UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIEROS INDUSTRIALES VALENCIA

TRABAJO FIN DE MASTER EN INGENIERÍA INDUSTRIAL

DESARROLLO DE INTERFACES DE COMUNICACIÓN PARA UN ROBOT DE REHABILITACIÓN DE MIEMBROS INFERIORES

AUTOR: ALEMANY IBOR, SERGIO

TUTOR: VALERA FERNANDEZ, ANGEL

Curso Académico: 2017-18

RESUMEN

La propuesta de Trabajo Fin de Máster (TFM) está enmarcada en el ámbito del proyecto de investigación del Plan Nacional del Gobierno de España 'Metodología de Diseño de Sistemas Biomecatrónicos'. Aplicación al desarrollo de un Robot Paralelo híbrido para diagnóstico y rehabilitación (referencia DPI2013-44227-R).

Se ha desarrollado un sistema de creación de interfaces funcionales fácilmente ampliable que permitan a los usuarios realizar ejercicios con el robot de rehabilitación mencionado sin tener conocimientos profundos sobre el mismo.

Todo esto ha sido desarrollado para el framework ROS (Robot Operating System), montado sobre Ubuntu Linux, programado en C++ utilizando las librerías del framework Qt para las interfaces gráficas y usando el programa informático QtCreator.

El programa ha sido desarrollado desde cero, desde su diseño y análisis hasta su implementación, pasando por la integración del framework Qt en ROS.

Palabras clave: ROS, robot, operating, system, rehabilitación, Qt, QtCreator, framework, interfaz

RESUM

La proposta del Treball de Fi de Màster (TFM) està emmarcada en l'àmbit del projecte d'investigació del Pla Nacional del Govern d'Espanya 'Metodologia de Disseny de Sistemes Biomecatrònics'. Aplicació al desenvolupament d'un Robot Paral·lel híbrid per al diagnòstic i rehabilitació (referència DPI2013-44227-R).

S'ha desenvolupat un sistema de creació d'interfícies funcionals fàcilment ampliable que permeten als usuaris realitzar exercicis amb el robot de rehabilitació mencionat sense tindre coneixements profunds sobre el mateix.

Tot açò ha sigut desenvolupat per al framework ROS (Robot Operating System), muntat sobre Ubuntu Linux, programat en C++ utilitzant les llibreries del framework Qt per a les interfícies gràfiques i utilitzant el programa informàtic QtCreator.

El programa ha sigut desenvolupat des de zero, des del seu disseny i anàlisi fins a la seua implementació, passant per la integració del framework Qt en ROS.

Palabras clave: ROS, robot, operating, system, rehabilitació, Qt, QtCreator, framework, interficies

ABSTRACT

This End of Master Project (TFM) proporsal is framed within the scope of the research project of the National Plan of the Government of Spain 'Methodology of Biomechatronic System Design'. Application to the development of a Hybrid Parallel Robot for diagnosis and rehabilitation (reference DPI2013-44227-R).

A system has been developed to create easily expandable functional interfaces that allow users to perform exercises with the mentioned rehabilitation robot without having in-depth knowledge about it.

All of this has been developed for the framework ROS (Robot Operating System), mounted on Ubuntu Linux, programmed in C++ using the libraries of the Qt framework for the graphical interfaces and using the computer program QtCreator.

The program has been developed from scratch, from its design and analysis to its implementation, including the integration of the Qt framework in ROS.

Palabras clave: ROS, robot, operating, system, rehabilitation, Qt, QtCreator, framework, interface

DOCUMENTOS CONTENIDOS EN EL TFM

- Memoria
- Presupuesto

ÍNDICE DE LA MEMORIA

ÍNDICE DEL PRESUPUESTO

MEMORIA

1. ESTRUCTURA DEL DOCUMENTO

El documento introduce progresivamente las nociones necesarias para la comprensión del trabajo realizado.

En el apartado 2 se introducen los objetivos del trabajo y la motivación para su realización. El apartado 3 describe el ámbito de aplicación que cubrirá el trabajo. El apartado 4 analiza brevemente la existencia de aplicaciones similares.

El apartado 5, 6 y 7 introducen los conceptos necesarios para comprender el trabajo realizado, empezando por el robot paralelo sobre el que se ha basado el trabajo, siguiendo con ROS (Robot Operating System) y finalizando con el framework utilizado, Qt.

En el apartado 8 se describe todo el desarrollo realizado para llevar a cabo el trabajo., incluyendo la instalación de los componentes necesarios, la integración del proyecto, el diseño de las pantallas y el desarrollo del sistema de ficheros, entre otros. El apartado 9 enumera los trabajos que se podrían aplicar para mejorar la aplicación creada.

Finalmente, se añaden 2 anexos. El primero exponiendo el contenido del fichero utilizado para la integración, mientras que el segundo es una guía sobre como añadir nuevos componentes a la aplicación.

2. MOTIVACIÓN Y OBJETIVOS DEL TRABAJO

La propuesta de Trabajo Fin de Máster (TFM) está enmarcada en el ámbito del proyecto de investigación del Plan Nacional del Gobierno de España Metodología de Diseño de Sistemas Biomecatrónicos. Aplicación al desarrollo de un Robot Paralelo híbrido para diagnóstico y rehabilitación (referencia DPI2013-44227-R).

Se trata de un proyecto de investigación interdisciplinar en el que participan investigadores del Dpto. de Ingeniería de Sistemas y Automática, del Dpto. de Ingeniería Mecánica y de Materiales y del Instituto de Biomecánica de Valencia, todos ellos adscritos a la Universitat Politècnica de València.

El proyecto tiene por objetivo el desarrollo de una nueva metodología de diseño de sistemas bio-mecatrónicos destinados a monitorizar y controlar el movimiento de segmentos corporales. Dicha metodología permitirá analizar y controlar de forma conjunta el sistema máquina-cuerpo humano para el desarrollo de aplicaciones en ámbitos como la valoración

funcional, las técnicas de apoyo al diagnóstico de patologías del aparato locomotor y los sistemas de rehabilitación motora [1].

En el proyecto de investigación se ha desarrollado un robot de rehabilitación de miembro inferior. Se trata de un robot paralelo de 3 grados de libertad: un movimiento lineal (coordenada Z) y dos movimientos angulares (roll y pitch) con el que se están desarrollando controladores de posición, fuerza y controladores híbridos fuerza-posición.

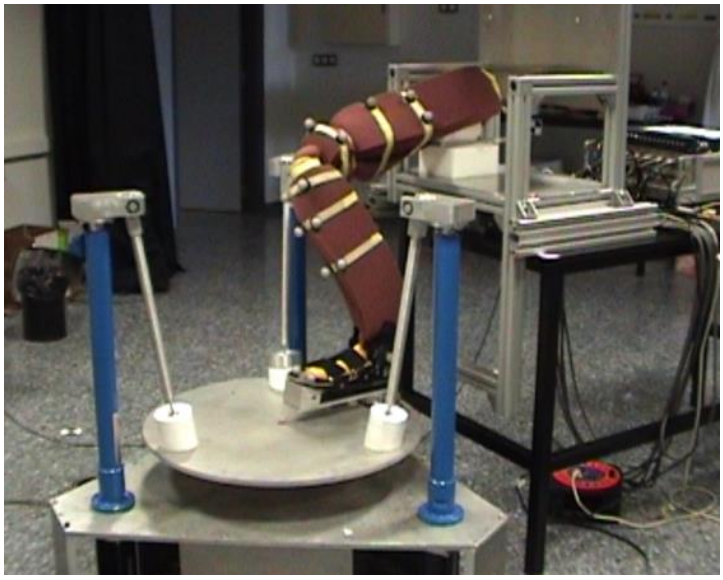


Figura 1: Robot de rehabilitación de miembro inferior. Fuente: https://mebiomec.ai2.upv.es/

Hasta ahora, dicho robot carece cualquier tipo de interfaz que permita comunicarse directamente con el usuario, por lo que la interacción hombre-máquina resulta sumamente complicada.

Si el objetivo final del robot de rehabilitación es el de poder usarse en aplicaciones reales, la creación de interfaces que interactúen con el usuario pasa a ser un requisito indispensable.

El presente TFM propone por tanto, trabajar con el desarrollo de interfaces de comunicación para el robot paralelo. Gracias a estos interfaces se podrá mejorar la comunicación médico-robot y paciente-robot.

En concreto, uno de los principales usos para el robot de rehabilitación en los que es imprescindible el uso de interfaces es en la de la realización de ejercicios por parte de los pacientes, los cuales han de seguir una serie de referencias de posición o fuerza sobre el mismo robot.

Sin las interfaces, el paciente no tiene método de saber en tiempo real, si están siguiendo correctamente la referencia ni si deben o no realizar correcciones.

Por tanto, son necesarias interfaces que proporcionen al usuario la información inmediata del estado del robot, y además, cuáles deben ser las referencias en un momento concreto.

Además, es necesario tener en cuenta que no todos los usuarios se adaptan igual a diferentes métodos de representación. A un paciente le puede resultar más cómodo poder observar una gráfica con los valores de referencia en el tiempo en el eje Y, y su evolución en el tiempo con el eje X, mientras que otro paciente se sienta más cómodo observando solamente el valor de la referencia en ese mismo momento.

No basta, por tanto, desarrollar una sola interfaz para todos los pacientes, sino que es necesario contar con un sistema en el cual los diferentes elementos de la interfaz sean modificables.

Por último, hay que tener en cuenta que en cada ejercicio no se pretenden controlar todas las señales del robot de rehabilitación a la vez, puesto que resultaría en un ejercicio demasiado complicado para el paciente. Lo más usual es controlar una sola señal, como puede ser la altura en el eje z de la plataforma del robot, aunque también pueden haber ejercicios que controlen más de una señal, como el control de la inclinación de la plataforma en ambos ejes (roll y pitch), en cuyo caso sería necesario controlar 2 señales.

El objetivo final, por tanto, es el desarrollo de un sistema de creación de interfaces funcionales fácilmente ampliable que permitan a los usuarios realizar ejercicios con el robot de rehabilitación sin tener conocimientos profundos sobre él.

3. ÁMBITO DE APLICACIÓN

La aplicación ha sido desarrollada para correr en Ubuntu Linux, dentro del framework ROS (Robot Operating System), al ser el mismo sistema que utiliza el robot de rehabilitación objeto del trabajo.

El sistema permite crear interfaces personalizadas teniendo siempre en mente que el sistema robótico destino es el propio robot de rehabilitación, aunque debido a la naturaleza genérica de éste, puede ser usado en cualquier sistema robótico que necesite graficar una señal respecto a una referencia.

4. ESTADO DEL ARTE Y ANTECEDENTES

Si hablamos de interfaces en general, queda poco por innovar en este sentido. El desarrollo de interfaces de usuario es un tema que está mucho más que estudiado. Poco hay que innovar en este aspecto.

Sin embargo, si concretamos a las interfaces que funcionen en ROS, el número se reduce drásticamente. En la actualidad, son pocas las interfaces que se han desarrollado pensadas para que funcionen en ROS, debido a varios factores:

- La utilización de ROS principalmente para tareas industriales en los que el robot no interactúa directamente con el usuario, por lo que no hay necesidad de desarrollar interfaces de usuario.
- La ausencia de soporte para la creación de interfaces de usuario por parte de los lenguajes soportados por ROS.
- Derivado del punto anterior, se hace necesaria la utilización de frameworks para su desarrollo, los cuales no son todos compatibles con ROS.
- La existencia de RQT, el cual es un módulo para ROS que permite crear interfaces para sistemas robóticos.

Podríamos decir que RQT tiene una funcionalidad similar al objetivo de este trabajo, aunque tenga diferencias con el mismo. La principal diferencia radica en que RQT mantiene un aspecto y funcionalidad más complejo, ya que está orientado para un ámbito más industrial, donde los que manejan las interfaces son aquellos que han estado trabajando con el sistema robótico, y tienen mayores conocimientos sobre este. Esto no sería aplicable para uso real en hospitales, donde los médicos y pacientes no tienen una formación específica.

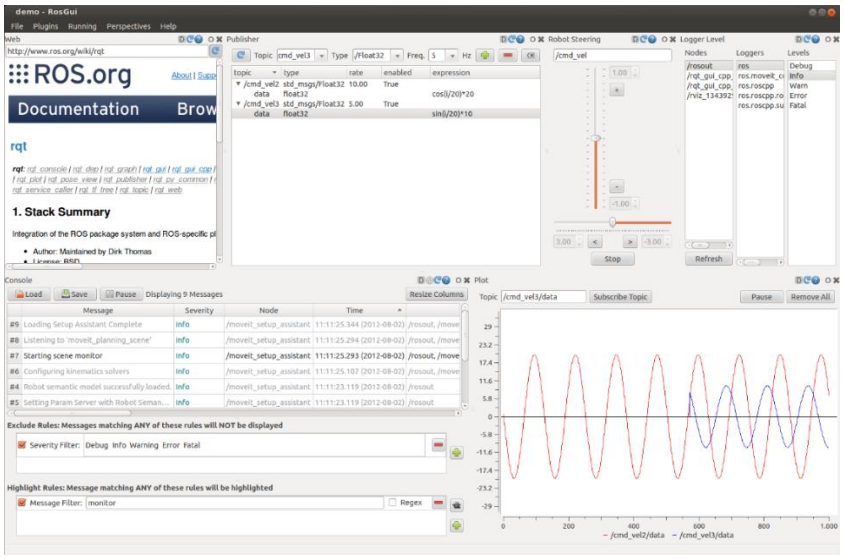


Figura 2: Vista general RQT

5. INTRODUCCIÓN A LA ROBÓTICA. ROBOT PARALELO

Resulta difícil poder definir formalmente en qué consiste la robótica industrial, ya que diferentes organismos robóticos y de estándares plantean diferentes definiciones, las cuales, aunque enormemente similares, poseen ligeras diferencias entre sí que pueden provocar que una máquina considerada como robot en una definición, no lo sea en otra.

Una de las definiciones más aceptadas es la proporcionada por el Robotics Institute of America (RIA), quien definió al robot industrial en 1979 como un "*manipulador multifuncional reprogramable, capaz de mover materias, piezas, herramientas, o dispositivos especiales, según trayectorias variables, programadas para realizar tareas diversa*s" [2].

Sin embargo, la Asociación Japonesa de Robótica Industrial (JIRA), propone una definición más genérica que la americana, lo define como "*dispositivos capaces de moverse de modo flexible análogo al que poseen los organismos vivos, con o sin funciones intelectuales, permitiendo operaciones en respuesta a las órdenes humanas"* [3].

Otra definición la encontramos en la International Standard Organization (ISO) que define al robot industrial como un *"Manipulador multifuncional, controlado automáticamente, reprogramable en tres o más ejes, que puede estar fijo o móvil para uso en aplicaciones de automatización industrial"* [4].

Esta definición es más restrictiva que la anterior, ya que delimita aquellas máquinas que tengan tres o más ejes. Esta misma definición es la adoptada por la International Federation of Robotics (IFR).

Debido a la gran cantidad de definiciones, y al hecho de continuamente se van desarrollando nuevas máquinas autómatas, que no están incluidos en ninguna clasificación anterior, no es posible catalogar cada robot en un tipo concreto sin excederse en la generalidad de la definición.

5.1. El Robot paralelo

Dado que el objeto del trabajo es el desarrollo de un sistema de interfaces para un robot paralelo, a continuación, se exponen los conceptos más importantes y generales de un robot de este tipo.

El término paralelo se refiere a que el robot posee varias cadenas cinemáticas, donde cada una de ellas trabaja independientemente. No se refiere, por tanto, al sentido geométrico de la palabra, es decir, no se aplica ningún tipo de restricción a las alineaciones, sino al sentido topológico, es decir, la capacidad de trabajar a la vez.

En la siguiente figura se muestra un ejemplo de un robot paralelo con cuatro miembros o cadenas cinemáticas independientes que trabajan en paralelo:

Figura 3: QUATTRO ADDEPT. Fuente: http://www.adept.com

Esta configuración forma cadenas cinemáticas cerradas, las cuales se diferencian de las cadenas cinemáticas abiertas en que cada eslabón está conectado a dos o más eslabones. Es decir, no tiene ningún elemento en voladizo, como se muestra en la figura X:

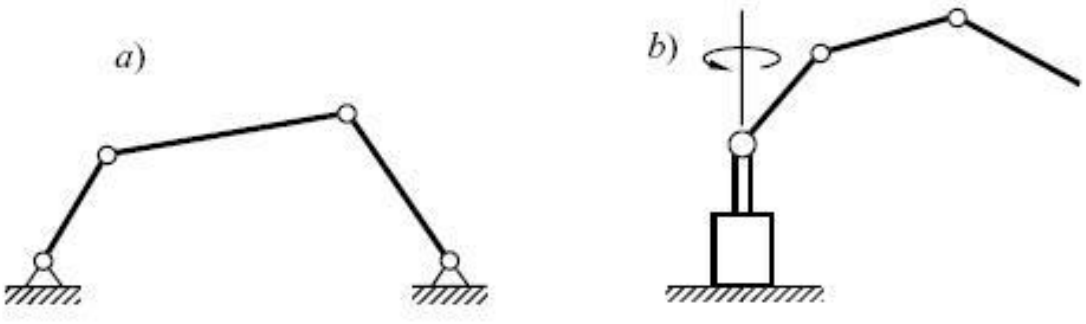


Figura 4: Cadena cinemática a) cerrada y b) abierta. Fuente: http://www.sitenordeste.com

Un robot paralelo, por tanto, es "*aquel cuya estructura mecánica está formado por un mecanismo de cadena cerrada en el que el efector final se une a la base por al menos dos cadenas cinemáticas independientes" [5].*

El robot paralelo se distingue de los robots coordinados (los cuales también forman cadenas cinemáticas cerradas), en que los primeros limitan cada cadena cinemática que está conectada a la base a un solo actuador, reduciendo así su complejidad.

El objetivo general de estos robots suele ser el control de la posición y/o orientación de la plataforma móvil situado al final de las cadenas cinemáticas para realizar las tareas asignadas.

Los robots paralelos tienen una serie de ventajas sobre los robots serie, que los hacen muy útiles para otras tareas:

- Las cargas situadas en la plataforma se reparten entre los miembros conectados a ella, por lo que la capacidad de manipular cargas pesadas es superior a la de los robots en serie.
- Cada miembro, al tener tan solo un actuador, permite que este se sitúe en la base del robot, lo que permite que las estructuras tengan un menor peso, sean más robustas, con mayor precisión, y alcancen mayores velocidades y aceleraciones.
- Debido a esto, los robots paralelos también suelen tener un coste de construcción menor al de los en serie, ya que su estructura suele ser más simple.
- La cinemática inversa es relativamente sencilla de calcular, lo que permite la simulación de modelos, para posteriormente aplicarlo al modelo real.

Sin embargo, los robots paralelos no son todo ventajas, sino que tienen una serie de características que podrían considerarse como desventajas respecto a los robots en serie:

- Suelen tener espacios de trabajo más reducidos debido a su configuración en cadenas cinemáticas cerradas.
- Tienen mayor facilidad de encontrarse con configuraciones singulares, las cuales deben resolverse para cada topología.
- Problemas a la hora de planificar su control debido a la inexistencia de un modelo dinámico general, lo que provoca que los robots paralelos se suelan controlar de forma desacoplada.

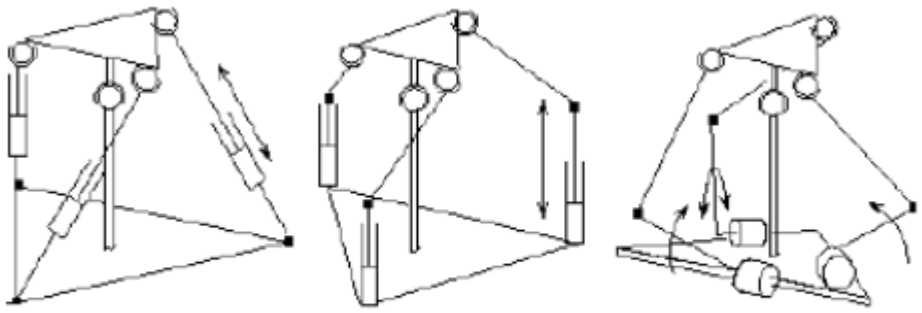


Figura 5: Ejemplos configuraciones robots.

El robot paralelo ha sido desarrollado de forma que, por su arquitectura y movimiento, con un mejor coste, es capaz de generar rotación angular en dos ejes (roll y pitch) y elevación como movimiento lineal.

Se consideraron dos arquitecturas alternativas: 3-RPS y 3-PRS. Tras comparar las ventajas e inconvenientes de cada una de las alternativas, finalmente, en este proyecto se optó por la arquitectura 3-PRS (prismáticarevolución-esférica), siendo una de las ventajas de la arquitectura PRS que los actuadores se localizan en la base fija, mientras que en la arquitectura 3-RPS, los actuadores se mueven junto con las articulaciones de revolución.

En la figura 6 se puede ver el robot paralelo utilizado. Se puede ver como tiene tres miembros que conectan la plataforma móvil con la base. Cada miembro consiste en:

1. Un motor, que mueve un husillo a bola.
2. Un deslizador.
3. Un vástago de conexión.

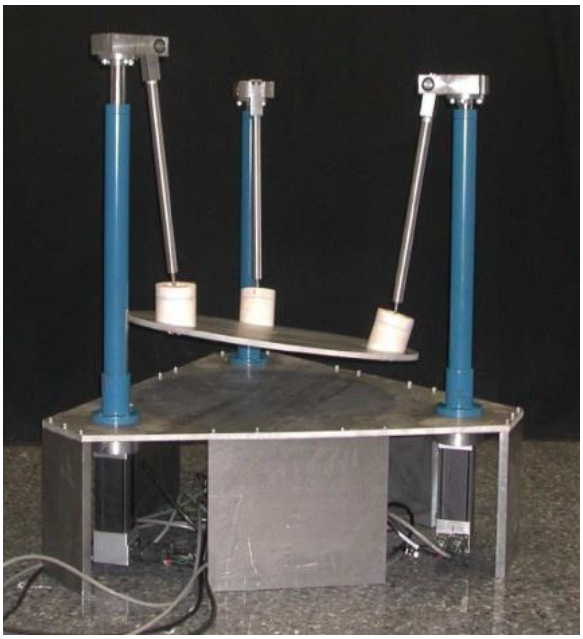


Figura 6: Robot Paralelo 3PRS utilizado

La plataforma móvil será el objeto de estudio, puesto que es donde el paciente situará su pie para la realización de los ejercicios de rehabilitación.

5.2. Estructura y componentes de un robot

A continuación, se describe la estructura general de un robot y la de sus componentes básicos. Más adelante veremos la utilidad de cada uno de ellos.

La configuración general de un robot se puede distribuir en tres unidades funcionales las cuales le conceden al usuario la posibilidad de actuar sobre el entorno del trabajo del robot:

- Unidad de programación: Está unidad se emplea para la programación y manipulación del robot. Mediante la unidad de programación, se pueden examinar y analizar los resultados de los comandos y de los programas que han sido ejecutados. La mayoría de veces consisten en una pantalla, un control remoto y un teclado industrial.
- Sistema de control: Este sistema es el que recibe las instrucciones y los comandos de la unidad de programación. Su tarea consiste en analizar las instrucciones y comandos y es producir las acciones de control oportunas para llevar a cabo dichas instrucciones. Estas acciones deben utilizar adecuadamente el sistema mecánico, que dependen de su estado, con el propósito de minimizar el error. Asimismo, esté sistema le notificara al usuario cuales son los resultados de las acciones de control mediante la unidad de programación. El sistema de control suele estar constituido por un equipo industrial, el cual está equipado con tarjetas de adquisición y conversión para poder comunicarse con el robot mecánico.
- Mecánica del robot. Es el encargado de transformar las acciones del sistema de control en movimientos del robot. También se encarga de informar periódicamente al sistema de control sobre el estado del robot. La mecánica del robot está formado por varios componentes individuales que están conectados a través de articulaciones, las cuales permiten el movimiento relativo de los componentes mediante actuadores, como pueden ser los motores eléctricos, hidráulicos o neumáticos. Por otro lado, los sensores, tales como los encóders, permiten conocer en cada instante la posición de una articulación.

En la figura 7 se puede observar el esquema general de un robot industrial y las diversas interacciones entre los distintos elementos.

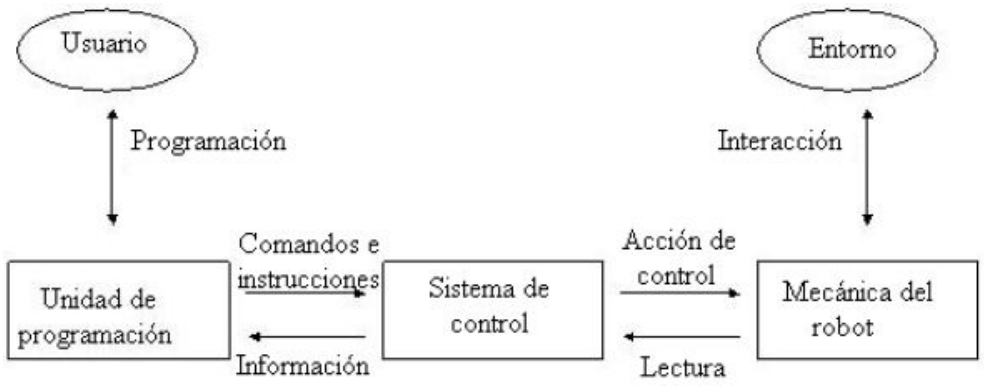


Figura 7: Esquema general de un robot

Tal como se ha mencionado antes, el sistema de control está constituido por un equipo industrial y diversos componentes añadidos:

- Convertidor D/A.
- Transformadores.
- Tarjetas de adquisición de datos A/D.
- Tarjetas de comunicación (serie, paralelo, Ethernet).

Donde sus tareas principales son:

- Capacidad para supervisar la interfaz de comunicación con el usuario, del mismo modo que debe ser capaz de interpretar los comandos enviados por el usuario y transformarlas en acciones de control. Los comandos proporcionados se interpretan y ejecutan de manera inmediata. Asimismo, el usuario, también puede insertar los programas en la unidad de programación, donde se podrán interpretar y ejecutar por el sistema de control.
- Producir y supervisar los movimientos del robot a través de señales de referencia que manejan los actuadores del robot. Durante todo el proceso, la referencia se coteja con los valores obtenidos a través de los sensores del sistema mecánico. El sistema de control debe procurar minimizar la diferencia entre la salida y la referencia a través de la ejecución de un algoritmo de control.

5.3. Cinemática en robots

La cinemática es una rama de la física que estudia el movimiento prescindiendo de las fuerzas que lo producen [6]. En ella, se estudia la posición, la velocidad, aceleración y todas las variantes de estas con respecto al tiempo o cualquier otra variable.

Por tanto, la cinemática de robots consiste en el estudio de la trayectoria que tiene el robot en función de las variables de configuración de las articulaciones del robot.

Los robots están formados por una serie de eslabones conectados mediante articulaciones que posibilitan el movimiento relativo entre eslabones vecinos. Estos eslabones pueden formar cadenas abiertas o cerradas, como se ha visto en la figura 4.

La cantidad de grados de libertad que un robot posee coincide con el número de variables de posición independientes que deberían ser especificadas para localizar todas las partes del mecanismo. En el caso de los robots industriales el número de grados de libertad suele equivaler al número de articulaciones siempre y cuando cada articulación tenga un solo grado de libertad.

Modelo cinemático directo

La cinemática de un robot es el estudio de los movimientos de un robot. El modelo cinemático directo es un sistema de ecuaciones matemáticas que permiten calcular la posición y orientación del elemento final del robot.

Dados una serie de valores específicos, denominados parámetros, para las articulaciones, el problema cinemático directo calcula la posición y orientación del marco de referencia del extremo final con respecto al marco de la base.

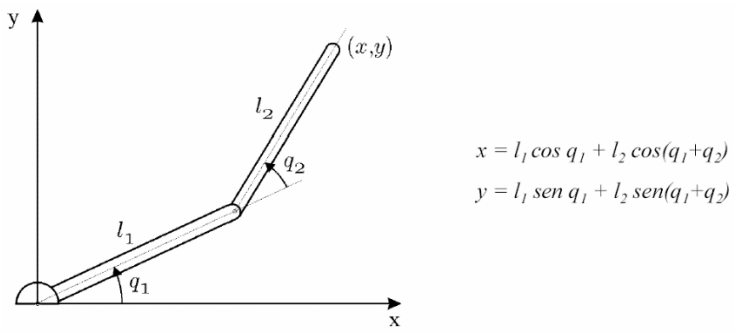


Figura 8: Modelo cinemático directo. Fuente: Apuntes de robótica

Modelo cinemático inverso

Dada la posición y orientación del elemento final del robot, el problema cinemático inverso consiste en calcular todos los posibles conjuntos de parámetros para las articulaciones del robot que podrían usarse para obtener la posición y orientación deseada.

El problema cinemático inverso es más complicado que la cinemática directa ya que las ecuaciones no son lineales, sus soluciones no son siempre fáciles o incluso posibles en una forma cerrada. También surge la existencia de una o de diversas soluciones. La existencia o no de la solución lo define el espacio de trabajo de un robot dado. La ausencia de una solución significa que el robot no puede alcanzar la posición y orientación deseada porque se encuentra fuera del espacio de trabajo del robot o fuera de los rangos permisibles de cada una de sus articulaciones.

A pesar de su complicación, conocer este modelo resulta de vital importancia para poder realizar la programación del robot, ya que en las instrucciones dadas al robot por el sistema de programación, no se proporcionan los parámetros de las articulaciones, sino que simplemente se le indica al robot a que posición debe dirigirse, por ejemplo, y es el sistema de control el que debe realizar la conversión a parámetros de articulaciones utilizando el modelo de cinemática inversa.

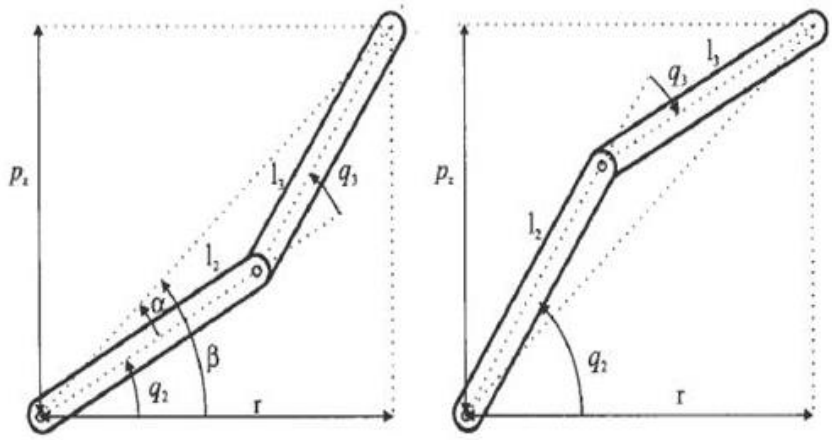


Figura 10: Configuraciones cinemática inversa

5.4. Control Dinámico de robots

La dinámica se ocupa de la relación entre las fuerzas que actúan sobre un cuerpo y el movimiento que en él se origina. De manera que el modelo dinámico de un robot posee como finalidad saber la relación que hay entre el movimiento del robot y las fuerzas involucradas en el mismo.

Esta relación se obtiene a través del modelo dinámico, el cual establece una serie de relaciones entre:

- La localización del robot definida por los parámetros de las articulaciones o por las coordenadas de localización de su extremo, así como su velocidad y aceleración.
- Las fuerzas de par aplicadas en las articulaciones (o en el extremo del robot).
- Los parámetros dimensionales del robot, como longitud, masa e inercias de sus elementos.

La obtención del modelo para sistemas con escasos grados de libertad (1 o 2) no es demasiado complicado, aunque esta tarea aumenta de dificultad a medida que ese número se incrementa.

El problema de la obtención del modelo dinámico de un robot es, por lo tanto, uno de los aspectos más complejos de la robótica, lo que ha llevado a ser obviado en numerosas ocasiones. Sin embargo, el modelo dinámico es imprescindible para determinados fines tales como:

- Simulación del movimiento del robot.
- Diseño y evaluación de la estructura mecánica del robot.
- Dimensionamiento de los actuadores.
- Diseño y evaluación del control dinámico del robot.

5.5. Modelos cinemáticos del robot 3PRS

El robot paralelo objeto de este trabajo puede verse como tres patas que conectan la plataforma móvil con la base fija. Cada pata consiste en un motor que mueve un actuador de husillo de bolas y un vástago de conexión.

5.5.1. Cinemática directa

La cinemática directa del manipulador paralelo consiste en, dados los movimientos lineales de los actuadores, encontrar los ángulos de balanceo (β) y alabeo (ϒ) y la elevación (z). Se puede utilizar la notación de Denavit-Hartenbert para establecer las coordenadas generalizadas del modelo cinemático. La figura 11 muestra los parámetros D-H del robot considerado.

i	1	2	3	4	5	6	7	8	9
d_i	q_1	0	0	0	0	q_6	0	q_8	0
α_i	0	0	l_a	0	0	0	0	0	0
θ_i	π/6	q_2	q_3	q_4	q_5	5π/2	q_7	-π/2	q_9
α_i	0	π/2	0	π/2	π/2	0	π/2	0	π/2

Figura 11: Parámetros D-H para el robot de 3PRS

A partir de la figura 11 se puede ver como a partir de 9 coordenadas generalizadas, se puede definir la cinemática del robot. La localización de los sistemas de coordenadas para modelar la cinemática se muestra en la figura 12.

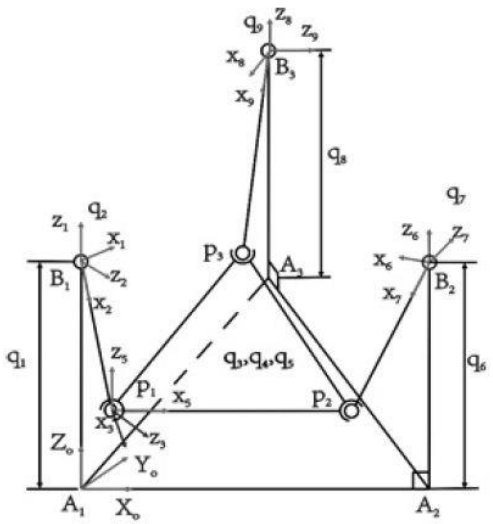


Figura 12: Localización sistemas de coordenadas

Las tres patas del robot paralelo se conectan a la plataforma móvil formando un triángulo equilátero. Por ello se puede ver que la longitud entre p_i y p_j es constante e igual a l_m. De este modo,

$$f_1(q_1,q_2,q_6,q_7) = \left\|\left(\vec{r}_{A_1B_1} + \vec{r}_{B_1p_1}\right) - \left(\vec{r}_{A_1A_2} + \vec{r}_{A_2B_2} + \vec{r}_{B_2p_2}\right)\right\| - l_{m=0} \quad (1)$$

$$f_2(q_1,q_2,q_8,q_9) = \left\|\left(\vec{r}_{A_1B_1} + \vec{r}_{B_1p_1}\right) - \left(\vec{r}_{A_1A_3} + \vec{r}_{A_3B_3} + \vec{r}_{B_3p_3}\right)\right\| - l_{m=0} \quad (2)$$

$$f_1(q_6,q_7,q_8,q_9) = \left\|\left(\vec{r}_{A_1A_3} + \vec{r}_{A_3B_3} + \vec{r}_{B_3p_3}\right) - \left(\vec{r}_{A_1A_2} + \vec{r}_{A_2B_2} + \vec{r}_{B_2p_2}\right)\right\| - l_{m=0} \quad (3)$$

En la cinemática directa, la posición de los actuadores es conocida, así, el sistema de ecuaciones (1) - (3) se puede ver como un sistema no-lineal con q_2, q_7 y q_9 como desconocidas. Para la resolución del sistema no lineal, el problema cinemático directo utiliza el método numérico de Newton-Rhapson ya que tiene una convergencia muy rápida (convergencia cuadrática) cuando la estimación inicial está cerca de la solución deseada. El método es iterativo y se puede escribir como:

$$\begin{bmatrix} q_2 \\ q_7 \\ q_9 \end{bmatrix}^{i+1} = \begin{bmatrix} q_2 \\ q_7 \\ q_9 \end{bmatrix}^{i} - J_i^{-1} \begin{bmatrix} f_1(q_2,q_7) \\ f_1(q_2,q_9) \\ f_1(q_7,q_9) \end{bmatrix}^{i} \quad (4)$$

En la ecuación anterior, *i* significa que las variables y funciones se evalúan en la iteración i. La matriz J es la matriz Jacobiana de f_i con respecto a las variables [q_2, q_7, q_9]. El proceso iterativo finaliza cuando:

$$\sqrt{(f_1(q_2,q_7)^i)^2 + (f_1(q_2,q_9)^i)^2 + (f_1(q_7,q_9)^i)^2} < \varepsilon \quad (5)$$

El parámetro ε es una cantidad positiva pequeña establecida en la programación del control.

El método de Newton-Rhapson requiere una aproximación inicial tan cercana como sea posible al valor solución. En este caso, esto no supone ningún problema ya que la posición inicial de la articulación que conecta la plataforma con el actuador es aproximadamente 2π/5.

La subsecuente aproximación inicial considera los valores de la posición previa del robot del robot.

La localización de la plataforma móvil se define usando el sistema de coordenadas unido a él. Una vez encontradas las coordenadas generalizadas para las patas del robot, se puede encontrar la posición de los puntos pi. Estos tres puntos comparten el plano de la plataforma. Basado en estos puntos se puede construir la matriz rotacional de la plataforma con respecto a la base. Se define un eje local X_p como un vector unitario $\vec{u}$ con la dirección dada por p_1, p_2.

El eje Z_p se define mediante un vector $\vec{u}$ y es un eje perpendicular al plano definido por los puntos p_1, p_2 y p_3.

Finalmente, el eje Y p se define mediante la dirección de los ejes $\vec{w}$ el cual está determinado mediante el producto vectorial entre los ejes $\vec{u}$ y $\vec{v}$. La matriz de rotación de la plataforma móvil viene dada por,

$$R_p = [\vec{u}^T \quad \vec{v}^T \quad \vec{w}^T] \tag{6}$$

A partir de la matriz de rotación se pueden encontrar el resto de coordenadas generalizadas q_3, q_4 y q_5.

5.5.2. Cinemática inversa

La cinemática inversa consiste en, dados los ángulos de balanceo y alabeo de la plataforma (β, ϒ) y la elevación (z), encontrar el movimiento lineal de los actuadores. Usando el sistema de ángulos fijo X-Y-Z; la matriz de rotación e puede definir como:

$$R_p = \begin{bmatrix} c_\alpha c_\beta & c_\alpha s_\beta s_\gamma - s_\alpha c_\gamma & c_\alpha s_\beta c_\gamma - s_\alpha c_\gamma \\ s_\alpha c_\beta & s_\alpha s_\beta s_\gamma - c_\alpha c_\gamma & s_\alpha s_\beta c_\gamma - c_\alpha s_\gamma \\ -s_\beta & c_\beta s_\gamma & c_\beta c_\gamma \end{bmatrix} \tag{7}$$

donde c* y s* hacen referencia al cos(*) y sen(*) respectivamente. Dados ϒ y β, el ángulo de cabeceo (α) se puede encontrar como sigue:

$$\alpha = arctan2(s_\beta s_y, (c_\gamma + c_\beta)) \tag{8}$$

Tras encontrar el ángulo , se pueden encontrar el resto de términos de la matriz de rotación. Las posiciones de los actuadores se pueden encontrar mediante las siguientes expresiones (9) - (11),

$$q_1 = p_x^2 + p_y^2 + p_z^2 + 2h(p_x u_x + p_y u_y + p_z u_z) - 2gp_x - 2ghu_x + g^2 + h^2 \quad (9)$$

$$q_6 = p_x^2 + p_y^2 + p_z^2 + h(p_x u_x + p_y u_y + p_z u_z) + \sqrt{3}h(p_x v_x + p_y v_y + p_z v_z) + g(p_x - \sqrt{3}p_y) + gh(u_x - \sqrt{3}u_y)/2 + gh(v_x - \sqrt{3}v_y)/2 + g^2 + h^2 \quad (10)$$

$$q_8 = p_x^2 + p_y^2 + p_z^2 + h(p_x u_x + p_y u_y + p_z u_z) + \sqrt{3}h(p_x v_x + p_y v_y + p_z v_z) + g(p_x - \sqrt{3}p_y) + gh(u_x - \sqrt{3}u_y)/2 + gh(v_x - \sqrt{3}v_y)/2 + g^2 + h^2 \quad (11)$$

donde: $h = \frac{l_m}{\sqrt{3}},\ g = \frac{l_b}{\sqrt{3}},\ p_x = -hu_y,\ p_y = -h(u_x - v_x),\ p_z = zy,$
l_b: $longitud\ entre\ A_i A_j$.

6. ROS (ROBOT OPERATING SYSTEM)

6.1. Definición de ROS

ROS (Robot Operating System) es un framework flexible para desarrollo de software para robots. Es una colección de herramientas, librerías y convenciones cuyo objetivo es simplificar la tarea de crear comportamientos complejos y robustos en una gran variedad de sistemas robóticos.

Esto, explican en la propia página de ROS, es debido a que es muy difícil conseguir que los robots hagan ciertos movimientos o respondan a ciertas situaciones que para nosotros los humanos son muy sencillas u obvias, y que, sin embargo, desde el punto de vista del robot puede resultar muy complicado, si, como programadores, hemos de enfrentarnos a la programación de los mismos de forma individual.

A pesar de que ROS no es un sistema operativo como tal, proporciona servicios estándar de un sistema operativo como abstracción de hardware, gestión de directorios, paso de mensajes entre procesos, mantenimiento de paquetes y control de dispositivos a bajo nivel.

Es un software libre disponible para sistemas operativos con base Unix. El sistema operativo oficialmente soportado es Ubuntu Linux, pero existen otras versiones de ROS para otros sistemas operativos como Fedora Linux y macOS, que son soportados por la comunidad. Existe también una versión experimental para Windows.

ROS está diseñado para ser lo más ligero posible, y permitir que el código utilizado en ROS pueda ser utilizado e integrado con otros softwares de control de robots.

También se busca la compatibilidad con el máximo número de lenguajes de programación. Funciona correctamente con Python, C++ y Lisp, y ahora mismo existen y están en desarrollo librerías experimentales con Java y Lua.

										
Box	**C**	**Diamond**	**Electric**	**Fuerte**	**Groovy**	**Hydro**	**Indigo**	**Jade**	**Kinetic**	**Lunar**
March 2010	August 2010	March 2011	August 2011	April 2012	December 2012	September 2013	July 2014	May 2015	May 2016	May 2017

Figura 13: Distribuciones ROS

Al igual que las distribuciones de Windows o Ubuntu, existen varias versiones de ROS, que han ido saliendo a lo largo del tiempo, incorporando nuevas funcionalidades. Cada versión es soportada oficialmente por una o varias (pocas) distribuciones de Ubuntu. En este trabajo se ha trabajado sobre ROS Kinetic instalado sobre Ubuntu 16.04 LTS.

6.1.1. Conceptos

Filosofía de ROS

Para comprender con mayor facilidad la filosofía en la que se basa este software es importante entender aspectos más concretos de la misma:

Está basado en un comportamiento peer-to-peer, o lo que es lo mismo, todos los pequeños programas que pueden formar un sistema ROS se comunican mediante mensajes que circulan de uno a otro sin pasar por una rutina de servicios central.

A su vez, se trata de un sistema basado en herramientas. Para el caso de ROS tenemos herramientas pequeñas que realizan tareas muy específicas, y éstas pueden ser modificadas o sustituidas por mejores implementaciones que realicen mejor la tarea deseada.

Otra de las cosas que destacan de ROS es, como ya se ha mencionado antes, que soporta un gran número de lenguajes de programación. Esto permite que para cada tipo de tarea o de función que se busca podamos utilizar el lenguaje más apropiado para ello, permitiendo así una mayor eficiencia en los programas utilizados.

Por último, mencionar que se trata de código de acceso libre y gratuito. ROS está bajo una licencia BSD, que nos permite un uso tanto comercial como no.

Terminología

Antes de adentrarnos en el mundo de ROS es necesario explicar parte de la terminología y algunos conceptos básicos de este software:

- Los Paquetes, que son la forma básica y principal de agrupar la información dentro de ROS. En estos paquetes podemos encontrar nodos, las librerías dependientes de ROS, conjuntos de datos, archivos de configuración o cualquier tipo de archivo o información que sea útil junto al resto de contenidos del paquete. Son la unidad más básica indivisible que se puede crear y publicar en ROS.
- Los metapaquetes, que se tratan de paquetes especializados cuya función es agrupar paquetes relacionados entre sí o que tienen una funcionalidad común.
- Los manifiestos: se tratan de archivos con extensión .xml que nos dan metadatos sobre los paquetes como por ejemplo: el nombre, la versión, la descripción, y más información adicional.

- Mensajes: son los ficheros que describen las estructuras de datos para los mensajes enviados en ROS, esto es, son los ficheros que contienen la información que se intercambia en cualquier proceso ROS.
- Servicios: son los ficheros que definen las solicitudes y las estructuras de datos de respuesta requeridas por los procesos de ROS.

Sistema de computación a nivel gráfico de ROS

Todos los procesos de este sistema están conectados mediante una red peer-to-peer en la que todos los elementos de la red son capaces de procesar datos y a su vez están todos conectados entre sí.

Los conceptos más básicos son los siguientes:

Master: Es el que da nombre y un "almacén" al resto del proceso. Sin el Master, no sería posible para los nodos encontrarse unos a otros, enviarse mensajes o invocar servicios.

Nodos: Son los procesos que realizan los cálculos. ROS está diseñado para trabajo modular detallado, por lo que el proceso de control de un robot está formado por diversos nodos. Estos nodos se compilan mediante la utilización de librerías de biblioteca de ROS como roscpp y rospy.

Servidor de parámetros: Forma parte del Master, permite el almacenaje de datos en una posición centralizada usando claves.

Mensajes: Son el método de comunicación entre nodos, se tratan de estructuras simples de datos que constan de variables. Los tipos de variable estándar están permitidas, así como matrices de este tipo de variables.

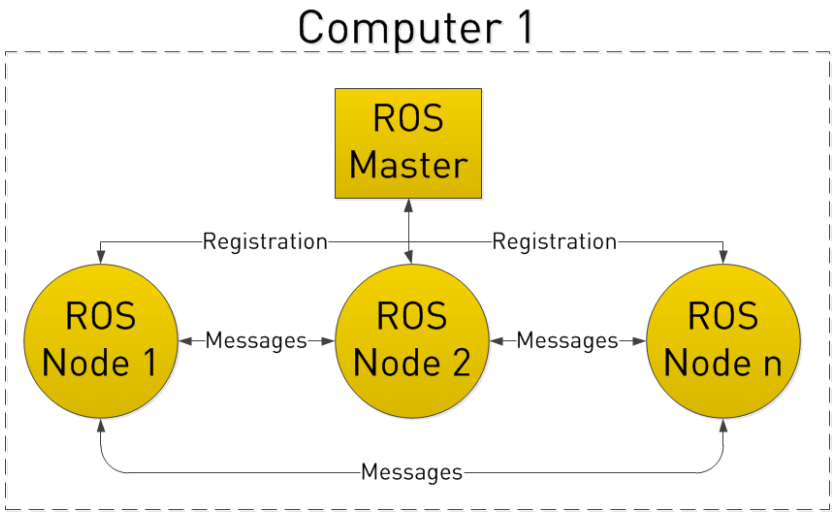


Figura 14: Estructura Master y Nodos Ros

Topic: Son el modo de transporte de los mensajes basados en un sistema de publicación/subscripción. Un nodo puede publicar un topic por el que puede enviar mensajes. Para recibir estos mensajes, otro nodo ha de suscribirse a este topic. A un topic pueden subscribirse tantos nodos como se deseen, así como un nodo puede publicar tantos topics como se necesiten.

Servicios: El método de comunicación publicación/subscripción a través de topics es una forma de comunicación muy flexible, pero solo permite la comunicación en un solo sentido y se trata de una comunicación "de uno a muchos", por lo que no es apropiado para solicitudes y respuestas, que suelen ser necesarios en un sistema distribuido, por lo que para esto ROS utiliza los servicios, que se definen como un par de estructuras de mensaje, uno para solicitud y otro para respuesta.

Estos conceptos serán ampliados y contextualizados más adelante.

Servicio roscore

Roscore es un servicio que nos propone ROS y que permite a los nodos que se puedan comunicar entre ellos a través del envío de mensajes. Hay que tener abierto un preceso roscore para cualquier aplicación de Ros.

Cada nodo se conecta al roscore en cuanto se inicia y le envía los detalles de los mensajes que desea enviar y a los que se quiere subscribir. También establece las conexiones peer-to-peer con otros nodos cuando uno nuevo aparece, por lo que podemos afirmar que no se trata de una conexión cliente/servidor como las que tenemos para servicios web, pero tampoco una red peer-to-peer pura; en la arquitectura ROS tenemos algo intermedio.

Cada nodo informa al roscore de a qué mensajes quiere subscribirse y qué mensajes envía y es entonces cuando roscore informa de las direcciones a aquellos interesados en dichos mensajes.

Roscore también nos provee de un servicio de parámetros que es usado por los nodos para su configuración. Proporciona a los nodos la posibilidad de guardar y recuperar datos arbitrarios como pueden ser descripciones de robots, parámetros de algoritmos, etc. Como la mayoría de los servicios de ROS, tenemos un comando sencillo denominado rosparam.

6.1.2. Catkin y workspaces

ROS contiene un gran gran número de paquetes, por lo que podemos tener muchas dependencias entre paquetes a priori que además utilizan distintos lenguajes de programación.

Debido a esta complejidad de ROS, muchos usuarios no compilan ROS desde el código fuente directamente, sino que instalan una serie de paquetes precompilados (como los archivos .deb en Ubuntu). Estos paquetes se instalan en el sistema, típicamente /opt/ros/<nombre_distribución>.

Catkin es el sistema de compilación oficial de ROS. Combina macros CMake con scripts Python para proporcionar mayor funcionalidad que el flujo normal de CMake. Este paquete nos proporciona generaciones de targets a partir de puro código y permite que pueda ser usado por el usuario final. Estos tárgets se pueden traducir en librerías, programas ejecutables, interfaces o cualquier cosa que no sea código estático.

Para construir estos paquetes, el sistema de construcción necesita información como la localización de los componentes de la cadena de herramientas (como el compilador de c++), la localización de los códigos fuente, las dependencias tanto de los códigos como externas, donde se encuentran esas dependencias, qué targets se deben construir y donde. Esta información es la que, con el sistema CMake, encontramos en los ficheros llamados habitualmente “CMakeLists.txt” que se encuentran en todos los paquetes de ROS.

Llamamos workspace (espacio de trabajo) a la carpeta donde se modifican, compilan e instalan los paquetes catkin.

El principal objetivo de catkin es facilitar la compilación y el funcionamiento de código ROS a través de la utilización de herramientas y convenciones para reducir el proceso de desarrollo. Esta herramienta es en la que se centra la construcción de los espacios de trabajo (workspaces a partir de ahora).

El workspace supone por tanto, una ampliación de la instalación de ROS, por lo que los paquetes desarrollados en el workspace funcionan exactamente igual que si el paquete estuviera instalado en el núcleo de ROS. Esto permite a los desarrolladores una mayor comodidad al desarrollar nuevos paquetes.

Por tanto, antes de realizar cualquier programa en ROS es necesario crear un workspace donde se guardará y se trabajará con el código. Se puede tener dentro de una máquina una gran cantidad de workspaces, pero solo se puede trabajar con uno a la vez .

Destacar los comandos más importantes para trabajas con workspaces, que en este caso son el catkin_init_workspace que nos crea el fichero CMakeList.txt y el comando catkin_make que es la herramienta estándar para compilar el código del workspace. Catkin_make nos genera dentro del workspace dos nuevos directorios: build y devel. El primero es donde

almacenaremos los resultados de la ejecución del catkin, como pueden ser ejecutables y librerias, mientras que en el segundo tenemos una gran cantidad de ficheros y directorios de configuración del workspace.

6.1.3. Paquetes

Como ya se ha comentado anteriormente, un paquete es la unidad fundamental de agrupar la información en ROS. Estos paquetes se almacenan dentro del directorio src que se ha creado previamente en el workspace y es imprescindible que contengan tanto un fichero CMakeLists.txt como un package.xml. Estos archivos describen al comando catkin cómo debe interactuar con ellos cuando es ejecutado.

En estos paquetes es donde almacenamos los programas realizados en otros lenguajes de programación y los ficheros que son necesarios para que la funcionalidad del paquete pueda ejecutarse correctamente.

6.1.4. Comandos importantes de ROS

Roscore, rosrun y roslaunch

Roscore: Como se ha explicado anteriormente, es el primer comando que se ha de lanzar a la hora de ejecutar cualquier aplicación de ROS y permite la comunicación entre nodos. Hay que tener abierto un roscore para poder ejecutar cualquier aplicación de Ros.

Rosrun: nos permite lanzar los programas sin que exista la necesidad de buscar la ubicación de los mismos dentro del sistema de archivos de Ubuntu (esto puede ser tedioso si no se está familiarizado con el uso de la terminal). Únicamente utiliza el nombre del paquete y el programa concreto que se desea ejecutar, sin importar dónde se encuentre la terminal situada dentro del sistema de ficheros. La única condición que se ha de cumplir previamente es que se haya ejecutado el comando roscore y éste esté activo.

Roslaunch: nos permite lanzar varios nodos de forma simultánea. A la hora de ejecutarse, se hace de forma muy similar al rosrun, pero en este caso se necesita un tipo de ficheros especiales con extensión .launch. Son ficheros XML que almacenan toda la información necesaria de los nodos que se quiere lanzar. También nos permite ejecutar programas de forma remota en otros ordenadores a través del protocolo ssh, esto se utilizará posteriormente en la comunicación con el rbcar.

Otros comandos

Tenemos otra serie de comandos, mucho más sencillos, pero no por ello menos importantes que nos ayudarán a realizar todas las acciones que deseamos y nos permitirán navegar por el sistema de archivos de ROS:

- rospack: Nos permite recibir información sobre los distintos paquetes que se encuentran en nuestra máquina, como por ejemplo, que devuelva por pantalla la ubicación de un paquete con el argumento 'find' tras el comando.

- roscd: Este comando es el equivalente a cd para ROS. Nos permite la navegación por las distintas carpetas o paquetes desde la terminal de Ubuntu.

- rospd: Similar al comando anterior, pero este comando recuerda la última ubicación en la que estábamos.

- rosls: Nos da una lista de los paquetes que podemos encontrar en el directorio en el que nos encontramos actualmente.

- rosls: Similar al anterior, pero esta vez devuelve una lista de todos los archivos que podemos encontrar en el directorio.

- rosed: Nos permite la modificación de un fichero que se encuentra dentro de un determinado paquete desde la terminal.

- roscp: Este comando es utilizado para copiar un fichero de un paquete a otro.

Estos comandos se encuentran dentro de un paquete que está incorporado a ROS, llamado rosbash, y que además nos permite la opción de, a la hora de trabajar con ellos, autocompletar cuando presionamos la tecla tabulador, lo cual ahorra mucho tiempo y errores de escritura al trabajar desde la terminal directamente.

6.1.5. Sistemas de comunicación entre nodos

Topics

Topics: la forma más común de comunicación entre nodos. Un topic es un nombre que se le da la retransmisión de mensajes de un tipo definido. Se utilizan junto con un mecanismo de comunicación basado en publicaciones/suscripciones. Antes de enviar la información los nodos han de informar (como ya se ha mencionado anteriormente, al roscore) del nombre del topic y del tipo de datos que se van a transmitir, entonces pueden realizar el envío de información. Para realizar la suscripción es algo similar, pero informando de qué topic estamos recibiendo información.

Dentro de los nodos es donde se indica a qué topics se publica/suscribe. Cada lenguaje de programación establece el tipo de mensaje y la forma de transmisión de datos como corresponde.

Dentro de un programa se puede tener un gran número de topics transmitiendo mensajes, además de que puede ser necesario conocer cierta información referente a éstos, por lo que ROS habilita también un comando, rostopic, que simplifica mucho la tarea de trabajar y conocer los parámetros de los topics.

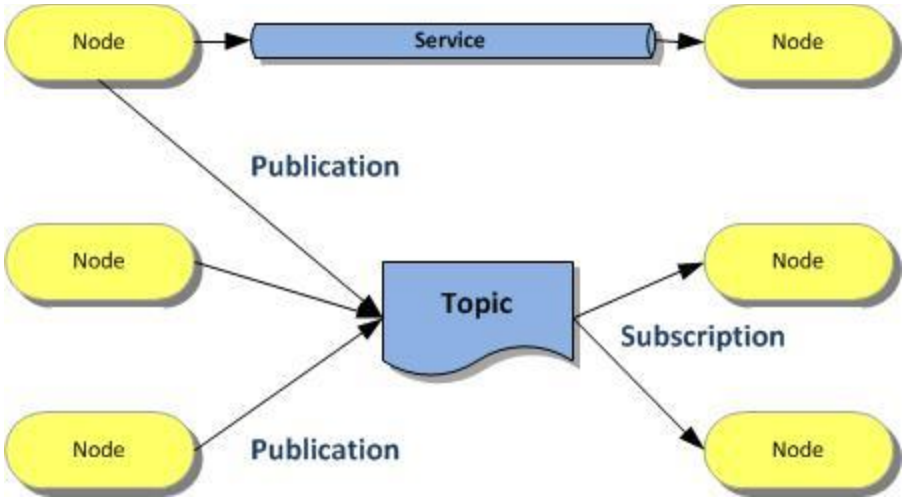


Figura 15: Estructura Tòpics Ros

Services

Services: Otro sistema de transmisión de datos que nos ofrece ROS. Se tratan de llamadas a procedimientos remotas y sincronizadas. Nos permiten que un nodo realice la llamada a una función que ejecuta otro nodo. La información recibida y enviada mediante este procedimiento se define de forma muy similar a los mensajes. El servidor (aquell que envía el servicio) especifica una callback (una retrollamada) para lidiar con la solicitud, y anuncia el servicio. El cliente (el que solicita el servicio) entonces accede al servicio mediante un proxy local.

Las llamadas a servicios son muy apropiadas para funcionalidades que, a lo largo de nuestra aplicación utilicemos un número reducido de veces y precisen obligatoriamente de un tiempo de proceso, por ejemplo, algún tipo de cálculo que se quiera distribuir a otros ordenadores. Otras funciones discretas que pueden realizarse perfectamente con services pueden ser el encendido de un sensor o la realización de una fotografía con una cámara.

7. QT

Qt es un framework multiplataforma orientado a objetos para el desarrollo de software en escritorio, sistemas móviles y sistemas embebidos. Las plataformas soportadas por Qt incluyen Linux, OS X, Windows, Android, iOS y otros.

Qt no es un lenguaje de programación por sí solo, es un framework escrito en C++. Se utiliza un preprocesador MOC (Meta-Object Compiler) para extender el lenguaje C++ con funcionalidades como por ejemplo las señales, aunque la principal funcionalidad añadida es la posibilidad de crear interfaces de usuario en C++.

Despues de la precompilación, el MOC combina los archivos de código fuente escritos en C++ enriquecido con Qt y genera archivos escritos en C++ estándar a partir de ellos. Estos archivos generados pueden ser compilados con un compilador usual de C++ como puede ser gcc, icc, MinGW y msvc.

7.1. MOC (Meta-Object Compiler)

El Meta-Object Compiler (de ahora en adelante, MOC) es el programa que maneja las extensiones Qt de C++.

El MOC lee la cabecera del archivo C++. Si encuentra una o más declaraciones de clases que contengan alguna librería de Qt, produce un archivo de código fuente escrito en C++ conteniendo el código del meta-objeto para esas clases. Estos archivos generados deberán ser compilados y vinculados con la implementación de la clase.

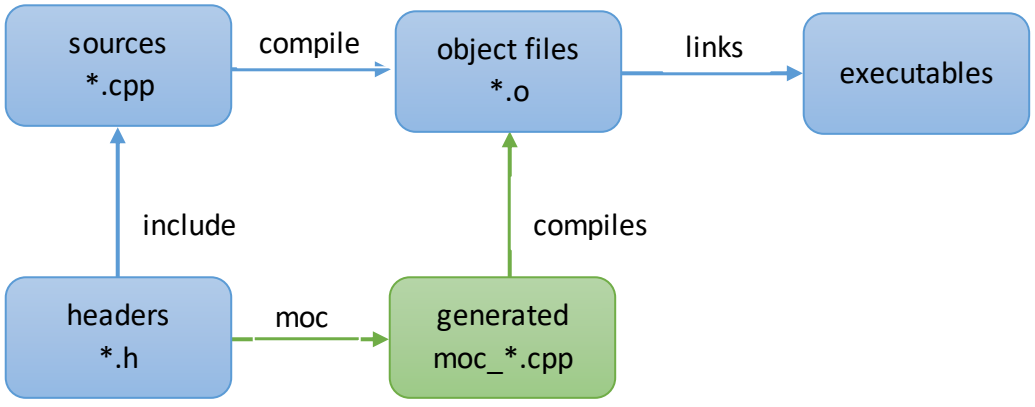


Figura 16: Compilación con MOC

El resultado de este método de compilación son clases que son conscientes de si mismas en tiempo de ejecución, por lo que podemos saber que tipo de clases son, identificar una clase específica e incluso pasar mensajes entre clases. Este concepto es el núcleo de la comunicación Qt por medio de señales.

7.2. QMake

La preparación de la compilación en un proyecto Qt se hace usando un Makefile y la herramienta de compilación estándar, aunque el proceso de generación manual del Makefile puede resultar un tanto engorroso. Por lo tanto, a pesar de que se puede utilizar cualquier sistema de compilación, como puede ser cmake, para construir las aplicaciones, Qt proporciona su propio sistema de compilación, QMake.

QMake es una herramienta que simplifica el proceso de compilación para proyectos de diferentes plataformas. Qmake automatiza la generación de archivos Makefile, por lo que tan solo se necesitan unas pocas líneas de información para crear un archivo Makefile. Qmake se puede usar tanto si se usa Qt como si no.

Qmake genera un Makefile basándose en un archivo de proyecto con extensión .pro, los cuales son creados por el desarrollador y tienen una estructura determinada, normalmente más simple que un Makefile. También contiene funcionalidades adicionales que simplifican el proceso de compilación con Qt, ya que añade automáticamente reglas para MOC y UIC (User Interface Compiler).

```
#############################################################################
# Automatically generated by qmake (3.0) Mon Nov 14 08:32:53 2016
#############################################################################

TEMPLATE = app
TARGET = Qt1
INCLUDEPATH += .
# note we need to add wigets module for gui
QT += widgets
# Input
SOURCES += main.cpp
```

Figura 17: Ejemplo app.pro

7.3. Módulos Qt

Las librerías de las que está compuesto Qt no se encuentran todas juntas, sino que se dividen en módulos. Estos módulos se pueden clasificar de dos formas diferentes:

Por un lado, podemos distinguir entre los módulos considerados 'esenciales' y los 'add-ons' o adiciones. Los módulos esenciales definen la base de Qt en todas las plataformas; son módulos generales y útiles en la mayoría de las aplicaciones Qt. Los módulos 'añadidos' proporcionan valor añadido para diferentes propósitos, pero pueden no ser compatibles en todas las plataformas.

Por otro lado, podemos clasificar los módulos entre aquellos orientados al desarrollo de interfaces gráficas (GUI) y aquellos en los que su objetivo es otro, por ejemplo, para el acceso a bases de datos, o la comunicación en redes TCP/IP.

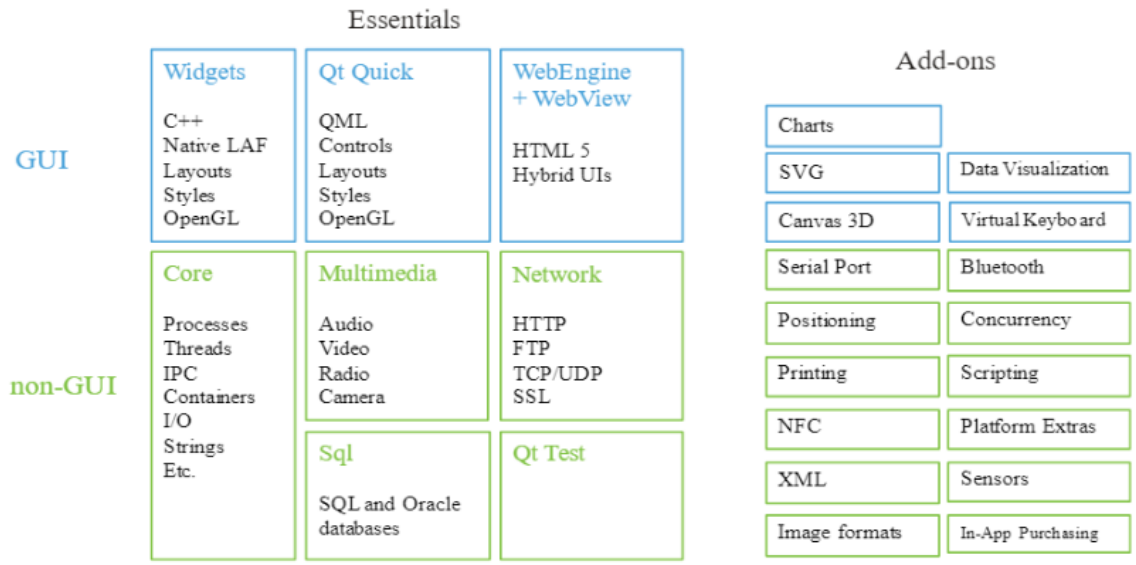


Figura 18: Módulos QT

Para poder utilizar una librería de un determinado módulo no basta con añadir un *include* en el archivo de código fuente que se vaya a necesitar, sino que es necesario importar dicho módulo declarándolo en el qmake.pro.

Por ejemplo, supongamos que necesitamos incluir la librería Layouts a un fichero de código. En este caso, deberíamos añadir el módulo Widgets en el archivo qmake.pro:

```
QT += widgets
```

Y a continuación, ya podríamos incluir la librería en el fichero de código fuente:

```
#include <QtLayouts>
```

7.4. QML

QML (del inglés, Qt Meta Language) en un lenguaje basado en JavaStript creado para el diseño y la creación de aplicaciones orientadas a la interfaz de usuario. Forma parte del módulo QtQuick, y por lo tanto, será necesario importar dicho modulo.

El lenguaje QML se usa principalmente para aplicaciones móviles, donde la entrada táctil, las animaciones fluidas y una buena experiencia de usuario son cruciales. Los documentos QML describen un árbol de elementos. Los elementos de QML que vienen por defecto con Qt son un sofisticado conjunto de bloques, elementos gráficos (como rectángulos o imágenes) y comportamientos (como animaciones y transiciones). Estos elementos pueden ser combinados para construir componentes más complejos incluso para completar aplicaciones conectadas a Internet.

La figura X muestra un ejemplo de código escrito en QML, mientras que la figura X muestra el resultado que se mostraría por pantalla:

```
import QtQuick 2.0

Rectangle {
    width: 400; height: 300
    Text {
        text: qsTr("Hello")
        font.pointSize: 25;
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
    }
}
```

Figura 19: Ejemplo QML

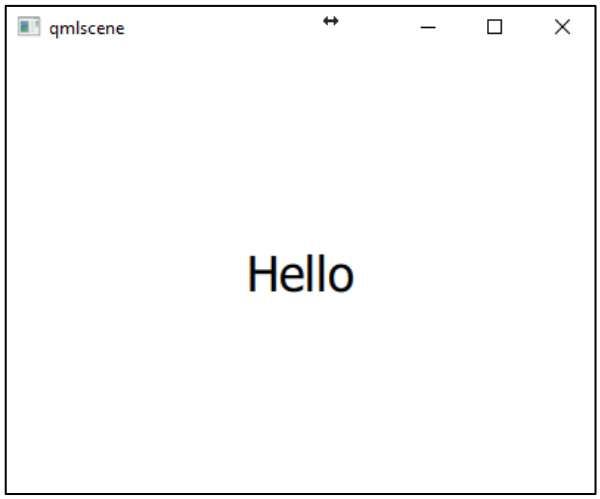


Figura 20: Resultado QML

7.5. Qt creator

Qt Creator es un IDE multi plataforma programado en C++, JavaScript y QML el cual es parte de SDK para el desarrollo de aplicaciones con Interfaces Gráficas de Usuario (GUI) con las bibliotecas Qt, Incluye un depurador visual y un diseñador de Interfaces de Usuario integrado. Entre las funcionalidades del editor se incluye el resaltado de errores de sintaxis y el autocompletado. Los sistemas operativos que soporta en forma oficial son: GNU/Linux Mac OS X Windows. Este IDE será el utilizado para el desarrollo del trabajo.

8. DESARROLLO PRÁCTICO

8.1. Instalación

Para el desarrollo del proyecto, ha sido necesaria la instalación de varios paquetes. En concreto, ha sido necesario instalar:

- La distribución ROS Kinetic con los paquetes QT mediante el comando:

  ```
  sudo apt-get install ros-kinetic-qt-build
  ```

- El software QtCreator con un plugin para crear proyectos en ROS:

  ```
  sudo apt-get install qt57creator-plugin-ros
  ```

- Qt 5.9.3 Opensource a través de la propia página web de Qt.

  https://www.qt.io/download

8.2. Integración

En una compilación típica de un paquete ROS, se usa el comando catkin_make, el cual acude al fichero CMakeLists.txt en busca de las dependencias del proyecto, y ejecuta CMake para compilar el proyecto.

En cambio, para compilar software desarrollado con módulos Qt, se usa el comando qmake, es cual accede al fichero del proyecto con extensión .pro, que contiene las dependencias del proyecto con un estilo mucho más simplificado, crea automáticamente el Makefile correspondiente incluyendo reglas para MOC (Meta Object Compiler) y UIC (User Interface Compiler) y ejecuta el CMake para compilar el programa.

Vemos pues, que los métodos de compilación para cada uno de los aspectos del proyecto son diferentes, por lo que la integración de Qt en ROS no es inmediata. Es necesario adaptar el archivo CMakeLists.txt para que al ejecutar el comando *catkin_make*, se ejecuten también todas las reglas necesarias para compilar un proyecto Qt.

En concreto, para realizar las mismas tareas que realizaría el comando qmake, es necesario realizar las siguientes tareas:

Encontrar los módulos QT a utilizar. Para poder utilizar los diferentes módulos de QT, es necesario incluirlos explícitamente en el CMakeLists.txt mediante el siguiente comando:

```
find_package(Qt5 COMPONENTS Core Gui Qml Quick Widgets REQUIRED)
```

Indicar dónde puede encontrar tanto los recursos utilizados, las cabeceras del código fuente y el propio código fuente:

```
file(GLOB QT_RESOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} resources/*.qrc)
file(GLOB_RECURSE QT_MOC RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} FOLLOW_SYMLINKS include/gui/*.hpp)
file(GLOB_RECURSE QT_SOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} FOLLOW_SYMLINKS src/*.cpp)
```

Ejecutar el MOC (Meta Object Compiler): Ejecutaremos el compilador de los objetos pertenecientes a Qt para transformarlos en archivos .cpp. Para ello, le indicaremos cuales son los archivos cabecera de código fuente (.hpp) y creará aquellos objetos Qt que encuentre:

```
QT5_WRAP_CPP(QT_MOC_HPP ${QT_MOC})
```

Compilar los archivos .cpp:

```
add_executable(gui ${QT_SOURCES} ${QT_RESOURCES_CPP} ${QT_FORMS_HPP} ${QT_MOC_HPP})
```

Vincular las librerías Qt:

```
qt5_use_modules(gui Quick Core Widgets)
target_link_libraries(gui ${QT_LIBRARIES} ${catkin_LIBRARIES})
target_include_directories(gui PUBLIC include)
```

Instalar finalmente la aplicación en su ruta destino:

```
install(TARGETS gui RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

El contenido del CMakeLists.txt final está incluido en el Anexo I al final de este documento, junto con una breve explicación de los comandos utilizados.

8.3. Estructura del proyecto

El proyecto presenta la siguiente estructura de ficheros:

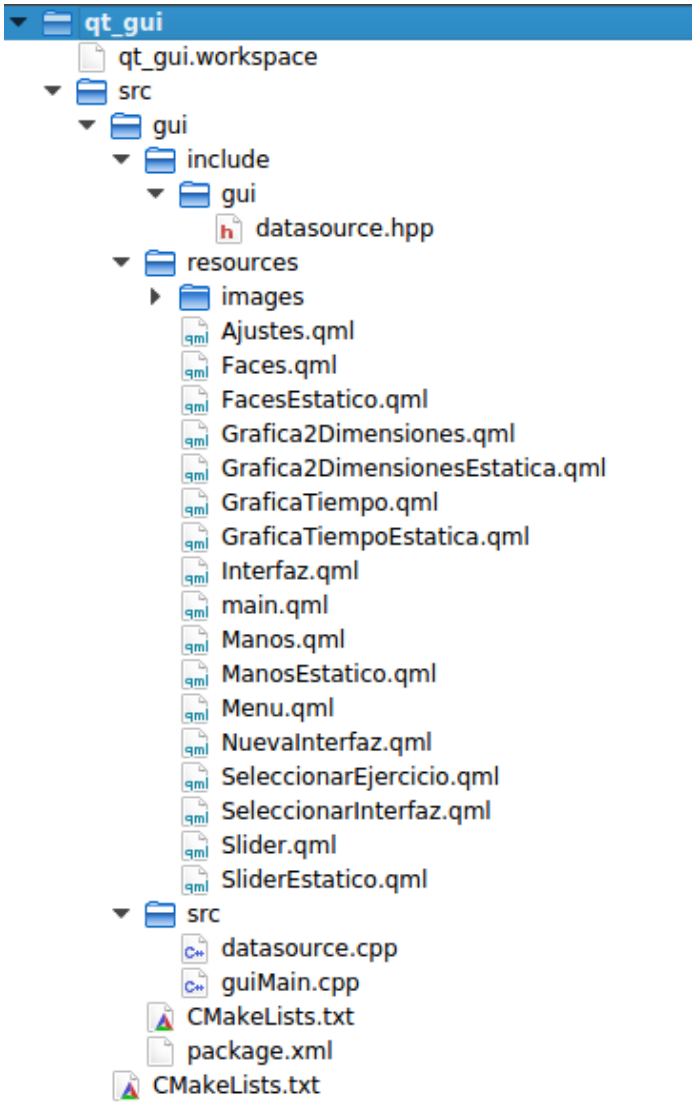


Figura 21: Estructura proyecto

Se puede observar como la carpeta raíz corresponde al *workspace* de *catkin*, y dentro de la carpeta *src* es donde se encuentra el paquete "*gui*", que contiene todo el código fuente del proyecto.

En la carpeta *include* se encuentra la cabecera del *datasource,* que define los métodos y variables que permiten la comunicación entre la interfaz y los datos. En la carpeta *resources* se encuentran todos los ficheros .qml que definen las interfaces de la aplicación, así como las imágenes utilizadas dentro de la carpeta images.

Por último, en la carpeta src se incluye el fichero datasource.cpp, que implementa las funciones definidas en la cabecera; y el fichero guiMain.cpp, que contiene la función main() y por tanto, es donde se inicializa la aplicación.

8.4. Funcionalidades

Para el desarrollo de la aplicación, se han definido una serie de funcionalidades que debe cumplir la aplicación. Estas funcionalidades concretan el objetivo del trabajo y definen exactamente qué es lo que la aplicación hará y no hará.

- Representación a tiempo real de valores de referencia y valores de señales de un robot.
- Obtención de los valores de referencia a partir de ficheros.
- Obtención de los valores de las señales del robot a partir de subscripciones a tópicos en ROS.
- Elección del tipo de gráfica, es decir, de que manera se representarán las referencias y señales a comparar.
- Elección del tipo de feedback, es decir, de que manera se le notificará al usuario de su rendimiento en el ejercicio.
- Elección de las señales a medir, obtenidas a partir de los tópicos presentes en ese momento en el master de ROS. El máximo número de señales a medir simultáneamente serán dos.
- Posibilidad de elegir qué ejercicio realizar, es decir, que conjunto de señales de referencia se van a utilizar.
- Guardar los resultados de un ejercicio tras su finalización para su posterior análisis.
- Posibilidad de modificar las rutas donde estén los diferentes ficheros utilizados/generados
- Posibilidad de modificar la frecuencia de actualización de las gráficas.
- Posibilidad de modificar los parámetros relativos a los límites de las gráficas, es decir, los máximos y mínimos de los ejes X e Y.
- Creación de interfaces directamente a través de ficheros.
- Adicción de nuevos elementos sin necesidad de compilación, ya sea en forma de nuevas gráficas o nuevos tipos de feedback.
- Interfaces independientes del tamaño de la pantalla, adaptables a cualquier tamaño.

Destacar que, de cara a la realización de proyectos en la vida laboral, es importante definir dichas funcionalidades para evitar cualquier tipo de conflicto o malentendido sobre el resultado final de la aplicación.

8.5. Diseño interfaz

El diseño de las interfaces es un aspecto muy importante a la hora de desarrollar una aplicación, ya que este es el método de interacción principal con el usuario. En este apartado se muestran los diferentes pasos seguidos en el diseño de interfaces: empezando por el análisis de perfil del usuario medio, el diagrama de las pantallas y por último, el diseño de cada pantalla.

8.5.1. Análisis perfil usuario

Antes de empezar a definir una interfaz, es importante saber a qué tipo de usuarios estará orientada, ya que de ello dependerá el nivel de complejidad y profundidad que tendrá nuestra aplicación.

En este caso, la aplicación está orientada a su uso junto con el robot paralelo 3PRS para la rehabilitación de miembros inferiores. Por tanto, es de suponer que la aplicación será usada principalmente por dos tipos de personas: los médicos encargados de las rehabilitaciones, y los pacientes que se someten a ellas.

No podemos suponer que, ni los médicos ni los pacientes tengan conocimientos informáticos, por lo que es crucial que la aplicación sea amigable al usuario. Es decir, las interfaces deben ser lo más sencillas posibles, sin que sea necesario navegar por una gran cantidad de submenús y que los elementos sean lo más visuales e intuitivos posibles.

Este ha sido uno de los pilares fundamentales a la hora de desarrollar las interfaces, y se ha tenido en cuenta durante todo el proceso.

8.5.2. Diagrama de pantallas

El diagrama de pantallas representa el flujo entre pantallas dentro de una aplicación. A continuación, se presenta un diagrama en el que se muestran las transiciones entre las diferentes pantallas que tiene la aplicación. Si dos pantallas están unidas por una flecha significa que es posible navegar entre esas pantallas en la dirección de la flecha:

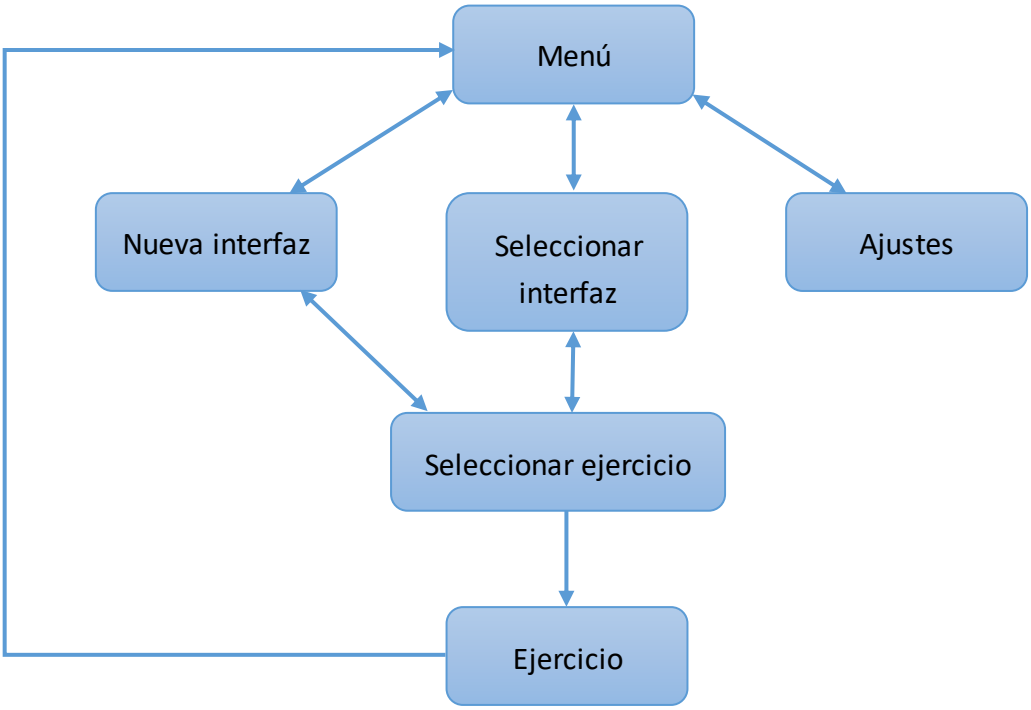


Figura 22: Diagrama de pantallas

La aplicación empieza con el menú principal, desde donde se puede acceder a las pantallas de **Nueva Interfaz**, **Seleccionar Interfaz** y **Ajustes**, así como salir de la propia aplicación.

En la pantalla **Seleccionar Interfaz**, el usuario es capaz de seleccionar entre varias interfaces predefinidas. Las cuales se definen mediante ficheros con un formato especifica en la ruta indicada. Este tema se abordará más adelante en sucesivos apartados.

En caso de que la interfaz deseada no se encuentre entre las listadas, el usuario tiene la posibilidad de crear una nueva, accediendo a la pantalla Nueva Interfaz. En ella, el usuario elegirá el tipo de grafica que desea, cuál será el método de feedback (explicado más adelante), las señales que desea representar y el nombre que desea darle a la aplicación.

Tanto si se ha seleccionado una interfaz existente como si se ha creado una nueva, el usuario pasa a seleccionar un ejercicio predefinido, los cuales se definen también mediante ficheros y que contienen las señales de referencia a seguir por el paciente.

Una vez cargado el ejercicio, el usuario pasa por fin a la pantalla de la ejecución del ejercicio de rehabilitación. En esta pantalla se llevará a cabo el ejercicio propiamente dicho.

Cuando terminamos el ejercicio y pulsamos en el botón de **parar**, el resultado del ejercicio se guardará en otro fichero para su posterior análisis del médico y la aplicación se dirigirá de nuevo al **Menú principal**, lista para empezar un nuevo ejercicio.

8.5.3. Diseño de pantallas

Después de varias iteraciones, se muestran a continuación las pantallas finales de la aplicación. Todas las pantallas se han diseñado de manera 'responsive', es decir, se mantiene el aspecto aunque se cambien las dimensiones de la pantalla. Esto permite que la aplicación se adapte perfectamente a diferentes tamaños de pantalla:

Menú principal:

Figura 23: Menú principal

La pantalla del menú principal es la primera pantalla que aparece al ejecutar la aplicación. Proporciona vínculos a las pantallas de **'Nueva Interfaz'**, **'Seleccionar interfaz'**, y **'Ajustes'**, así como una manera simple de salir de la aplicación.

Seleccionar interfaz:

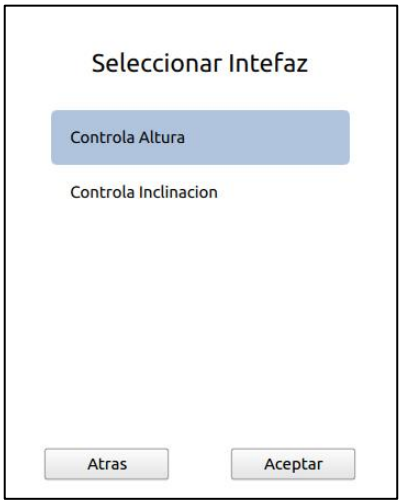


Figura 24: Seleccionar Interfaz

En la pantalla **Seleccionar interfaz** se muestran las interfaces predefinidas disponibles. El usuario es capaz de seleccionar una de ellas y pulsar aceptar, con lo que se mostraría la pantalla de **Selección de ejercicio**. En caso de pulsar el botón atrás, la aplicación vuelve al menú principal.

Seleccionar ejercicio:

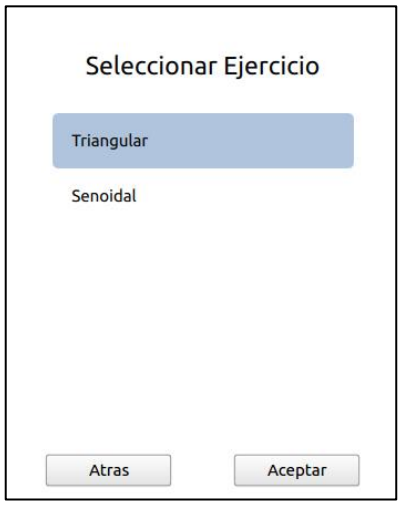


Figura 25: Seleccionar ejercicio

En la pantalla seleccionar interfaz se muestran los diversos ejercicios disponibles. El usuario es capaz de seleccionar uno de ellos y pulsar aceptar, con lo que la aplicación se dirigiría a la pantalla de ejecución de ejercicio. En caso de seleccionar el botón atrás, la aplicación vuelve a la pantalla de selección de interfaz, o la de crear nueva interfaz, según corresponda.

Ajustes:

Figura 26: Ajustes

En esta pantalla se muestran los diversos campos configurables que afectan a la aplicación. En este caso, estos campos incluyen las 3 rutas que se usan para acceder a los ficheros de ejercicios, interfaces y resultados, así como la frecuencia de actualización de la gráfica, expresado en milisegundos.

Nueva interfaz:

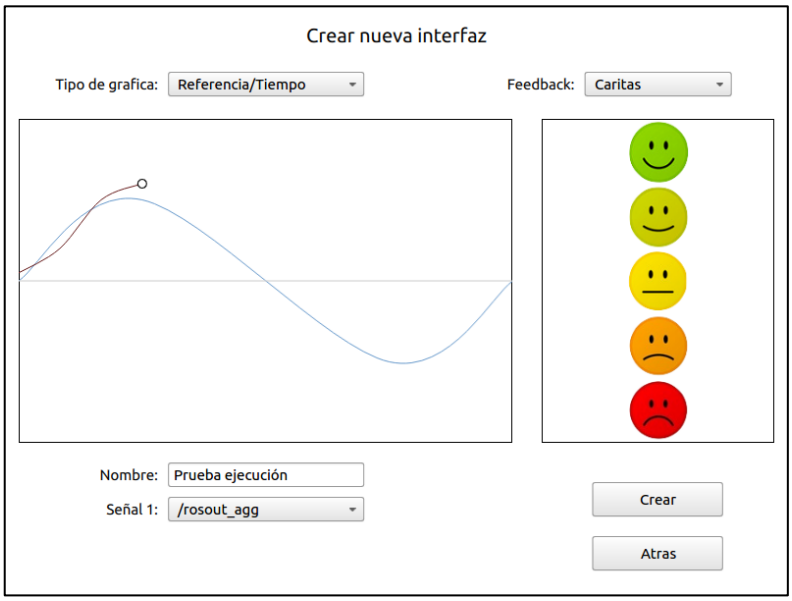
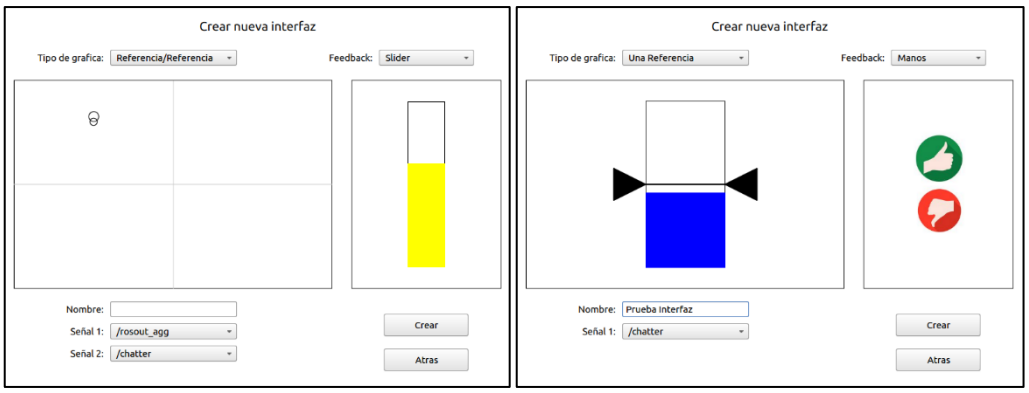


Figura 27: Nueva interfaz

En esta pantalla, el usuario puede crear nuevas interfaces diferentes a las predefinidas. Para ello, se ha de seleccionar el tipo de gráfica, el tipo de feedback, las señales de referencia y dándole un nombre a la gráfica. En el caso de que alguno de estos campos no esté informado correctamente, se mostrará una pantalla de alerta indicando cual es el campo que falta por informar.



Figuras 28 y 29: Ejemplos diferentes configuraciones interfaz.

Figura 30: Alerta campo vacío

Ejecución de ejercicio:

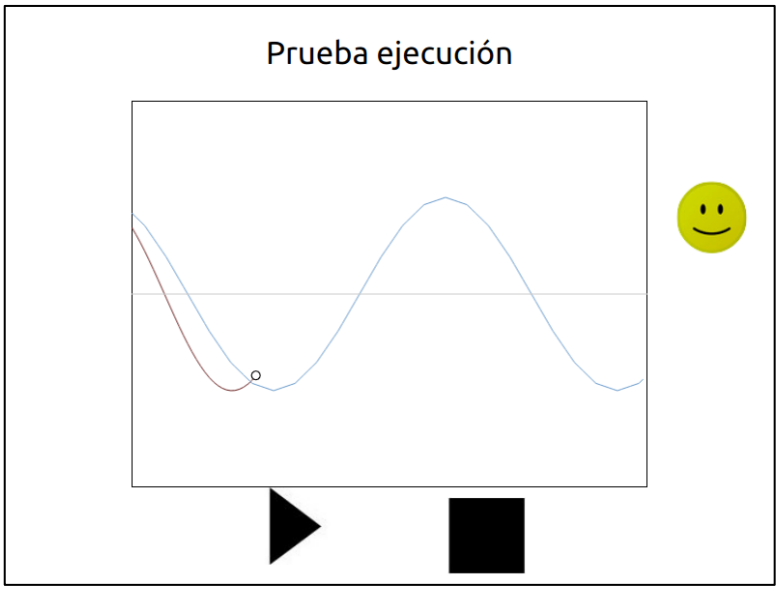


Figura 31: Ejecución de ejercicio

Ya sea cargando una interfaz a partir de ficheros como creándola eligiendo sus parámetros, la pantalla de ejecución de ejercicio tendrá una forma similar a la de la figura, permitiendo iniciar y parar el ejercicio.

8.5.4. Gráficas

Las gráficas son el medio utilizado para informar al usuario tanto del estado del valor de la referencia a seguir como del valor de la señal del robot que el mismo está provocando. En algunos casos incluso permitirá ver cuál será el valor de la referencia en momentos futuros.

La elección de utilizar gráficas fue debido a la necesidad de utilizar elementos visuales intuitivos para que su comprensión por parte del usuario fuera lo más inmediata posible.

En concreto se han desarrollado tres posibles gráficas (que se explicarán más adelante), pero en el caso de que se deseara modificar o añadir nuevas gráficas, su adición resultaría sencilla siguiendo los pasos descritos en el Anexo II, en el que se describe como añadir elementos a la interfaz. Además, teniendo en cuenta que los archivos QML que forman las interfaces son leídos en tiempo de ejecución y no en tiempo de compilación, no sería necesario en ningún momento volver a compilar la aplicación.

Actualización de grafica

Para que la gráfica muestre las señales y referencias en tiempo real, es necesario que se actualice cada cierto periodo de tiempo T, configurable a partir de la pantalla Ajustes de nuestra aplicación.

Cada T milisegundos entonces, se solicitará al datasource, tanto el valor de la señal proveniente del robot en ese momento, como el valor correspondiente a la referencia para el mismo momento.

En el caso en el que la gráfica represente también valores futuros y pasados, como es el caso de la gráfica referencia tiempo, se solicitará también la actualización de los vectores que representan estos valores.

Dibujado de gráfica

Las gráficas utilizadas están formadas por un elemento de tipo Canvas en QML, el cual permite el dibujado de múltiples líneas tal como si fuera un lienzo.

Su funcionamiento se asemeja al modo de programación de OpenGL, permitiendo dibujar una gran cantidad de polígonos y redibujarlos a voluntad, a una gran tasa de refresco en caso de ser necesario, pues su rendimiento apenas se ve afectado debido a que es un elemento pensado específicamente para este fin.

Por tanto, una vez obtenidos los valores necesarios, se procederá a redibujar la propia gráfica, llamando al método canvas.requestPaint().

```
ctx.save();
ctx.clearRect(0,0,canvas.width,canvas.height);
ctx.beginPath();
ctx.moveTo(0, height/2 - height*dataSource.getPath(0)/cotaTotal);
for(var i=1;i<tiempoReal;i++){
    ctx.lineTo(width*i/tiempoReal,height/2 - height*dataSource.getPath(i)/cotaTotal)
}
ctx.moveTo(width*(i-1)/tiempoReal,height/2 - height*dataSource.getPath(i-1)/cotaTotal);
ctx.closePath();
ctx.stroke();

ctx.beginPath();
ctx.moveTo(0, height/2 - height*dataSource.getPathRef(0)/cotaTotal);
for(var i=1;i<tiempoReal;i++){
    ctx.lineTo(width*i/tiempoReal,height/2 - height*dataSource.getPathRef(i)/cotaTotal)
}
ctx.moveTo(width*(i-1)/tiempoReal,height/2 - height*dataSource.getPathRef(i-1)/cotaTotal);
ctx.closePath();

ctx.stroke();
ctx.restore();
```

Figura 32: Ejemplo actualizado de gráfica

A continuación se expondrán las diferentes gráficas desarrolladas así como una breve explicación de cada una de ellas:

Grafica de una Referencia:

Esta gráfica sigue una señal y una referencia. En ella, las flechas negras se mueven de acorde a la señal de referencia, subiendo o bajando según varíe esta. La señal del robot, en cambio, está representada por el nivel de llenado del rectángulo interior representado por el rectángulo azul.

El objetivo del usuario es tratar de que el nivel de llenado (la cara superior del rectángulo azul) coincida con el nivel de referencia (la línea que une las flechas negras).

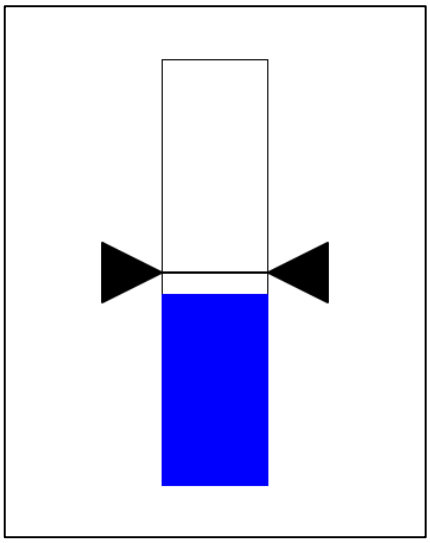


Figura 33: Gráfica de una referencia.

Gráfica Referencia/Tiempo

En esta gráfica, al igual que en la anterior, también se sigue una señal y una referencia. Sin embargo, esta gráfica tiene la peculiaridad que permite ver tanto valores de referencia y señales pasados, como los valores de referencia futuros.

La referencia a seguir está representada por la línea azul, mientras que la señal se representa mediante una línea rojiza. El momento actual está representado por la línea verde vertical, por lo que el círculo al final de la línea rojiza representa el valor de la señal en ese momento, al igual que el punto de intersección de la línea azul con la verde representa el valor de referencia en el mismo momento.

La gráfica se va desplazando con el tiempo hacia la izquierda, por lo que las líneas situadas a la parte izquierda de la línea verde constituyen un histórico de las señales en los últimos segundos de ejecución, mientras que la parte derecha de la línea verde constituye los valores futuros que tendrá la referencia.

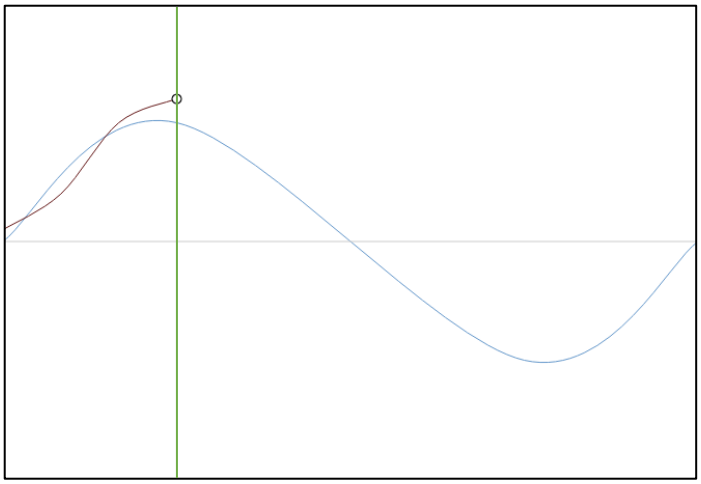


Figura 34: Gráfica referencia tiempo

Grafica de dos Referencias

Esa gráfica tiene como peculiaridad, que sigue dos señales y 2 referencias. Una de las señales se representa a través del eje X de la gráfica, es decir, que refleja el movimiento horizontal del objeto. La otra señal se representa con el eje Y de la gráfica, que refleja su movimiento vertical.

En esta gráfica, el valor de las referencias estará representado por un círculo que se mueve por el plano según se ha especificado más arriba. A su vez, el valor de las señales estará representado por otro círculo más pequeño que seguirá el mismo comportamiento.

El objetivo del usuario es mantener el círculo pequeño dentro del círculo grande.

Figura 35: Grafica de dos Referencias

8.5.5. Feedback

El feedback o realimentación es el medio por el que el usuario puede conocer como está realizando un ejercicio en un momento dado, y cuantificar lo que está haciendo bien y lo que está haciendo mal. Esta información es importante para el usuario puesto que no siempre es fácil saber si se está realizando un ejercicio de forma correcta o no. Por ejemplo, supongamos que el usuario está realizando un ejercicio que utiliza dos señales y dos referencias. En un momento dado, el circulo pequeño no está completamente dentro del circulo grande, ¿significa eso que el usuario está realizando el ejercicio erróneamente?¿o existen ciertos grados de tolerancia en los que se considera que el usuario está dentro de los límites?

Esta información se refleja en el espacio de feedback. Tal vez a mi, como usuario, no me parezca que la señal y la referencia estén muy distantes, pero si en la pantalla me aparece una cara roja triste, me plantearé si lo estoy haciendo correctamente o si debo intentar acercarme más.

Actualización de feedback

Al igual que pasa con la gráfica, el feedback se actualiza con un periodo de actualización T. Cada T milisegundos, se solicitará la puntuación del ejercicio en ese momento, que depende de la diferencia entre la señal del robot y el valor de referencia. Según se necesite, podemos obtener tres tipos de puntuaciones para medir el rendimiento del usuario:

- Puntuación instantánea simple: Se devuelve la diferencia instantánea entre el valor de referencia y la señal del robot.
- Puntuación instantánea doble: Se devuelve el módulo de las diferencias instantáneas entre las referencias y las señales. Es decir:

$$punt = \sqrt{punt_1^2 + punt_2^2}$$

- Puntuación histórica: Se devuelve la suma de las diferencias entre señal y referencia en un cierto espacio de tiempo.

Cada una de estas puntuaciones deberá ser usada cuando sea necesaria, dependiendo de la gráfica que se utilice y del número de señales a medir. No es lógico utilizar la puntuación instantánea doble en un ejercicio en el que tan solo se mida una señal, por ejemplo.

A continuación se describirán los tipos de feedback desarrollados, así como una breve explicación de cada uno de ellos. Destacar que, al igual que pasa con las gráficas, es posible añadir nuevos tipos de feedback sin tener que recompilar la aplicación.

Caras felices/tristes

Este tipo de feedback consiste en una serie de caras tristes, neutras y alegres, las cuales van cambiando según la puntuación obtenida. Tan solo se mostrará una de las caras a la vez. Si la puntuación es baja, es decir, la diferencia entre la referencia y la señal también es baja, se mostrará la cara verde sonriente. En el caso de que la puntuación aumente, y por tanto, la diferencia entre la referencia y la señal, se irá mostrando cada vez una cara más y más triste, hasta llegar a la cara roja de la parte inferior.

Este feedback nos proporciona 5 grados visuales de realimentación.

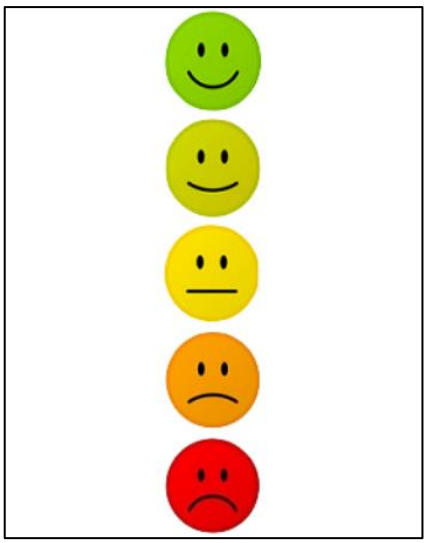


Figura 36: Feedback caritas

Pulgar arriba/abajo

El funcionamiento de este feedback es similar al de las caras, con la diferencia que el número de grados visuales de realimentación se reduce a 2, lo que resulta en un modelo más restrictivo. Tan solo se mostrará o el pulgar hacia arribo o el pulgar hacia abajo. No hay término medio, o está bien o está mal.

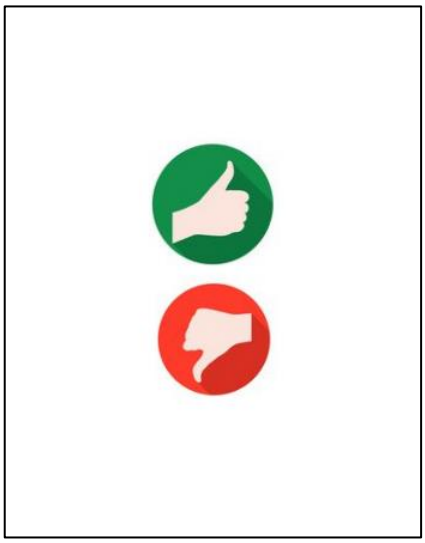


Figura 37: Feedback pulgares

Barra

El último feedback desarrollado consiste en una barra que se vacía o rellena según la puntuación obtenida. Si la puntuación es buena, la barra estará prácticamente llena, mientras que si la puntuación es muy mala, la barra estará vacía. En este caso desaparecen los grados visuales.

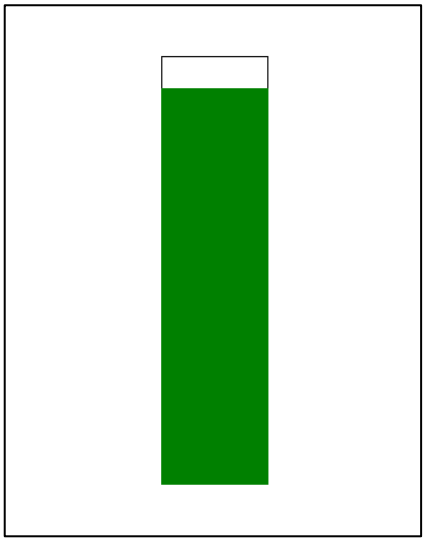


Figura 38: Feedback barra

Además, el color de la barra también se modificará en función del estado de llenado de la barra, llendo del verde fuerte cuando la barra esta llena al rojo cuando la barra esta vacia, pasando por el verde claro, amarillo y naranja entre medias.

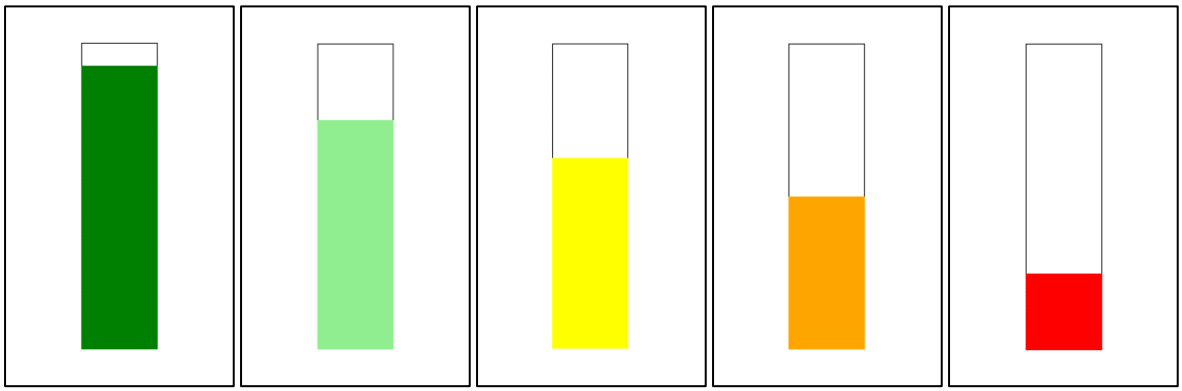


Figura 39: Estados del feedback barra

Nuevos elementos

Todos los elementos gráficos se han desarrollado siguiendo la premisa de que sean lo más visuales posibles, y eligiendo aquellos elementos que son más representativos. Pero podría darse el caso que en un futuro se desearan añadir elementos más específicos, o menos orientados a la simplicidad visual. Por ejemplo, es posible que en un futuro se desee saber cuál es el valor exacto de la referencia o de la señal, y se desea que se muestre por pantalla en todo momento. En este caso, sería necesaria añadir dicho elemento al sistema.

Esta aplicación está preparada para estas eventualidades, ya que es posible añadir nuevos elementos al sistema sin tener que recompilar la aplicación. Esto se debe en gran parte gracias a que la lectura de los ficheros QML se produce en tiempo de ejecución y no en tiempo de compilación.

Para ello, se han creado una serie de métodos genéricos para acceder a las señales y referencias, y también para actualizarlas, a los que deberán llamar estos nuevos elementos creados con el fin de vincularlos a la aplicación.

En el Anexo II situado al final del documento se encuentra explicado con más detalle todo el proceso necesario a seguir para realizar esta integración.

8.6. Desarrollo

8.6.1. Arquitectura

La arquitectura seleccionada para este proyecto corresponde a una arquitectura en 2 capas: una capa de presentación, y una capa de negocio y datos.

La capa de presentación o interfaz gráfica es la que ve el usuario e interactúa con él. Su función consiste en comunicar al usuario y capturar información en un mínimo de proceso.

En este proyecto, esta capa corresponde a los diversos archivos '.qml' que definen las diferentes pantallas de nuestra aplicación.

La capa de negocio y datos es la que se encarga tanto de acceder a los datos necesarios (ya sea en fichero o como topics en ROS), así como de procesar estos datos.

En este proyecto, esta capa corresponde a los ficheros .cpp del proyecto, especialmente el fichero datasource.cpp, que es el que se encarga de comunicarse directamente con la interfaz gráfica.

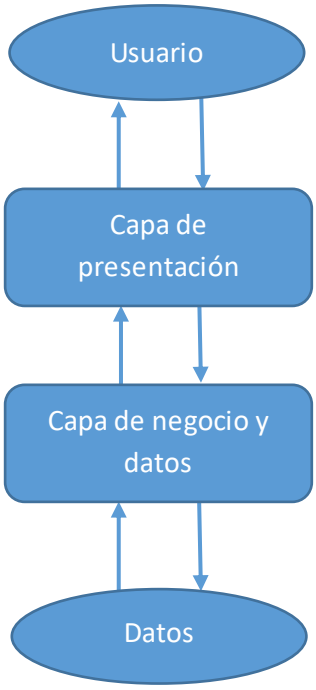


Figura 40: Arquitectura en 2 capas

Es necesario comentar que, para proyectos más grandes, lo ideal sería dividir esta segunda capa en 2, distinguiendo entre la capa de negocio, que se encargaría de los procesamientos, y la capa de datos, que se encargaría del acceso a los datos, permitiendo una mayor modularización del proyecto.

8.6.2. Ficheros ejercicios

Los ejercicios a realizar se guardarán dentro de una carpeta en el sistema de archivos, configurable a partir de ajustes. Para que se puedan cargar correctamente, estos ficheros deben seguir un formato específico.

Este formato sigue la siguiente especificación:

- El fichero estará formado por N líneas, donde cada línea representa un punto de referencia en el tiempo.
- Cada línea estará formada por 2 o 3 valores de tipo *double*, separados por tabulador, en los que el primer valor corresponde a la referencia de tiempo, mientras que el segundo y tercer valor corresponden a las referencias de las señales en el tiempo indicado. Es decir, si se trata de una gráfica que tan solo utiliza una referencia, la línea tendrá 2 valores, mientras que, si la gráfica utiliza 2 referencias, la línea deberá tener 3 valores separados por tabuladores.
- El valor de la referencia tiempo de la primera línea(es decir, el primer valor) deberá tener el valor 0, para que corresponda con el inicio del ejercicio.
- Los valores de las referencias correspondientes al tiempo deberán estar ordenados de forma estrictamente creciente. Es decir, el valor tiempo de la línea N-1 será estrictamente menor al valor tiempo de la línea N.

Si el fichero seleccionado como ejercicio cumple este formato, el fichero se cargará correctamente en memoria, interpolando linealmente entre cada separación de tiempos según la frecuencia de actualización de la gráfica, para poder representar gráficamente de forma correcta el ejercicio.

En el caso que el fichero no cumpla el formato correcto, se mostrará un mensaje por pantalla indicándoselo al usuario.

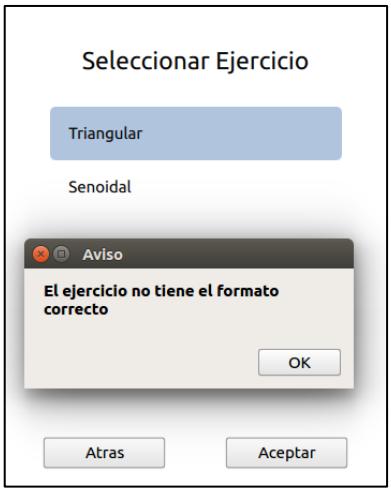


Figura 41: Aviso Formato fichero

A continuación se muestran 2 ejemplos de ficheros con formato correcto junto con su interpretación:

El primer ejemplo corresponde a un ejercicio con tan solo una señal de referencia, donde el fichero contendría las siguientes líneas:

0	0
2	1
6	-1
10	1
14	-1
18	1
22	-1
26	1
28	0

Figura 42: Ejemplo 1 formato fichero ejercicio

El ejercicio resultante para este ejemplo seria el que se muestra en la Figura X.:

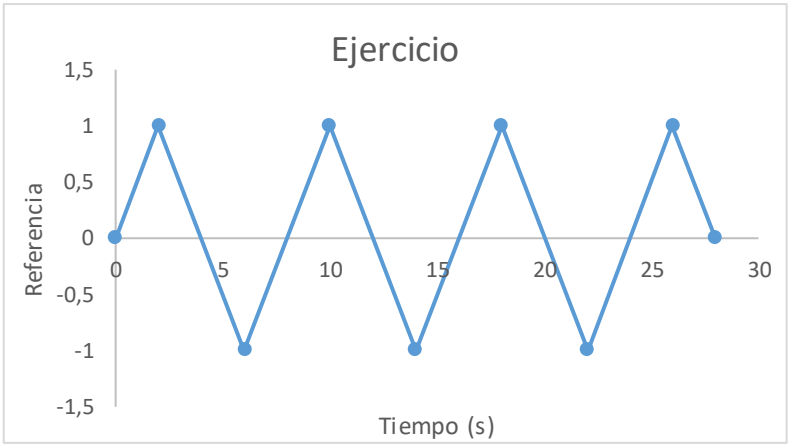


Figura 43: Ejemplo 1 formato fichero ejercicio

El segundo ejemplo corresponde a un ejercicio con dos señales de referencia:

0	1	1
5	1	-1
10	-1	-1
15	-1	1
20	1	1

Figura 44: Ejemplo 2 formato fichero ejercicio

Las referencias para cada señal en este ejemplo serían las siguientes:

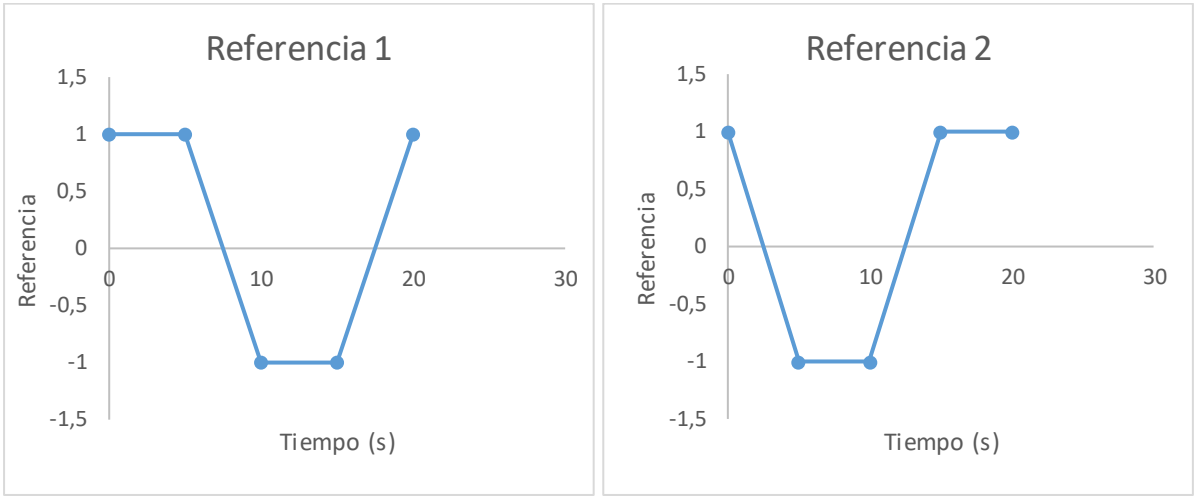


Figura 45 y 46: Ejemplo 2 formato fichero ejercicio

Que correspondería a la siguiente trayectoria de forma cuadrada en una gráfica referencia1 / referencia2:

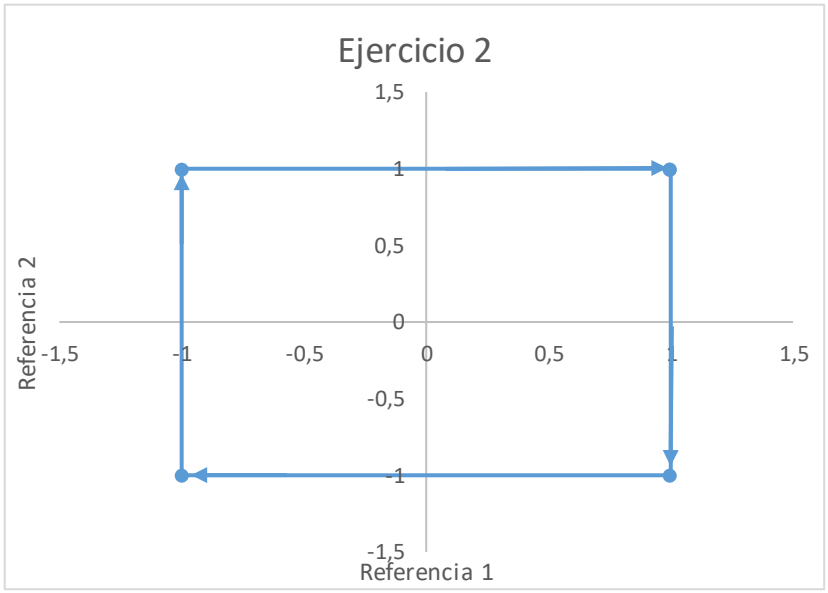


Figura 47: Trayectoria ejemplo ejercicio 2

Estos ejemplos son ejemplos sencillos con pocos puntos de referencia, pero lo recomendable seria añadir más puntos de referencias en tiempos intermedios para suavizar los cambios en las velocidades, como por ejemplo, el fichero usado más adelante para las pruebas unitarias que representa una señal senoidal.

Este diseño de pares tiempo-referencias es más sencillo para entender por el usuario comparado con el formato de los ficheros de resultados explicado más adelante, que tienen una línea para cada actualización de la gráfica. Esto permite crear ejercicios de forma manual y sin preocuparse por la frecuencia de actualización de la gráfica.

8.6.3. Ficheros resultados

Al finalizar un ejercicio, el resultado del ejercicio se guarda en un fichero .dat en la carpeta de resultados.

Estos ficheros también tienen un formato específico, que se define de la siguiente manera:

- El fichero estará formado por N líneas, donde cada línea representa una lectura de las señales en el tiempo. El número de líneas coincidirá con el cociente entre la duración del ejercicio y la frecuencia de actualización de la gráfica. Es decir, por cada actualización de la gráfica, se escribirá una línea en el fichero de resultados.
- Cada línea estará formada por 2 o 3 valores de tipo *double*, separados por tabulador, en los que el primer valor corresponde al valor del tiempo en un momento dado, mientras que el segundo y tercer valor corresponden a los valores de las señales para ese momento dado. Al igual que en los ficheros de ejercicios, si se trata de una gráfica que tan solo utiliza una referencia, la línea tendrá 2 valores, mientras que si la gráfica utiliza 2 referencias, la línea deberá tener 3 valores separados por tabuladores.
- Los valores correspondientes al tiempo estarán ordenados de forma estrictamente creciente.
- La diferencia entre el valor de tiempo de la línea K y el valor del tiempo de la línea K+1 será igual a la frecuencia de actualización de la gráfica.
- El valor de la referencia tiempo de la primera línea(es decir, el primer valor) deberá tener el valor 0, que corresponderá con el inicio del ejercicio.

Este formato permite la exportación de los datos a otros softwares, como por ejemplo Matlab para su análisis.

Además, el nombre del fichero generado también seguirá el siguiente formato:

<año_ejecución>_<mes_ejecución>_<dia_ejecución>_<nombre_ejercicio>.dat

Lo que permite identificar fácilmente las diversas ejecuciones de ejercicios.

8.6.4. Ficheros interfaces

Por último, las interfaces predefinidas también se almacenen en ficheros, y tienen un formato específico:

- El fichero deberá tener o 3 o 4 líneas, con las especificaciones que se nombran a continuación.
- La primera línea específica la gráfica representada, por lo que deberá contener el nombre exacto del fichero .qml que contiene la gráfica deseada.
- La segunda línea deberá contener el nombre exacto del fichero .qml que contiene el tipo de feedback deseado.
- La tercera y cuarta línea especifican las señales utilizadas. Si la interfaz deseada tan solo obtiene datos de una señal, entonces la cuarta línea no deberá estar informada. Contiene el nombre exacto del topic o de los topics a los que desea suscribirse, es decir, que señales hay que representar gráficamente.
- Por último, el propio nombre del fichero, excluyendo la extensión .dat, especificará en nombre de la interfaz.

Por ejemplo, el siguiente fichero:

```
GraficaTiempo.qml
Faces.qml
/chatter
```

Figura 48: Fichero interfaz Prueba.dat

Define una interfaz que contiene la gráfica “GraficaTiempo.qml”, el feedback “Faces.qml” y estará representando la señal /chatter. Su interfaz final tendrá el siguiente aspecto:

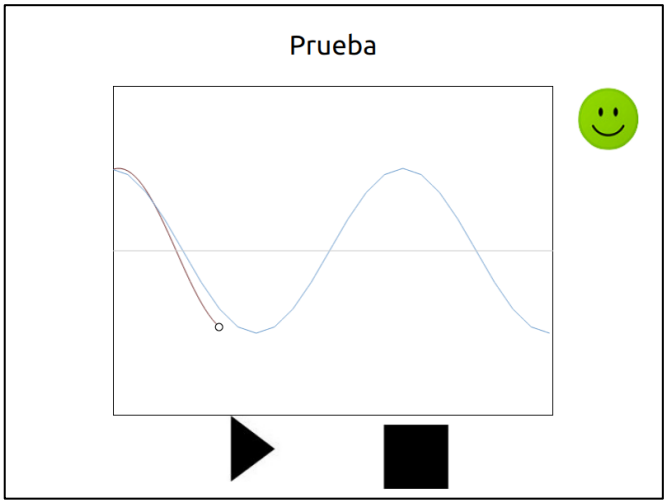


Figura 49: Ejemplo interfaz predefinida

8.6.5. Lista de ficheros disponibles

Para obtener tanto los ejercicios como las interfaces disponibles, es necesario leer la carpeta contenedora de dichos ficheros. En la actualidad, C++ no incorpora ningún mecanismo para actuar sobre un sistema de ficheros cualquiera. Por tanto, es necesario utilizar un mecanismo propio de Linux que nos permita obtener dicha información. Para ello incluiremos la siguiente librería:

```
#include <dirent.h>
El código utilizado para obtener una lista de ficheros en un directorio
es el siguiente:
  dir = opendir(str.c_str());
  while((ent = readdir (dir))!=NULL){
    if((strcmp(ent->d_name,".")!=0)&&(strcmp(ent->d_name,"..")!=0)){
      nombreEjercicios<<ent->d_name;
      res++;
    }
  }
  return res;
```

Cabe destacar, que en la lista proporcionada por la librería 'dirent', aparecen también las entradas "." y "..", correspondientes a la propia dirección del directorio y a la dirección de la carpeta contenedora, por lo que será necesario filtrarlos.

8.6.6. Lectura de topics disponibles

Para poder seleccionar entre los diferentes topics disponibles en el master en ese momento, es necesario obtener antes dichos topics. Para ello, se utilizará el siguiente fragménto de código, el cual itera entre los topics del master y los guarda en una lista:

```
QStringList DataSource::getTopics(){
  QStringList items;
  ros::master::V_TopicInfo master_topics;
  ros::master::getTopics(master_topics);

  for (ros::master::V_TopicInfo::iterator it = master_topics.begin()
; it != master_topics.end(); it++) {
      const ros::master::TopicInfo& info = *it;
      QString aux = info.name.c_str();
      items << aux;
  }
  return items;
}
```

8.6.7. Subscripción a topics

El método para subscribirse a un topic está definido por el siguiente formato:

```
Ros::NodeHandle n;
n.subscribe(<nombre_topic>, <buffer>, <callback>)
```

Donde <nombre_topic> es el nombre del tópic a utilizar, <buffer> es el número máximo de mensajes del subscriber que se guardaran en memoria mientras no se llame al callback, y <callback>

```
void DataSource::setSubscriber1(QString str){
  ros::NodeHandle n;
  sub1 = n.subscribe(str.toStdString(), 0, &DataSource::sub1Callback,
this);
}

void DataSource::sub1Callback(const std_msgs::String::ConstPtr& msg)
{
  setData1(msg->data.c_str());
}
```

8.6.8. Funcionamiento intrínseco interfaces

El funcionamiento intrínseco de las interfaces consiste en una interfaz fija llamada main.qml que solamente contiene en su interior un loader de las diferentes pantallas, las cuales van cambiando conforme se va navegando por la aplicación.

El motivo de este diseño es doble: uno visual y otro funcional.

Por una parte, se evita estar cerrando y abriendo pantallas nuevas, ya que cada vez que se abre una nueva pantalla, ésta se sitúa en un espacio diferente de la pantalla, lo que provoca un efecto visual muy molesto al cambiar de pantalla, debido a que parece que la aplicación se va moviendo por la pantalla conforme se navega por ella.

Por otro lado, permite mantener de forma sencilla el vínculo entre las interfaces y la clase datasource, evitando tener que vincularlos cada vez que se cambie de pantalla.

9. CONCLUSIONES Y TRABAJOS FUTUROS

Se ha desarrollado una aplicación para ROS que cumple con todas las funcionalidades especificadas.

Esta aplicación, permitirá a los pacientes que usen el robot 3PRS en ejercicios de rehabilitación, la posibilidad de observar el estado del ejercicio en todo momento.

Se han planteado diversas funcionalidades que, pese a no encontrarse entre los objetivos finales del trabajo, conformarían unos excelentes complementos a la aplicación desarrollada. Entre estas nuevas funcionalidades que supondrían un posible trabajo futuro, cabe desarrollar las siguientes:

En primer lugar, sería muy interesante disponer de una forma de visualización de los resultados de un ejercicio una vez terminado el ejercicio. De esta manera, no sería necesario acceder a herramientas externas para poder analizar dichos resultados.

También sería interesante añadir un histórico de pacientes, permitiendo controlar que ejercicios ha realizado cada paciente, y sus resultados, lo que permitiría seguir el proceso de rehabilitación de un paciente de manera más eficiente.

Por último, cabe destacar la utilidad que supondría la integración en la nube de todos los datos utilizados en la aplicación, desde la obtención de los diferentes ejercicios y configuraciones de gráficas, hasta el guardado de los archivos resultados. Esto permitiría la centralización del mantenimiento de la aplicación, y facilitaría el acceso a los ficheros de resultados.

ANEXO I: CMAKELISTS.TXT

```
##############################################################################
#CMake / Catkin
##############################################################################

#La primera parte simplemente configura el paquete ROS, como se espera en un
#CMakeLists.txt de un proyecto normal
cmake_minimum_required(VERSION 2.8.0)
project(gui)
find_package(catkin REQUIRED COMPONENTS qt_build roscpp)
set(QML_IMPORT_PATH "${QML_IMPORT_PATH};${CATKIN_GLOBAL_LIB_DESTINATION}" )
set(QML_IMPORT_PATH2 "${QML_IMPORT_PATH};${CATKIN_GLOBAL_LIB_DESTINATION}" )
include_directories(${catkin_INCLUDE_DIRS})
catkin_package()

##############################################################################
# Qt Environment
##############################################################################

#Indicamos a catkin que incluya el paquete Qt5, y le decimos que necesitaremos
#los components Core, Gui, Qml, Quick y Widgets
find_package(Qt5 COMPONENTS Core Gui Qml Quick Widgets REQUIRED)

##############################################################################
# Sections
##############################################################################

#Indicamos a cmake donde se encuentran tanto los recursos utilizados, como las
#cabezeras de los archivos .cpp para poder precompilarlos.
file(GLOB QT_RESOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} resources/*.qrc)
file(GLOB_RECURSE QT_MOC RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} FOLLOW_SYMLINKS
include/gui/*.hpp)
QT5_ADD_RESOURCES(QT_RESOURCES_CPP ${QT_RESOURCES})
QT5_WRAP_CPP(QT_MOC_HPP ${QT_MOC})

##############################################################################
# Sources
##############################################################################

#Indicamos a cmake donde se encuentran los archivos de código fuente.
file(GLOB_RECURSE QT_SOURCES RELATIVE ${CMAKE_CURRENT_SOURCE_DIR} FOLLOW_SYMLINKS
src/*.cpp)

##############################################################################
# Binaries
##############################################################################

#Finalmente, creamos un nodo ROS (ejecutable) que vincula los archivos de código
#fuente con las cabezeras y recursos precompilados.
add_executable(gui ${QT_SOURCES} ${QT_RESOURCES_CPP} ${QT_FORMS_HPP}
${QT_MOC_HPP})
qt5_use_modules(gui Quick Core Widgets)
target_link_libraries(gui ${QT_LIBRARIES} ${catkin_LIBRARIES})
target_include_directories(gui PUBLIC include)
install(TARGETS gui RUNTIME DESTINATION ${CATKIN_PACKAGE_BIN_DESTINATION})
```

ANEXO II: CREACIÓN DE NUEVOS ELEMENTOS

Al añadir nuevos elementos, distinguiremos entre nuevos elementos de tipo gráfica, y nuevos elementos de tipo feedback.

Nueva gráfica

Para añadir un nuevo elemento de tipo gráfica, es necesario crear dos archivos QML, uno estático y otro dinámico. Estos archivos se situarán en la carpeta resources del directorio del proyecto, junto a los otros archivos QML.

El primer archivo QML servirá para mostrarse en la pantalla de Nueva Interfaz como un ejemplo de dicha gráfica. Este archivo representará una escena estática del elemento en sí.

Para que dicho elemento aparezca en la pantalla **Nueva Interfaz** primero hay que añadir una nueva entrada en la lista del ComboBox utilizado para seleccionar el tipo de gráfica en el archivo NuevaInterfaz.qml.

```
ComboBox{
    anchors.verticalCenter: parent.verticalCenter
    id:tipoGrafica
    width: 200
    anchors.horizontalCenter: parent.horizontalCenter
    model: ListModel{
        id:itemsGrafica
        ListElement{ text: "--Seleccionar--"}
        ListElement{ text: "Referencia/Tiempo"}
        ListElement{ text: "Referencia/Referencia"}
        ListElement{ text: "Una Referencia"}
    }

    onCurrentTextChanged: cambioGrafica(itemsGrafica.get(currentIndex).text)
}
```

Figura 50: Combo Box gráfica

A continuación, se añadirá una nueva entrada a la cadena if..else.. de la función cambioGrafica(text)del mismo archivo (NuevaInterfaz.qml). La estructura de esta nueva entrada es la siguiente:

```
if (text == <Texto de lista>){
     loaderGrafica.source = "<Archivo QML estático>"
     opcionSeñal1.visible = true
     opciónSeñal2.visible=<true si utiliza 2 señales / false en caso
                          contrario>
     dataSource.setGrafica("Archivo QML dinámico")
}
```

```
function cambioGrafica(text){
    if (text=="Referencia/Tiempo"){
        loaderGrafica.source = "GraficaTiempoEstatica.qml"
        opcionSeñal1.visible=true
        opcionSeñal2.visible=false
        dataSource.setGrafica("GraficaTiempo.qml")
    }else if (text=="Referencia/Referencia") {
        loaderGrafica.source = "Grafica2DimensionesEstatica.qml"
        opcionSeñal1.visible=true
        opcionSeñal2.visible=true
        dataSource.setGrafica("Grafica2Dimensiones.qml")
    }else if (text=="Una Referencia") {
        loaderGrafica.source = "Grafica1DimensionEstatico.qml"
        opcionSeñal1.visible=true
        opcionSeñal2.visible=false
        dataSource.setGrafica("Grafica1Dimension.qml")
    }else{
        loaderGrafica.source = ""
        opcionSeñal1.visible=false
        opcionSeñal2.visible=false
    }
}
```

Figura 51: Cambio Gráfica

Con este código no solo cambiamos la imagen estática que se muestra, sino que también fijamos el nombre del archivo QML que se cargará como la gráfica de la interfaz. Así, cuando el sistema vaya a crear la interfaz a partir de los datos de entrada, simplemente cargará el archivo QML especificado con el método setGrafica(gráfica).

Una vez realizada esta parte, es necesario vincular el archivo QML dinámico a los valores de señal y referencia. Para ello, se incluirá un objeto de tipo *Timer* en el archivo QML, desde donde ejecutaremos tanto la actualización de los datos con el método actualizaPath() como el redibujado de la gráfica en el caso que esté representado por un objeto de tipo Canvas.

Este objeto Timer deberá tener como atributo 'interval' una invocación al método getFrecuencia(), que nos devuelve la frecuencia de actualización de la gráfica.

```
Timer{
    id:timerGrafica
    interval:dataSource.getFrecuencia()
    running:true
    repeat:true
    onTriggered: {
        dataSource.actualizaPath()
        canvas.requestPaint()
    }
}
```

Figura 52: Timer

Para obtener los valores de referencia en un momento dado, se utilizan los métodos getXRef() y getYRef(), mientras que para obtener los valores de las señales se utilizan los métodos getXReal() y getYReal(). El añadido X e Y significa que se refieren a la primera y segunda señal a medir. En el caso de que tan solo se mida una señal no se deberán usar los métodos getYReal() y getYRef(), ya que siempre devolverán 0.

También tenemos los valores maxX, maxY, minX, minY, disponibles como variables por el archivo Interfaz.qml, que representan los valores máximo y mínimo en la gráfica.

En resumen, los métodos disponibles para la creación de gráficas son:

actualizaPath(): provoca que se actualicen los valores de señal y referencia internamente. Es necesario llamar a otros métodos para acceder a estos valores.

getFrecuencia(): devuelve el parámetro de frecuencia de actualización de la gráfica.

getXReal(): devuelve el valor de la primera señal a medir.

getYReal():devuelve el valor de la segunda señal a medir.

getXRef():devuelve el valor de la primera referencia.

getYRef(): devuelve el valor de la segunda referencia.

Con todas estas herramientas se es capaz de crear un nuevo elemento de tipo gráfica que obtenga las señales y las referencias en tiempo real, y todo ello sin necesidad de recompilar la aplicación.

Nuevo feedback

Para añadir un Nuevo tipo de feedback, necesitaremos de nuevo dos archivos qml, uno estático y otro dinámico.

Por la parte que respecta a las modificaciones en NuevaInterfaz.qml, los pasos a seguir son prácticamente los mismos, excepto en el detalle en que se debe modificar la lista correspondiente al ComboBox de los feedbacks y no de las gráficas, y la función a modificar será cambiaFeedback(text). Por el resto será exactamente igual.

La única diferencia la encontramos en los métodos disponibles para actualizar el feedback. Tendremos a nuestra disposición tres posibles métodos, todos ellos relacionados con las puntuaciones:

- getPunctSimple():Devuelve la diferencia instantánea entre el valor de referencia y la señal del robot.
- getPunctDoble(): Devuelve el módulo de las diferencias instantáneas entre las referencias y las señales.

- getPunctHist(): Devuelve la suma de las diferencias entre señal y referencia en un cierto espacio de tiempo.

Con estos métodos, junto con los valores de máximo y minimo de las señales, maxX, maxY, minX, minY, el usuario tiene la suficiente información para crea el elemento de tipo feedback.

BIBLIOGRAFÍA

[1] https://mebiomec.ai2.upv.es/content/principal

[2] Robot Institute of America, 1979

[3] http://platea.pntic.mec.es/vgonzale/cyr_0204/ctrl_rob/robotica/industrial.htm

[4] http://www.aeratp.com/aer-atp/robotica-industrial-y-de-servicio/

[5] Robots Paralelos: Maquinas con un Pasado para una Robótica del Futuro, Rafael Aracil, Roque J. Saltarén, José Mª Sabater, Oscar Reinoso

[6] http://dle.rae.es/?id=9GCh2qU

[7] http://www.sitenordeste.com/mecanica/maquinas_mecanismos.htm

[8] https://ifr.org/img/office/Industrial_Robots_2016_Chapter_1_2.pdf

[9] https://nccastaff.bournemouth.ac.uk/jmacey/ASE/

[10] http://doc.qt.io/

[11] Implementación basada en OROCOS de controladores dinámicos para un robot paralelo. Jose Ignacio Cazalilla Morenas, 2012

[12] Desarrollo de una aplicación para el guiado automático de un vehículo eléctrico, Claudia Anais González Moreno

PRESUPUESTO

1. Introducción al presupuesto

A continuación se presenta el presupuesto detallado de este proyecto. Este presupuesto consta de 3 capítulos nombrados a continuación:

- Capítulo 1: Licencias de software
- Capítulo 2: Equipo informático
- Capítulo 3: Mano de obra

2. Capítulo 1: Licencias de software

Para llevar a cabo el trabajo se han utilizado diferentes programas y sistemas informáticos. En concreto, es necesario remarcar el sistema operativo utilizado Ubuntu, el framework ROS, y el programa informático QTCreator, junto con las librerías Qt 5.9.3:

Software	Unidades	Precio Unitario (€/ud)	Coste
Ubuntu 16.04 LTS	1	Licencia gratuita	0,00€
ROS Kinetic Kame	1	Licencia gratuita	0,00€
Qt 9.5.3 Opensource	1	Licencia gratuita	0,00€
Qt-Creator Opensource	1	Licencia gratuita	0,00€
		Total	0,00€

Tabla 1: Licencia de software

3. Capítulo 2: Equipo informático

El ordenador utilizado tiene las siguientes características:

- Sistema operátivo Windows 10 Pro 64-bit
- Procesador Intel Core i5 4670K 3.40GHz
- Memoria RAM 8.00GB DDR3 1199MHz
- Placa base ASUSTeK COMPUTER INC. Z97-K (SOCKET 1150)
- Gráfica 2047MB NVIDIA GeForce GTX 660 (Gigabyte)
- Pantalla BenQ GL2250H
- Almacenamiento 119GB SanDisk SDSSDHP128G

Se considera un periodo de amortización de 3 años, con una utilización media de 8 horas diarias durante 240 días al año (1920 horas al año).

Equipo	Uds.	Horas/año	Horas de trabajo	Precio unitario (€/ud)	Amortización	Coste (€)
PC	1	1920	300	1.000	5,21%	52,08€
					Total	52,08€

Tabla 2: Equipo informático

4. Capítulo 3: Mano de obra

Tarea	Horas	Precio Unitario	Coste
Diseño y análisis	15h	30€	450,00€
Prototipado	30h	30€	900,00€
Integración QT/ROS	45h	30€	1350,00€
Desarrollo	180h	30€	5.400,00€
Pruebas unitarias	30h	30€	900,00€
		Total	9.000,00€

Tabla 3: Mano de obra

5. Coste total

Concepto	**Coste (€)**
Licencia de software	0,00€
Equipo informático	52,08€
Mano de obra	9.000,00€
Subtotal	9.052,08€
Beneficio industrial (6%)	543,12€
I.V.A. (21%)	1900,94€
TOTAL	11496,14€

Tabla 4: Coste total