://hallsi.ugr.es/cursoLatex/>

[24] matplotlib: python plotting.
<http://matplotlib.sourceforge.net/>

[25] matplotlib Installing.
<http://matplotlib.sourceforge.net/users/installing.html>

[26] matplotlib API.
<http://matplotlib.sourceforge.net/api/index.html>

[27] OpenCV (Open Source Computer Vision).
<http://opencv.willowgarage.com/wiki/>

[28] OpenCV InstallGuide.
<http://opencv.willowgarage.com/wiki/>

[29] OpenCV API.
<http://cgi.cs.indiana.edu/ oleykin/website/OpenCVHelp/>

[30] Lucas–Kanade method.
<http://en.wikipedia.org/wiki/Lucas-Kanade_method>

[31] Low-pass filter.
<http://en.wikipedia.org/wiki/Low-pass_filter>

[32] Optical flow.
<http://en.wikipedia.org/wiki/Optical_flow>

Firmado:

Bellaterra, Septiembre de 2011

Resumen

La programación de robots móviles autónomos hace necesario el uso de librerías para la comunicación y procesamiento de datos. Una de estas librerías es ROS (Robot Operating System), que es el sistema sobre el que se ha desarrollado este proyecto. Para simplificar el diseño de experimentos en este sistema, se ha propuesto desarrollar una herramienta gráfica que facilita la experimentación con robots. Esta herramienta consta de tres partes: diseño y ejecución de procesos, representación de los procesos activos y visualización de los datos. También se ha implementado un proceso para el sistema ROS que calcula el Optical Flow.

Resum

La programació de robots mòbils autònoms fa necessari l'us de llibreries per a la comunicació i processament de dades. Una d'aquestes llibreries és ROS (Robot Operating System), que és el sistema sobre el qual s'ha desenvolupat aquest projecte. Per simplificar el disseny d'experiments per a aquest sistema s'ha proposat desenvolupar una eina gràfica que facilita l'experimentació amb robots. Aquesta eina consta de tres parts: disseny i execució de processos, representació dels processos actius i visualització de les dades. També s'ha implementat un procés pel sistema ROS que calcula l'Optical Flow.

Abstract

Programming autonomous mobile robots requires the use of libraries for communication and data processing. This project has been developed on top of ROS (Robot Operating System), which is a set of such libraries. This project proposes a graphical tool in order to ease experiment design for this system. This tool is divided in three parts: process design and execution, active processes representation and data visualization. In addition a ROS process for Optical Flow calculation has been implemented.