Universidad de Alcalá
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

Control de la mano robótica BarrettHand BH8-262 en entorno ROS

ESCUELA POLITECNICA SUPERIOR

Autor: Alejandro Gómez Rubio

Tutor: D. Rafael Barea Navarro

2014

UNIVERSIDAD DE ALCALÁ
Escuela Politécnica Superior

GRADO EN INGENIERÍA EN ELECTRÓNICA Y AUTOMÁTICA INDUSTRIAL

Trabajo Fin de Grado

“Control de la mano robótica BarrettHand BH8-262 en entorno ROS”

Autor: Alejandro Gómez Rubio

Director: D. Rafael Barea Navarro

TRIBUNAL:

Presidente: Dª. Elena López Guillén

Vocal 1º: D. Luis Miguel Bergasa Pascual

Vocal 2º: D. Rafael Barea Navarro

CALIFICACIÓN: ....

FECHA: ....

Agradecimientos

A toda mi familia, en especial a mis padres Alejandro e Inés, por todo su apoyo.

A mis compañeros de clase con los que he pasado estos últimos años.

A mi tutor, Rafael Barea, por su ayuda y buenos consejos.

ÍNDICE GENERAL

ÍNDICE DE FIGURAS ... XI
ÍNDICE DE TABLAS ... XIII
GLOSARIO ... XV
RESUMEN ... XVII
ABSTRACT ... XIX
RESUMEN EXTENDIDO ... XXI
1. INTRODUCCIÓN ... 3
1.1 Planteamiento ... 3
1.2 Estado del arte ... 3
1.2.1 Grippers industriales ... 3
1.2.2 Manos robóticas antropomórficas ... 4
1.2.3 Paquetes de control de manos robóticas en ROS ... 5
2. ARQUITECTURA DEL SISTEMA ... 9
2.1 Introducción ... 9
2.2 ROS ... 9
2.2.1 Introducción ... 9
2.2.2 Paquetes ... 10
2.2.3 Nodos ... 11
2.2.4 Métodos de comunicación entre nodos ... 12
2.3 BarrettHand ... 15
2.3.1 Introducción ... 15
2.3.2 Estructura mecánica ... 16
2.3.3 Electrónica de control ... 18
2.3.4 Librería de control ... 19
2.4 Brazo robot industrial IRB 120 ... 23
2.5 Kinect ... 24
3. PAQUETE DE CONTROL DE LA BARRETTHAND EN ROS ... 27
3.1 Introducción ... 27
3.2 Nodo bhand_server ... 27
3.3 Nodo bhand_client ... 30
4. APLICACIÓN DE CAPTURA DE UNA PELOTA EN MOVIMIENTO ... 33
4.1 Introducción ... 33
4.2 Arquitectura general de la aplicación ... 33
4.3 Estudio del movimiento pendular ... 34
4.3.1 Movimiento armónico simple ... 35
4.3.2 Movimiento armónico amortiguado ... 36
4.4 Péndulo utilizado en la aplicación ... 36
4.4.1 Dimensiones ... 36
4.4.2 Parámetros del péndulo real ... 37
4.4.3 Modelo del péndulo real en simulink ... 38
4.5 Limitaciones del sistema ... 42
4.5.1 Brazo robot IRB 120 y controlador IRC5 ... 43
4.5.2 Comunicación Ethernet ... 43
4.5.3 PC con ROS ... 44
4.5.4 Resolución Kinect ... 46

4.5.5 Retardo Kinect 46
4.5.6 Retardo general del sistema 47
4.6 Paquete vision 47
4.6.1 Introducción 47
4.6.2 Nodo vision 47
4.6.3 Nodo model 55
5. APLICACIÓN DE SEGUIMIENTO 59
5.1 Introducción 59
5.2 Arquitectura general de la aplicación 59
5.3 Paquete tracking 59
5.3.1 Introducción 59
5.3.2 Nodo tracking 60
6. APLICACIÓN DE CLASIFICACIÓN DE OBJETOS 67
6.1 Introducción 67
6.2 Sistema de reconocimiento de objetos y estimación de posición 67
6.3 Control de la BarrettHand BH8-262 68
6.3.1 Servicio set_fingers 69
6.3.2 Servicio set_fingers_speed 70
6.4 Control del IRB 120 70
6.4.1 Nodo 1: object_attach 71
6.4.2 Nodo 2: attach 71
7. INTRODUCCIÓN A GRASPIT! 77
8. RESULTADOS 85
8.1 Introducción 85
8.2 Tasa de aciertos y fallos en la aplicación 85
8.3 Errores 85
9. CONCLUSIONES Y TRABAJO FUTURO 91
9.1 Conclusiones 91
9.2 Trabajo futuro 91
10. DIAGRAMAS 95
10.1 Diagrama de la arquitectura general del sistema 95
10.2 Aplicación de captura de una pelota en movimiento 95
10.3 Aplicación de seguimiento 97
11. PLIEGO DE CONDICIONES 101
11.1 Requisitos Hardware 101
11.2 Requisitos Software 101
12. PRESUPUESTO 105
12.1 Presupuesto de ejecución material 105
12.2 Importe total 105
13. MANUAL DE USUARIO 109
13.1 Introducción 109
13.2 Configuración de la BarrettHand 109
13.3 Instalación de las aplicaciones 109
13.3.1 Instalación ROS Hydro 109
13.3.2 Instalación del paquete ROS openni 110
13.3.3 Instalación del paquete ROS bhand 110
13.3.4 Instalación del paquete ROS vision 113
13.3.5 Instalación del paquete ROS tracking 113

13.3.6 Instalación de GraspIt! 113
13.4 Uso de las aplicaciones 114
13.4.1 Uso del paquete bhand.. 114
13.4.2 Uso del socket en IRC5.. 116
13.4.3 Ejecución Aplicación de captura de una pelota en movimiento....... 116
13.4.4 Ejecución Aplicación de seguimiento.. 117
13.4.5 Ejecución Aplicación de clasificación de objetos.............................. 117

ÍNDICE DE FIGURAS

FIGURA 1.1: SCHUNK GRIPPER 3
FIGURA 1.2: CONFIGURACIONES SCHUNK GRIPPER....... 4
FIGURA 1.3: ROBOTIQ GRIPPER 4
FIGURA 1.4: CONFIGURACIONES ROBOTIQ GRIPPER....... 4
FIGURA 1.5: SHADOW DEXTEROUS HAND....... 5
FIGURA 1.6: ELU-2 HAND 5
FIGURA 2.1: ARQUITECTURA GENERAL DEL SISTEMA....... 9
FIGURA 2.2: ESTRUCTURA PAQUETES ROS 10
FIGURA 2.3: ESTRUCTURA NODOS....... 12
FIGURA 2.4: COMUNICACIÓN POR TOPIC 12
FIGURA 2.5: COMUNICACIÓN POR SERVICIOS....... 14
FIGURA 2.6: BARRETTHAND BH8-262 15
FIGURA 2.7: FUENTE DE ALIMENTACIÓN 16
FIGURA 2.8: ARTICULACIONES BARRETTHAND 16
FIGURA 2.9: RANGO ARTICULACIONES DE LOS DEDOS 17
FIGURA 2.10: RANGO GIRO ENTORNO LA PALMA....... 17
FIGURA 2.11: OPERACIÓN TORQUESWITCH 17
FIGURA 2.12: ACOPLAMIENTO BARRETTHAND 18
FIGURA 2.13: ELECTRÓNICA DE CONTROL 18
FIGURA 2.14: IRB 120....... 23
FIGURA 2.15: IRC5 Y FLEXPENDANT 23
FIGURA 2.16: KINECT 24
FIGURA 3.1: SERVICIOS BHAND_SERVER 27
FIGURA 3.2: SECUENCIA CIERRE DE DEDOS 28
FIGURA 4.1: ARQUITECTURA APLICACIÓN DE CAPTURA 33
FIGURA 4.2: APLICACIÓN DE CAPTURA. COMPONENTES 34
FIGURA 4.3: PÉNDULO 35
FIGURA 4.4: MOVIMIENTO AMORTIGUADO....... 36
FIGURA 4.5: PÉNDULO REAL....... 37
FIGURA 4.6: PÉNDULO. POSICIÓN EJE X-TIEMPO....... 37
FIGURA 4.7: PÉNDULO. POSICIÓN EJE Y-TIEMPO....... 38
FIGURA 4.8: SYSTEM IDENTIFICATION TOOL 39
FIGURA 4.9: SYSTEM IDENTIFICATION TOOL. IMPORTAR DATOS....... 39
FIGURA 4.10: SYSTEM IDENTIFICATION TOOL. PARÁMETROS DEL MODELO 40
FIGURA 4.11: DIAGRAMA DE BLOQUES SIMULINK....... 40
FIGURA 4.12: MODELO. POSICIÓN EJE X E Y-TIEMPO 41
FIGURA 4.13: MODELO. VELOCIDADES EJE X E Y....... 41
FIGURA 4.14: MODELO. ACELERACIONES EJE X E Y....... 42
FIGURA 4.15: SECCIÓN CRÍTICA. PÉRDIDA DE INFORMACIÓN....... 44
FIGURA 4.16: SECCIÓN CRÍTICA. RETARDOS ACUMULABLES 45
FIGURA 4.17: RETARDO KINECT....... 46
FIGURA 4.18: APLICACIÓN DE CAPTURA. DIAGRAMA DE FLUJO....... 48
FIGURA 4.19: APLICACIÓN DE CAPTURA. SECUENCIA....... 48
FIGURA 4.20: PAQUETES SOCKET. 50
FIGURA 4.21: ESPACIO DE COLOR HSV 50
FIGURA 4.22: SEGMENTACIÓN. DIAGRAMA DE FLUJO 51
FIGURA 4.23: SEGMENTACIÓN. EJEMPLO 51
FIGURA 4.24: MOMENTOS DE ÁREA. EJEMPLO 52
FIGURA 4.25: SECUENCIA CAPTURA DE PELOTA 53
FIGURA 4.26: PREDICCIÓN....... 55
FIGURA 5.1: ARQUITECTURA APLICACIÓN SEGUIMIENTO 59
FIGURA 5.2: APLICACIÓN SEGUIMIENTO. DIAGRAMA DE FLUJO....... 60
FIGURA 5.3: APLICACIÓN SEGUIMIENTO. SECUENCIA....... 60
FIGURA 5.4: PREDICTOR DE POSICIÓN 61

Figura 5.5: Sistemas de coordenadas IRB 120 y Kinect 62
Figura 6.1: Articulaciones BarrettHand 68
Figura 6.2: Estructura servicios 69
Figura 6.3: Modelo IRB 120 con BarrettHand 71
Figura 6.4: Aplicación de clasificación. Situación inicial y final 72
Figura 6.5: Robot posicionando una caja 72
Figura 6.6: Robot posicionando una lata. 73
Figura 6.7: Robot posicionando una pelota. 73
Figura 7.1: Entorno de GraspIt! 77
Figura 7.2: Fichero de mundo .xml 78
Figura 7.3: Definición de posición y orientación en GraspIt! 78
Figura 7.4: Campos de articulaciones de la BarrettHand en GraspIt! 79
Figura 7.5: Eigengrasp en diferentes manos robóticas 80
Figura 7.6: Menú de Grasp Planner 80
Figura 7.7: Factor de calidad 81
Figura 7.8: Factor de calidad de agarre 81
Figura 7.9: Distintas configuraciones de agarre 82
Figura 8.1: Desviación del retardo general 86
Figura 8.2: Error de posición 86
Figura 10.1: Arquitectura general del sistema 95
Figura 10.2: Arquitectura Aplicación de captura 95
Figura 10.3: Aplicación de captura. Diagrama de flujo 96
Figura 10.4: Arquitectura Aplicación de seguimiento 97
Figura 10.5: Aplicación de seguimiento. Diagrama de flujo 98
Figura 13.1: Carpetas instalación paquete bhand 110
Figura 13.2: Carpetas instalación bhand versión 32bit 110
Figura 13.3: Instalación librerías BarrettHand 32bit 111
Figura 13.4: Carpetas instalación bhand versión 64bit 111

ÍNDICE DE TABLAS

TABLA 2.1: TIPOS DE DATOS EN MENSAJES 13
TABLA 2.2: TIPO DE DATOS EN SERVICIOS 14
TABLA 2.3: CARACTERÍSTICAS ENCÓDER 18
TABLA 3.1: CAMPOS DE LA VARIABLE COMPARTIDA DE LOS SERVICIOS OPEN Y CLOSE 28
TABLA 3.2: CARACTERES PARA SELECCIÓN DE DEDOS 28
TABLA 3.3: MARGEN DE ÁNGULOS DE ARTICULACIONES DE LA BARRETTHAND 29
TABLA 3.4: CAMPOS DE LA VARIABLE COMPARTIDA DEL SERVICIO SET_FINGERS 29
TABLA 3.5: CAMPOS DE LA VARIABLE COMPARTIDA DEL SERVICIO SET_FINGER 30
TABLA 3.6: CAMPOS VARIABLE COMPARTIDA DEL SERVICIO SET_FINGERS_SPEED 30
TABLA 4.1: VELOCIDADES Y ACELERACIONES MÁXIMAS EN EL PÉNDULO 42
TABLA 4.2: RETARDO MOVEJ 43
TABLA 4.3: TIEMPO DE RESPUESTA 44
TABLA 4.4: PERIODO ADQUISICIÓN DE IMÁGENES 45
TABLA 4.5: TIEMPO SECCIÓN CRÍTICA 45
TABLA 6.1: TIPOS DE CAMPOS EN VARIABLES COMPARTIDAS 68
TABLA 6.2: VARIABLE COMPARTIDA DEL SERVICIO SET_FINGERS Y RANGO DE ARTICULACIONES BARRETTHAND 70
TABLA 6.3: VARIABLE COMPARTIDA DEL SERVICIO SET_FINGERS_SPEED Y RANGO DE VELOCIDADES BARRETTHAND 70
TABLA 7.1: PARÁMETROS DE ARTICULACIONES DE LA BARRETTHAND EN GRASPIT! 79
TABLA 7.2: CONFIGURACIÓN ARTICULAR DE LAS DISTINTOS AGARRES 82
TABLA 8.1: TASA DE ACIERTOS. LANZAMIENTO 12° 85
TABLA 8.2: TASA DE ACIERTOS. LANZAMIENTO 16° 85
TABLA 8.3: TASA DE ACIERTOS. LANZAMIENTO 20° 85
TABLA 8.4: RESUMEN DESVIACIONES ESTÁNDAR 86
TABLA 12.1: COSTES EQUIPOS 105
TABLA 12.2: COSTES MANO DE OBRA 105
TABLA 12.3: COSTE TOTAL 105

GLOSARIO

HSV: Hue Saturation Value

MRU: Movimiento Rectilíneo Uniforme

PC: Personal Computer

PCL: Point Cloud Library

RGB: Red Green Blue

ROS: Robot Operating System

TCP: Transmission Control Protocol

TFG: Trabajo Fin de Grado

UDP: User Datagram Protocol

VFH: Viewpoint Feature Histogram

RESUMEN

En este Trabajo de Fin de Grado se ha utilizado ROS (Robot Operating System) bajo un sistema operativo Linux para controlar la mano robótica programable BarrettHand BH8-262.
Han sido diseñadas una serie de aplicaciones (seguimiento y manipulación de objetos) utilizando un brazo robot industrial IRB 120 equipado con una BarrettHand. En estas aplicaciones se utiliza una cámara Kinect como sistema de seguimiento.
De forma añadida, se incluye un análisis de las principales características de una herramienta de planificación de agarre para la BarrettHand.

Palabras clave: BarrettHand, ROS, IRB-120, OpenCV, GraspIt!

ABSTRACT

In this Final Year Proyect, ROS (Robot Operating System) working on a Linux platform has been used to control a programmable grasper BarrettHand BH8-262.
A series of applications have been designed (tracking and handling objects) using an IRB 120 robotic arm equipped with a BarrettHand. In these applications a Kinect camera is used as a tracking system.
Additionally, an analysis covering the main features of a grasp planning tool for BarrettHand is also included.

Keywords: BarrettHand, ROS, IRB-120, OpenCV, GraspIt!

RESUMEN EXTENDIDO

En este Trabajo de Fin de Grado en primer lugar, se llevará a cabo un análisis de las diferentes funcionalidades que ofrece el framework de aplicaciones robóticas ROS, con el fin de realizar una aplicación que permita controlar la mano robótica BarrettHand BH8-262.
A continuación se estudiarán las principales características y principios de funcionamiento de la BarrettHand, teniendo en cuenta los aspectos software y hardware.
A partir de los dos apartados anteriores se desarrolla la aplicación de control de la BarrettHand que permite utilizar las acciones de control más fundamentales:

- Establecer ángulos de las diferentes articulaciones.
- Apertura y cierre de dedos.
- Configuración de velocidad.

Esta aplicación permitirá a cualquier usuario de ROS hacer uso de la BarrettHand de forma sencilla, evitando tener que utilizar la librería de control proporcionada por el fabricante. Además su estructura permitirá en un futuro, implementar con facilidad acciones de control más avanzadas que no han sido tratadas en este TFG.

Se realizan tres demostraciones prácticas utilizando un brazo robot industrial ABB IRB 120 que sirve como plataforma para la BarrettHand.

En la **primera aplicación**, la plataforma formada por el IRB 120 y la BarrettHand serán controlados por una aplicación de ROS que se encargará de detectar una pelota que realiza un movimiento oscilatorio (péndulo) para poder cogerla mientras se encuentra en movimiento.

La aplicación está compuesta por las siguientes secciones:

- **Seguimiento:** Se emplea la librería de visión artificial OpenCV con la que se realiza una segmentación de color en las imágenes obtenidas de una cámara Kinect.
- **Control de la BarrettHand:** Se realiza mediante comunicación con la aplicación de control de la BarrettHand, que será la encargada de enviar los comandos pertinentes vía puerto serie.
- **Comunicación con el IRB 120:** Para enviar las diferentes órdenes de movimiento al brazo robot se utiliza una conexión por área local mediante comunicación por *socket*.

La **segunda aplicación** consiste en un sistema de seguimiento en el cual el brazo robot seguirá en todo momento el movimiento de una pelota.

La **tercera aplicación** realizada con la colaboración de tres alumnos de la Universidad de Alcalá de Henares, utiliza la plataforma descrita anteriormente para llevar a cabo una clasificación de objetos en función de sus características.

El sistema utilizado para realizar esta demostración se encuentra dividido en tres módulos bien diferenciados:

- **Reconocimiento de objetos**: El módulo realizado por Alberto Lázaro Enguita en "3D recognition and pose estimation using VFH descriptors" es el encargado de detectar y clasificar los diferentes objetos captados por medio de una cámara Kinect.
- **Movimiento del IRB 120:** Módulo realizado por Javier Moriano Martín en "Trajectory planning for the IRB 120 robotic arm using ROS" que lleva el control de las trayectorias del brazo robot.

- **Control de la BarrettHand:** Este módulo descrito en el presente documento, se ocupa de realizar las acciones con la mano robótica para poder coger y depositar los diferentes objetos.

Por último, se expondrán las principales características de la herramienta GraspIt!, que permite la planificación del agarre de objetos con la BarrettHand. Para ilustrar sus funcionalidades se presenta un ejemplo en entorno de simulación.

CAPÍTULO 1

INTRODUCCIÓN

1. INTRODUCCIÓN

1.1 Planteamiento

Actualmente la robótica es ampliamente utilizada en diversas áreas como pueden ser industria, medicina, servicios, etc. En muchos casos aparece la necesidad de manejar diferentes objetos, por ejemplo en cadenas de montaje de automóviles posicionando piezas pesadas; paletizado de productos en líneas de producción, etc.
Uno de los productos comerciales diseñado para este propósito es la BarrettHand. Se trata de una mano robótica con 3 dedos capaz de adaptarse para manipular objetos de diferentes formas y tamaños. Esta mano puede ser utilizada con facilidad en una gran variedad de robots industriales utilizando un PC para su control.
La incorporación del control de la BarrettHand al entorno ROS, permitiría utilizar este elemento hardware en las diferentes aplicaciones que lo requieran y que hayan sido desarrolladas con este framework. De esta forma se reduce el tiempo de desarrollo de todas aquellas aplicaciones que hagan uso de este recurso.

1.2 Estado del arte

En la actualidad existen múltiples soluciones comerciales que permiten a diferentes sistemas robóticos manipular objetos. Estos productos van desde simples y robustos *gripper,* pensados para trabajar en entornos industriales en los que se requieren realizar tareas repetitivas sobre objetos y que en la mayoría de las ocasiones trabajaran con un tipo concreto de objeto, a complejas manos robóticas antropomórficas que son utilizadas en medicina como prótesis debido a su gran destreza para adaptarse a los diferentes objetos que se presentan en la vida cotidiana.
A continuación se muestran algunos de estos dispositivos que actualmente se encuentran en el mercado.

1.2.1 Grippers industriales

- **Servo-electric 3-Finger Gripping Hand SDH**

Diseñado por la compañía SCHUNK [1], su funcionamiento es eléctrico y está pensado para trabajar en entornos con polvo y humedad.
Está compuesto por tres dedos y cuenta con sensores de presión que le proporcionan la capacidad de agarrar objetos de distintas formas y tamaños.

Figura 1.1: SCHUNK Gripper

Figura 1.2: Configuraciones SCHUNK Gripper

- **Robotiq 3-Finger Adaptive Gripper**

El gripper fabricado por la compañía Robotiq [2] cuenta con tres dedos y su accionamiento es eléctrico. Es un modelo de gripper diseñado para trabajar en la industria con el objetivo de ser capaz de manipular distintos tipos de objetos, obteniendo de esta forma una solución robusta y flexible.

Figura 1.3: Robotiq Gripper

Los grados de libertad que poseen los dedos permiten sujetar un amplio catálogo de objetos.

Figura 1.4: Configuraciones Robotiq Gripper

1.2.2 Manos robóticas antropomórficas

- **Shadow Dexterous Hand**

La Shadow Dexterous Hand [3] se trata de una mano robótica que trata de emular la forma y funcionalidad de una mano humana a la hora de coger y manipular objetos.

Figura 1.5: Shadow Dexterous Hand

Los veinte grados de libertad disponibles permiten controlar sus cinco dedos permitiendo coger objetos de pequeño tamaño y forma irregular.
De forma añadida, la mano cuenta con soporte en ROS lo que permite tanto su simulación como el control real.

- **ELU-2 Hand**

La ELU-2 Hand [4] es una mano robótica utilizada generalmente en aplicaciones de investigación en el campo de la medicina.

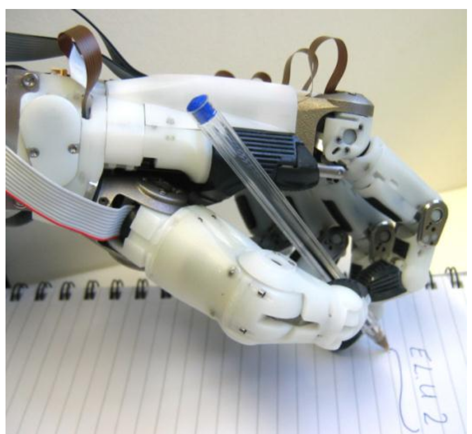


Figura 1.6: Elu-2 Hand

El conjunto de las articulaciones le confieren nueve grados de libertad y dispone de sensores de presión que permiten manipular objetos frágiles como vasos o herramientas.

1.2.3 Paquetes de control de manos robóticas en ROS

De los cuatro dispositivos mostrados en las páginas anteriores, tan solo uno de ellos ofrece soporte para el framework ROS. No disponer de aplicaciones o utilidades para controlar estos dispositivos hardware sobre ROS dificulta su inclusión en proyectos desarrollados con este framework, siendo necesario generar una interfaz que permita intercomunicar las aplicaciones de ROS con el hardware abstrayendo al desarrollador de la aplicación de las capas de más bajo nivel de la comunicación.
El objetivo de este TFG es generar esta interfaz para la mano robótica BarrettHand, pero antes de comenzar con el desarrollo conviene analizar las soluciones que se encuentran actualmente disponibles para realizar el control de la BarrettHand en entorno ROS:

- **gwam-ros-package**

Este paquete está pensado para controlar el brazo robot WAM y la BarrettHand de Barrett Technology desde entorno ROS. Ofrece varias funciones para el control de la BarrettHand que permiten la apertura y cierre de dedos y una selección de velocidad.
Las principales desventajas que desaconsejan el uso de este paquete para el control de la BarretHand son:

- Está diseñado para trabajar con el brazo robot WAM y para su funcionamiento es necesario disponer del WAM PC (ordenador de Barrett Technology utilizado para el control del brazo robot WAM). Por lo tanto, aunque solo se quiera utilizar la BarrettHand, sería necesario disponer de un WAM PC.

- Está construido para ser utilizado en ROS Electric (versión de ROS del año 2011), por lo que sería necesario portar el paquete a una versión más actual.

- **owd**

El paquete owd (OpenWAM driver) al igual que el paquete anterior permite el control del WAM y de la BarrettHand. Las funcionalidades que ofrece son muy similares y en este caso, el funcionamiento del paquete hace que no sea necesario el uso del WAM PC por lo que es más adecuado para el control individual de la BarrettHand.
El principal inconveniente de este paquete es que en su instalación en ROS Hydro presenta problemas de compatibilidad de librerías.

Debido a las dificultades en el uso de los paquetes de control de la BarrettHand que se encuentran disponibles, queda justificado el desarrollo de un nuevo paquete que sea de fácil instalación y compatible con las últimas versiones de ROS, como ROS Hydro.

CAPÍTULO 2

ARQUITECTURA DEL SISTEMA

2. ARQUITECTURA DEL SISTEMA

2.1 Introducción

En la figura 2.1 se muestra el conjunto de elementos que forman el sistema y que se utilizarán en este. En este capítulo se describirán sus principales características.

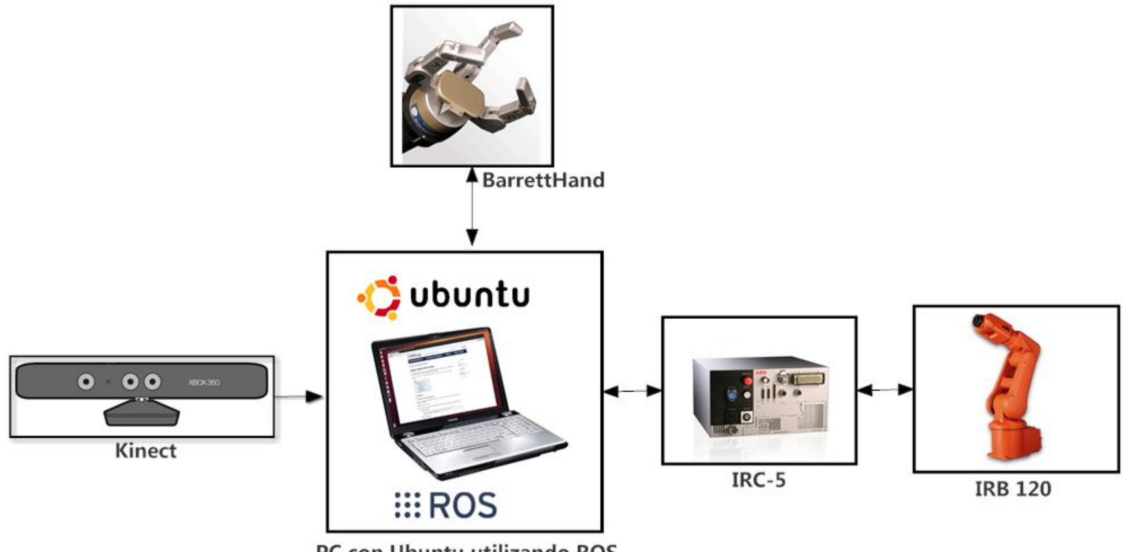


Figura 2.1: Arquitectura general del sistema

2.2 ROS

2.2.1 Introducción

Para entender qué es ROS [5], en primer lugar se ha de comprender el concepto de *framework* utilizado en el área del desarrollo de software.

Un framework es un conjunto de librerías, utilidades y métodos de programación que permiten el desarrollo de aplicaciones orientadas a cierta área concreta cómo puede ser: diseño web, visión artificial, robótica, etc.
Las principales ventajas del uso de un framework son:

- Reducción del tiempo de desarrollo de aplicaciones
- Mejora en la portabilidad de las aplicaciones desarrolladas
- Facilidad de mantenimiento del código

Por otro lado, su uso también conlleva una serie de desventajas:

- Curva de aprendizaje lenta
- Mayor consumo de recursos
- Puede complicar el desarrollo de aplicaciones sencillas

Con esta definición del concepto de framework, se puede entender ROS como un **framework destinado al diseño y posterior implementación de aplicaciones en el campo de la robótica**.
El funcionamiento de ROS se basa principalmente en tres elementos:

- Paquetes
- Nodos
- Métodos de comunicación entre nodos

2.2.2 Paquetes

Los paquetes son elementos que permiten agrupar una serie de aplicaciones (nodos) o librerías, facilitando su portabilidad e instalación en diferentes equipos.
Todos los paquetes tendrán una estructura básica común similar a la indicada en la figura 2.2.

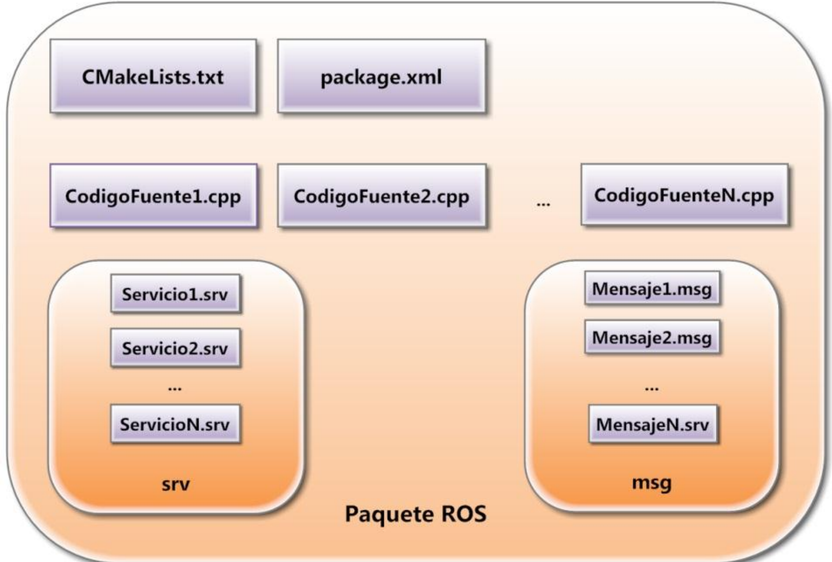


Figura 2.2: Estructura Paquetes ROS

Los elementos por los que está compuesto habitualmente el paquete se describen a continuación:

- **CMakeLists.txt**

  Este fichero contiene instrucciones que serán utilizadas para construir el paquete, como pueden ser: nombre del paquete, árbol de dependencias de otros paquetes, rutas de librerías, rutas de ficheros de mensajes o servicios a construir, ejecutables a generar, etc.

- **package.xml**

  De nuevo, se trata de un fichero utilizado para definir dependencias del paquete, su nombre, versión, etc Es muy similar a CMakeList.txt pero tiene un carácter más informativo.

- **CodigoFuente.cpp**

  Son aquellos códigos fuente que definen la aplicación o aplicaciones por las que está compuesto el paquete. El código fuente puede estar en lenguaje de programación C++ (.cpp) o bien en Python (.py). En ocasiones los códigos fuente pueden localizarse dentro de un directorio con nombre *src*.

- **Directorio srv**

  En este directorio se alojan, en caso de ser necesarios, los ficheros que definen la estructura de los servicios (véase apartado 2.1.4).

- **Directorio msg**

  En este directorio se incluirán los ficheros que definen la estructura de los mensajes (véase apartado 2.1.4), en caso de ser necesarios.

Esta estructura de los paquetes permite construir los ejecutables de forma sencilla con la herramienta *catkin*.
Esta herramienta incluida en ROS Hydro y que a su vez se trata de un paquete, es la aplicación encargada de construir paquetes generando los ejecutables.
En catkin quedan incluidas diferentes aplicaciones que llevan a cabo la construcción del paquete, como puede ser: *Cmake* o *GNU C++ Compiler.*
Por defecto en la instalación de ROS se incluyen múltiples paquetes con diversas funcionalidades. Algunos de los paquetes que son incluidos en la instalación de ROS y que han sido utilizados para este TFG son los siguientes:

- **opencv2**

  Contiene la librería OpenCV orientada a visión artificial.

- **sensor_msgs**

  Incluye diferentes estructuras de datos utilizadas en ROS que facilitan el trabajo con distintos tipos de sensores: temperatura, imagen, laser…
  Una de las estructuras más utilizadas de este paquete es *Image Message* en aplicaciones que hacen uso de imágenes.

- **cv_bridge**

  Su fin es convertir las imágenes de formato ROS a formato OpenCV. Debido a que los diferentes nodos de ROS que trabajan con imágenes utilizan un formato específico de ROS (*Image Message*), es necesario convertir este formato al formato utilizado en OpenCV (*Mat* o *IplImage*).

Además de los paquetes que vienen incluidos en la instalación de ROS, existe una gran variedad de paquetes creados por la comunidad: *ur_kinematics* para el cálculo de la cinemática directa e inversa de robots; *industrial_core* con aplicaciones para comunicarse y controlar robots industriales; *freenect* para el uso de la Kinect, etc.

2.2.3 Nodos

Los nodos no son más que las aplicaciones de ROS que se ejecutarán en el sistema para llevar a cabo las diferentes tareas pudiendo comunicarse entre sí.
Por ejemplo, en un robot móvil puede existir un nodo que se ocupe del control de los motores que mueven las ruedas, otro se encargaría de obtener las imágenes de un sistema de visión y un último nodo trataría las imágenes para detectar posibles obstáculos y se comunicaría con el nodo de control de motores para que envíe las ordenes adecuadas a los motores, con el fin de evitar el obstáculo.

Uno de los nodos más importantes y que se encuentra presente en todos los sistemas que utilizan ROS es el nodo *Master*.
Este nodo se puede considerar el núcleo del sistema ROS siendo necesario para el funcionamiento del resto de nodos.
Sus funciones principales son:

- Llevar un registro de los nodos que se están ejecutando.
- Permitir comunicación entre nodos por medio de servicios y mensajes.

En la figura 2.3 se puede observar la estructura que seguirían los nodos del ejemplo mencionado anteriormente.

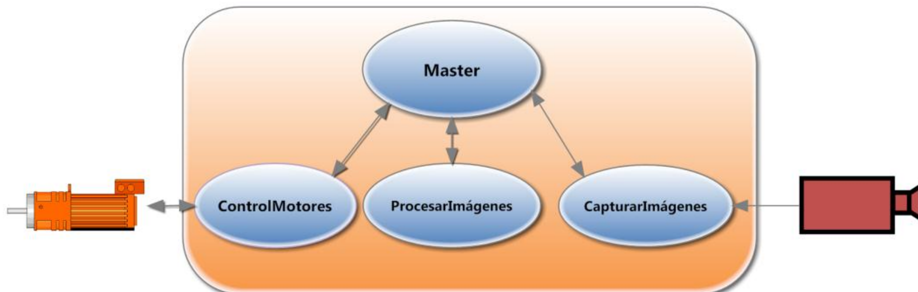


Figura 2.3: Estructura Nodos

Se puede observar cómo la comunicación entre nodos se hace siempre por medio del *Master*. En los siguientes esquemas y diagramas de flujo en los que aparezcan nodos, para simplificar, se omitirá el nodo *Master*, pero hay que tener presente que la comunicación entre los distintos nodos harán uso del *Master*.

2.2.4 Métodos de comunicación entre nodos

La comunicación entre los distintos nodos por que forman el sistema es algo fundamental para construir sistemas complejos que requieren del uso de múltiples nodos.
ROS provee de diferentes métodos para comunicar los nodos entre sí. Estos métodos permiten la comunicación de forma sencilla manteniendo una estructura común para facilitar la compatibilidad y modularidad entre distintos paquetes.

Los principales métodos de comunicación son los siguientes:

- Comunicación por *topic*
- Comunicación por servicios

A) Comunicación por topic

En ROS, el *topic* se puede entender como un tablón de anuncios en el que algunos nodos podrán ir dejando mensajes y otros podrán recogerlos.

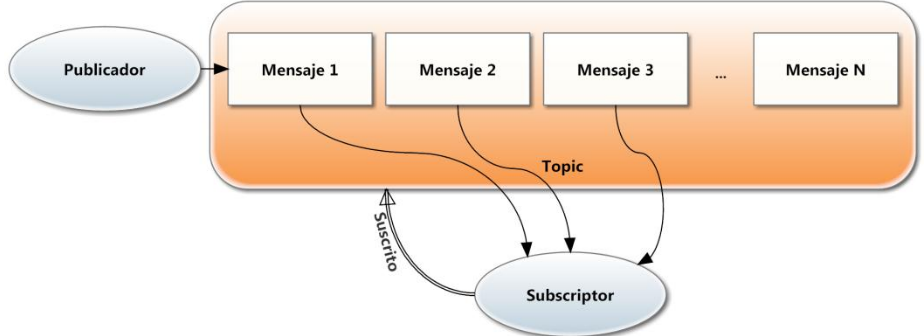


Figura 2.4: Comunicación por topic

En la comunicación por topic existen dos roles bien diferenciados:

- Publicador
- Subscriptor

El nodo que actúa como **publicador** será el encargado de crear el *topic* para poder difundir ciertos paquetes de información llamados *mensajes,* que podrán ser visibles por aquellos nodos que se encuentren suscritos al *topic.* Habitualmente esta información será publicada con cierta periodicidad.

Por otro lado, el **suscriptor** es el nodo que necesita utilizar la información que está siendo publicada por un nodo publicador en un *topic*.
Este nodo se deberá suscribir a los *topics* que necesite antes de poder acceder a los mensajes que están siendo publicados.
Como es habitual en ROS, con el fin de facilitar la modularidad de los paquetes, la estructura y contenido de los mensajes está bien definida.
El mensaje tiene una estructura sencilla de datos que contiene diferentes campos. Esta estructura queda reflejada en un **fichero de mensaje *.msg***.
En la tabla 2.1 se muestran algunos de los tipos de datos que pueden ser utilizados en los distintos campos del mensaje.

Tabla 2.1: Tipos de datos en mensajes

Tipos	Descripción del tipo
int8	Entero de 8 bit
int64	Entero de 64 bit
float32	Flotante de 32bit
string	Cadena
time	Variable de tiempo

Además de estos tipos de datos existen otros muchos tipos más específicos que pueden ser utilizados, como por ejemplo el tipo *Image* utilizado para almacenar imágenes obtenidas de cámaras.

El fichero de mensaje por norma general deberá estar contenido dentro de aquellos paquetes que hagan uso de éste, tanto en el caso en el que un nodo lo publique o se suscriba a un *topic* que utilice ese tipo de mensajes.
La comunicación por topic tiene como inconveniente que **no dispone de mecanismos para que el publicador tenga constancia de que uno de los mensajes publicados ha sido recibido por el suscriptor**.

B) Comunicación por servicios

Uno de los mayores inconvenientes de la comunicación por medio de *topic* es que no se dispone de un mecanismo que permita identificar si un cierto mensaje que se ha publicado ha sido recibido por un subscriptor concreto.
En algunos casos puede ser necesario transmitir órdenes con pequeños paquetes de información entre distintos nodos, asegurando que una vez se ha enviado la orden podamos conocer si ha sido recibida o no.
Para este fin ROS facilita la comunicación por medio de servicios.

Los servicios permiten una comunicación bidireccional entre nodos que posibilitan el envío de órdenes y la transmisión de información entre los diferentes nodos.
La estructura genérica de un servicio se muestra en la figura 2.5.

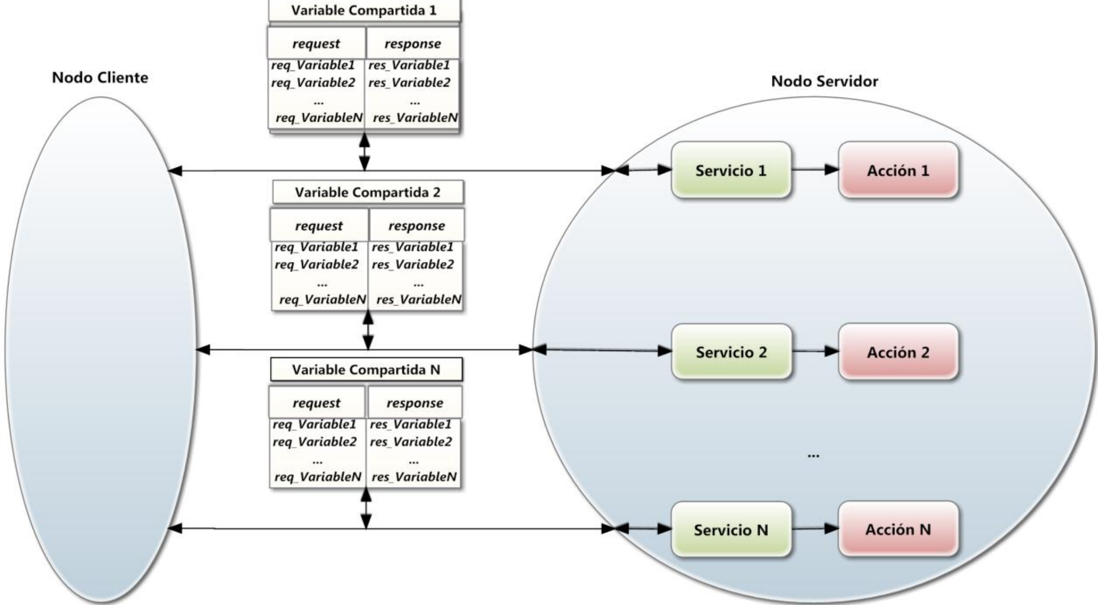


Figura 2.5: Comunicación por servicios

Se distinguen cuatro bloques bien diferenciados:

- **Servidor**

  Nodo encargado de suministrar los servicios a aquellos nodos que actúen como cliente.

- **Cliente**

  Es el nodo que hace uso de los servicios proporcionados por un nodo servidor. Podrá trasmitir información al servidor y también recibirla.

- **Variable de servicio**

  Es el elemento clave en la comunicación por servicios. Se trata de una variable compartida entre el cliente y el servidor con la que se puede transmitir información entre los dos nodos.
  Esta variable se trata de una estructura compuesta por los campos ***request*** y ***response***.
  El campo *request* contiene aquellas variables que serán escritas por el cliente para ser transmitidas al servidor.
  El campo *response* contiene las variables que serán rellenadas por el servidor y que se transmiten como respuesta al cliente.
  Los tipos que pueden ser utilizados para estas variables están definidos en la tabla 2.2.

Tabla 2.2: Tipo de datos en servicios

Nombre de la variable	Descripción
Int8, int16, int32, int64	Enteros de 8,16,32 y 64 bit
float32, float64	Coma flotante de 32 y 64 bit
string	Cadena de caracteres
time, duration	Estructuras de tiempo

- **Acción asociada al servicio**

  En el nodo del servidor existe una acción única asociada a cada servicio ofrecido. En esta acción el servidor tendrá acceso a la sección ***request*** de la variable compartida del servicio,

pudiendo utilizar los datos escritos por el cliente para realizar diferentes funciones: comunicación con periféricos, procesado de datos, etc.

Finalmente, una vez se han llevado a cabo las diferentes funciones, el servidor podrá retornar sus resultados en la sección ***response*** de la variable compartida. Estos resultados podrán ser leídos por el cliente.

Por otro lado, al margen de la variable compartida, estas acciones pueden retornar un valor que puede ser leído en el cliente tras hacer la llamada al servicio, utilizándose habitualmente para que el cliente compruebe que la acción asociada al servicio se ha ejecutado correctamente.

2.3 BarrettHand

2.3.1 Introducción

La BarrettHand BH8-262 es un modelo de mano robótica de tres dedos diseñada por la empresa estadounidense *Barrett Technology* [6] cuyo diseño pretende simplificar el agarre de objetos de diferentes tamaños y formas.

Figura 2.6: BarrettHand BH8-262

Algunas de sus principales características son:

- Peso de 2 kg
- Capacidad de carga de hasta 2kg por dedo
- Alta velocidad de las articulaciones (cierre completo de dedos a velocidad máxima en 1 segundo)

El control de la mano se realiza mediante un ordenador por medio de comunicación serie mediante el estándar *RS-232*. Para este fin, *Barrett Technology* facilita una extensa librería en lenguaje de programación C++ mediante la cual se realiza la comunicación de una forma sencilla y directa. Esta librería es soportada por los sistemas operativos Linux y Windows.

Debido al elevado consumo de los motores de las articulaciones, para su uso cuenta con una fuente de alimentación AC-DC facilitada por el fabricante, que se encarga de proporcionar los diferentes niveles de tensión requeridos.

Figura 2.7: Fuente de alimentación

2.3.2 Estructura mecánica

Los elementos principales en los que se sustenta la BarrettHand para realizar la función de coger objetos son tres dedos articulados compuestos por dos falanges.

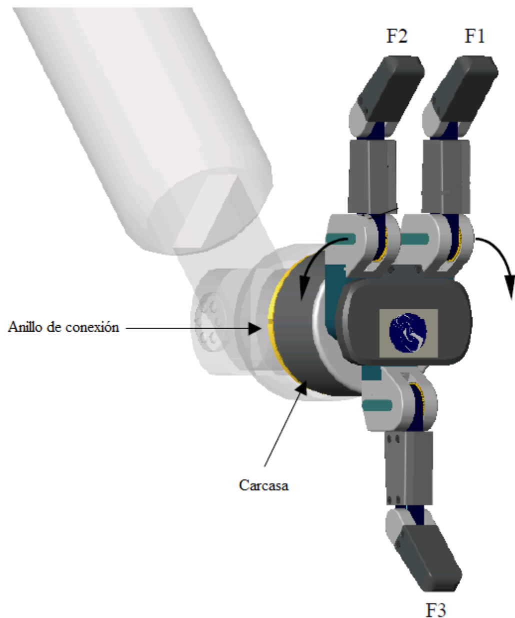


Figura 2.8: Articulaciones BarrettHand

El correcto posicionamiento de estos dedos permitirá agarrar gran variedad de objetos.
Cada dedo es controlado mediante un motor *brushless* alojado en la carcasa, que transmite el movimiento mediante transmisión mecánica compuesta por engranajes y cables.
Los dedos uno y dos cuentan con un grado de libertad extra, siendo capaces de rotar en torno a la palma de la mano como muestra la figura 2.8. Esta rotación está controlada por un cuarto motor y es común a los dos dedos, es decir, la rotación de estos dos dedos respecto de la palma de la mano se hace de forma simétrica. El dedo tres no dispone de este grado de libertad.

En la figura 2.9 y figura 2.10 aparecen los rangos máximos que permite cada articulación.

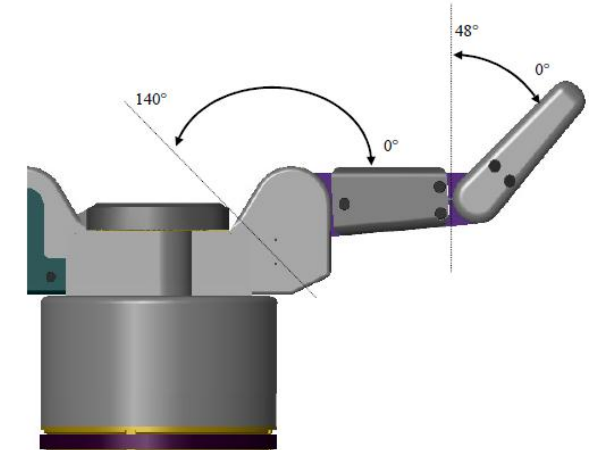


Figura 2.9: Rango articulaciones de los dedos

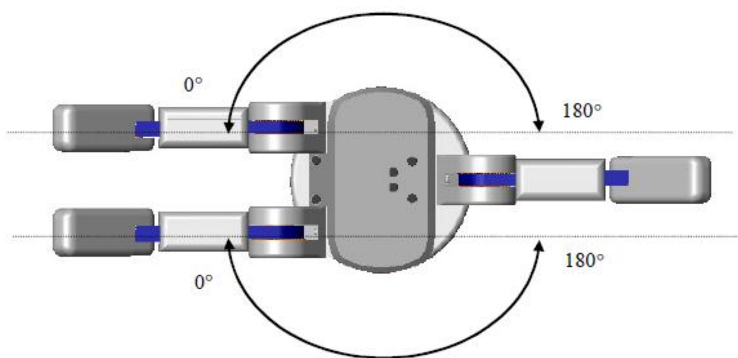


Figura 2.10: Rango giro entorno la palma

Una de las características más importantes de los dedos es que la falange final no es controlable de forma directa por el usuario. El sistema encargado de realizar este control se denomina ***TorqueSwitch.***

TorqueSwitch es un sistema mecánico que controla el ángulo de la última falange de cada dedo de forma automática. Este control dota a la BarrettHand de cierto grado de adaptabilidad permitiendo coger objetos de tamaños y formas variadas.

Su principio de funcionamiento se basa en un sistema mecánico que mide el par resistente ejercido sobre la primera falange. Cuando el par de la primera falange supera un umbral determinado (que puede ser configurado mediante software) un embrague desacopla la "Articulación 1" (figura 2.11) impidiendo que continúe moviéndose, pero permite que la "Articulación 2" (figura 2.11) continúe su movimiento para adaptarse a la forma del objeto.

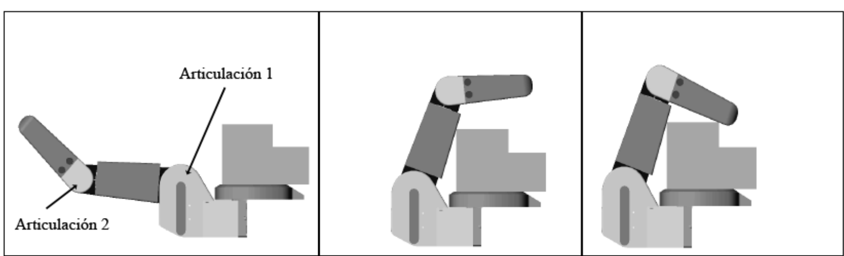


Figura 2.11: Operación TorqueSwitch

En lo referido a la conexión mecánica con un brazo robot industrial, la carcasa de la BarrettHand no dispone de roscas compatibles con las monturas de conexión de herramienta habituales, por ello debe ser fijada por medio de un adaptador proporcionado por el fabricante.

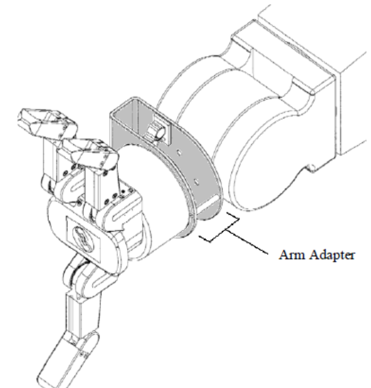


Figura 2.12: Acoplamiento BarrettHand

2.3.3 Electrónica de control

El núcleo de la electrónica de control de la BarrettHand se basa en un microcontrolador Motorola 68HC811E2FN encargado de la comunicación con el PC mediante puerto serie, ejecución del firmware y comunicación con los driver de control de motores.

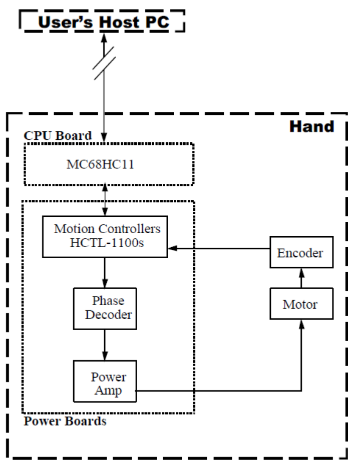


Figura 2.13: Electrónica de control

Otro de los elementos más importantes de la electrónica de control es el encóder. Cada motor está equipado con un encóder óptico que entrega 360 cuentas por cada vuelta del eje del motor. Debido a que las articulaciones no se encuentran conectadas directamente a los motores si no que cuentan con un reductor mecánico, los ángulos de los motores no se corresponderán con los de las articulaciones debido al factor de reducción de la reductora por engranajes.

En la tabla 2.3 se encuentran reflejadas las especificaciones que relacionan las cuentas de los encóder de los motores con el ángulo de las articulaciones que pueden ser controladas de forma directa por el usuario.

Tabla 2.3: Características encóder

Articulación	Cuentas máximas	cuentas/°	Resolución
Dedo 1	17500	125	0.008°
Dedo 2	17500	125	0.008°
Dedo 3	17500	125	0.008°
Rotación	3150	18	0.056°

2.3.4 Librería de control

La librería de control de la BarrettHand [7] es una librería de funciones para lenguaje de programación C++. Su propósito es, tanto controlar los diferentes parámetros de la BarrettHand: ángulos de las articulaciones, velocidad de giro u otros, como adquirir valores de los diferentes sensores con los que está equipada: temperatura, posición, fuerza, etc.
Para su uso se utiliza una clase de tipo *BHand* que contiene todas las funciones que se pueden manejar.
A continuación se describirán las diferentes funciones de la librería que han sido utilizadas en este TFG.

- **InitSoftware**

Sintaxis: int InitSoftware(int port)

Uso: Inicia la comunicación con la BarrettHand y resetea los diferentes registros a valores por defecto

Argumentos: port: Número del puerto serie para la comunicación

Valores: port: 0,1 ,2…

Ejemplo:

```
BHand bh;
bh.InitSoftware(1);  // COM1 para la comunicación
```

- **InitHand**

Sintaxis: int InitHand (char* motor)

Uso: Inicializa los motores de las articulaciones que controlan los dedos y calibra los encóder

Argumentos: motor: Especifica los motores que deben ser inicializados

Valores:

Valor	Descripción
"1"	Dedo 1
"2"	Dedo 2
"3"	Dedo 3
"4"	Rotación dedo 1 y 2
"G"	Dedo 1, dedo 2 y dedo 3
" " (Cadena vacía)	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2
"1234"	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2

Ejemplo:

```
bh.InitHand("1234");  // Inicializa todas las articulaciones
```

Nota: La cadena que se utiliza como argumento se puede construir con las diferentes combinaciones posibles de dedos. Por ejemplo, para actuar sobre el dedo 1 y el dedo 3, se puede construir la cadena "13", para actuar sobre el dedo 2 y el dedo 3 se utilizaría la cadena "23" y así sucesivamente.

- **Open**

Sintaxis: int Open (char *motor)

Uso: Apertura de dedos

Argumentos: motor: Especifica los motores que realizan el movimiento de apertura

Valores:

“1”	Dedo 1
“2”	Dedo 2
“3”	Dedo 3
“4”	Rotación dedo 1 y 2
“G”	Dedo 1, dedo 2 y dedo 3
“ “ (Cadena vacía)	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2
“1234”	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2

Ejemplo:

bh.Open(“12”); // Apertura dedo 1 y dedo 2
bh.Open(“14”); //Apertura dedo 1 y rotación dedos 1 y 2

Nota: Esta función no garantiza que se lleve a cabo de forma completa la acción de apertura, ya que si en la trayectoria de apertura alguno de los dedos encuentra un obstáculo se podrá producir la parada de dicho dedo.

- **Close**

Sintaxis: int Close (char *motor)

Uso: Cierre de dedos

Argumentos: motor: Especifica los motores que realizan el movimiento de cierre

Valores:

“1”	Dedo 1
“2”	Dedo 2
“3”	Dedo 3
“4”	Rotación dedo 1 y 2
“G”	Dedo 1, dedo 2 y dedo 3
“ “ (Cadena vacía)	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2
“1234”	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2

Ejemplo:

bh.Close(“12”); // Cierre dedo 1 y dedo 2
bh.Close(“124”); //Cierre dedo 1, dedo 2 y rotación dedos 1 y 2

Nota: El movimiento de cierre de las articulaciones sigue el sentido indicado en la figura 2.11. Esta función no garantiza que se lleve a cabo de forma completa la acción de cierre, ya que si en la trayectoria de cierre alguno de los dedos encuentra un obstáculo se podrá producir la parada de dicho dedo.

- **GoToDifferentPositions**

Sintaxis: int GoToDifferentPositions (int value1, int value2, int value3, int value4)

Uso: Posicionamiento de todos los motores en ángulos concretos

Argumentos: value1, 2, 3, 4: Especifica la posición del encóder para cada uno de los cuatro motores

Valores: value1 [0-17500]
value2 [0-17500]
value3 [0-17500]
value4 [0-3150]

Ejemplo: bh.GoToDifferentPositions(2000,3000,4000,1000);

- **GoToPosition**

Sintaxis: int GoToPosition (char* motor, int value)

Uso: Posicionamiento de uno o varios motores en un ángulo concreto

Argumentos: motor: Especifica el motor o motores que ejecutarán el movimiento y mantiene el ángulo del resto
value: Especifica la posición del encóder

Valores:

motor:

“1”	Dedo 1
“2”	Dedo 2
“3”	Dedo 3
“4”	Rotación dedo 1 y 2
“G”	Dedo 1, dedo 2 y dedo 3
“ “ (Cadena vacía)	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2
“1234”	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2

value [0-17500] para la articulación 1,2 3

value [0-3150] para la articulación 4

Ejemplo: bh.GoToPosition(“ 1”,8750);

- **Set**

La función Set permite modificar los valores de configuración de diferentes parámetros de la BarrettHand relacionados con el control de motores. La lista completa de los parámetros que pueden ser modificados se encuentra en el manual de usuario de la BarrettHand [8].

Sintaxis: int Set(char* motor, char* parameter, int value)

Uso: Configuración de parámetros de motores

Argumentos: motor: Especifica el motor o motores en los que se quieren modificar parámetros

parameter: Especifica el parámetro que va a ser modificado
value: Valor que adquirirá el parámetro

Ejemplo: bh.Set("1","MOV",500); //Configura el parámetro de máxima velocidad de apertura (*Maximum Open Velocity)* en 500 para el motor 1.

- **Get**

La función Get permite adquirir valores de diferentes parámetros de la configuración de los motores. La lista completa de los parámetros que pueden ser leídos se encuentra en el manual de usuario de la librería [8].

Sintaxis: int Get(char* motor, char* parameter, int result)

Uso: Lectura de parámetros de motores

Argumentos: motor: Especifica el motor o motores en los que se quieren modificar parámetros

parameter: Especifica el parámetro que va a ser modificado
result: Valor del parámetro

Ejemplo: bh.Get("1","P",&Position); //Adquiere la posición del encóder del motor 1

- **StopMotor**

Sintaxis: int StopMotor (char *motor)

Uso: Detiene el funcionamiento de uno o varios motores

Argumentos: motor: Especifica los motores a detener

Valores:

"1"	Dedo 1
"2"	Dedo 2
"3"	Dedo 3
"4"	Rotación dedo 1 y 2
"G"	Dedo 1, dedo 2 y dedo 3
" " (Cadena vacía)	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2
"1234"	Dedo 1, dedo 2, dedo 3 y rotación dedo 1 y 2

Ejemplo: bh.StopMotor("1234"); // Detiene los cuatro motores

2.4 Brazo robot industrial IRB 120

El IRB 120 es un brazo robot industrial diseñado y comercializado por la empresa ABB [9].
Se trata de un brazo robot de pequeñas dimensiones y bajo peso capaz de llevar a cabo múltiples tareas, como pueden ser: paletizado, ensamblado de electrónica de consumo, *pick and place*, etc.

Figura 2.14: IRB 120

Algunas de sus características más destacables son:

- Dispone de seis grados de libertad
- Capacidad de carga de hasta 4 kg
- Velocidad del eslabón final de hasta 2000mm/s
- 25 kg de peso
- Alta repetitividad en el posicionamiento de la herramienta con errores de hasta 0.01 mm
- Radio de área de trabajo de hasta 580mm
- Tiempo de aceleración de 0 a 1m/s de 0.07s

El control del IRB 120 se realiza por medio de un equipo denominado IRC5.
Es un controlador equipado con varios módulos que permiten controlar los diferentes brazos robot de la compañía ABB. Sus principales módulos son los siguientes:

- Hardware de control
- Interfaces de comunicación (E/S digitales y analógicas, Ethernet, DeviceNet, etc.)
- Interface de usuario FlexPendant
- Dispositivos de seguridad (paro de emergencia, bloqueo de frenos, etc.)

Figura 2.15: IRC5 y FlexPendant

2.5 Kinect

La cámara Kinect [10] es un sistema de visión desarrollado por Microsoft con el propósito de ser utilizada como interfaz para juegos en videoconsolas Xbox.

Figura 2.16: Kinect

Gracias a su bajo coste y múltiples funcionalidades, su uso se ha generalizado en otros muchos campos de aplicación, como pueden ser: navegación por el entorno en aplicaciones robóticas, scanner de objetos 3D, reconocimiento facial, etc.
Sus principales componentes hardware son:

- Cámara RGB para captura de imágenes en color (sensor Micron MT9M112) con resolución de 640 x 512 con una tasa de 30 imágenes por segundo.
- Sensor de profundidad compuesto por un emisor y sensor de infrarrojos.

La principal característica que distingue a la Kinect de una cámara convencional es su capacidad para obtener imágenes de profundidad. Esta característica es posible gracias al sensor integrado de profundidad, el cual permite obtener imágenes del entorno generando un mapa en tres dimensiones.

CAPÍTULO 3

PAQUETE DE CONTROL DE LA BARRETTHAND EN ROS

3. PAQUETE DE CONTROL DE LA BARRETTHAND EN ROS

3.1 Introducción

Para permitir el control de la BarrettHand BH8-262 en ROS, se ha creado el paquete ***bhand***. Hace uso de la librería de comandos en C++ [7] que suministra Barrett Technology y permite al resto de nodos de ROS realizar el control de la BarrettHand a alto nivel sin necesidad de utilizar los comandos de la librería del fabricante mencionada anteriormente.

Este paquete está compuesto por dos nodos:

- Nodo servidor: ***bhand_server***
- Nodo cliente: ***bhand_client***

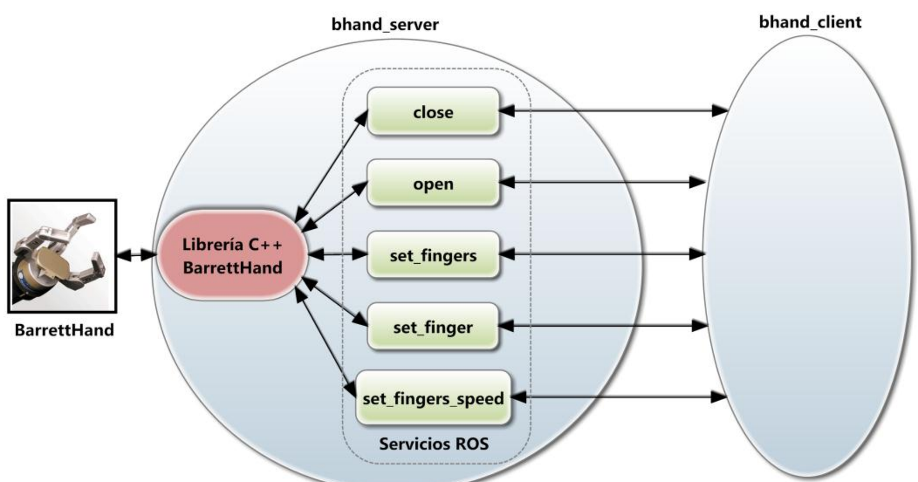


Figura 3.1: Servicios bhand_server

3.2 Nodo bhand_server

El funcionamiento de este nodo se sustenta en el uso de la comunicación por medio de servicios (véase apartado 2.1.4) y es el encargado de proporcionar los diferentes servicios disponibles al resto de nodos que lo requieran. Una vez inicializado se mantendrá ejecutándose en segundo plano realizando las diferentes acciones de los diferentes servicios cuando sea requerido.
Los servicios que proporciona este nodo son los siguientes:

- close
- open
- set_fingers
- set_finger
- set_fingers_speed

Figura 3.2: Secuencia cierre de dedos

A continuación se detalla la funcionalidad y los parámetros requeridos por cada servicio.

- **close**

Este servicio está encargado de efectuar el cierre de los dedos para poder agarrar objetos.
Para el uso de este servicio se utiliza una variable compartida del tipo **Open_Close_Hand**. En la tabla 3.1 se encuentran definidos los distintos campos por los que está compuesta la variable.

Tabla 3.1: Campos de la variable compartida de los servicios open y close

bhand::Open_Close_Hand	
request	**response**
string FingerSelect	float32 ThetaF1 [0-140]°
	float32 ThetaF2 [0-140]°
	float32 ThetaF3 [0-140]°
	float32 ThetaF4 [0-180]°

El campo *FingerSelect* es una cadena de caracteres que contiene la numeración de los dedos que deben ser accionados. Esta numeración se encuentra recogida en la tabla 3.2 y será utilizada en el resto de servicios.

Tabla 3.2: Caracteres para selección de dedos

Articulación	string FingerSelect
F1	‘1’
F2	‘2’
F3	‘3’
F4	‘4’

Las articulaciones F1, F2 y F3 hacen referencia a los tres dedos por los que está compuesta la mano (consultar figura 2.7 para localizar a que dedo se corresponden) mientras que F4 es la articulación que permite la rotación de los dedos respecto a la palma de la mano.
En el caso de querer utilizar varios dedos en una acción, se deberá componer una cadena que contenga la numeración de las articulaciones que se quieren utilizar sin dejar espacios entre caracteres.
Por ejemplo, si queremos cerrar las articulaciones F1 y F3 la cadena sería:

FingerSelect=’13’

En el caso de las articulaciones F2, F3 y F4 la cadena será:

FingerSelect=’234’

De este modo se pueden formar las distintas combinaciones posibles.

Los campos ThetaF1, ThetaF2, ThetaF3 y ThetaF4 contienen los ángulos finales de las articulaciones una vez se ha llevado a cabo la acción de cierre. Hay que tener en cuenta que cuando se agarra un objeto, los dedos normalmente no se cerrarán completamente, por lo que con estos campos podemos conocer como han quedado posicionados.
Los márgenes de valores posibles de cada articulación se muestran en la tabla 3.3.

Tabla 3.3: Margen de ángulos de articulaciones de la BarrettHand

float32 ThetaF1	[0-140]°
float32 ThetaF2	[0-140]°
float32 ThetaF3	[0-140]°
float32 ThetaF4	[0-180]°

- **open**

Este servicio es prácticamente idéntico a *close* y la única diferencia es que el sentido de giro de las articulaciones produce la apertura de los dedos.
La estructura de la variable compartida es Open_Close_Hand al igual que en el servicio anterior.

- **set_fingers**

Con este servicio se pueden fijar los ángulos de las cuatro articulaciones disponibles. Para ello se utiliza una variable compartida del tipo **Set_Fingers_Angles**.

Tabla 3.4: Campos de la variable compartida del servicio set_fingers

bhand::Set_Fingers_Angles	
request	**response**
float32 ThetaF1 [0-140]°	float32 ThetaF1 [0-140]°
float32 ThetaF2 [0-140]°	float32 ThetaF2 [0-140]°
float32 ThetaF3 [0-140]°	float32 ThetaF3 [0-140]°
float32 ThetaF4 [0-180]°	float32 ThetaF4 [0-180]°

Como se observa en la tabla 3.4, la estructura tiene campos del mismo nombre en la sección *request* y *response*. Los de la sección *request* serán rellenados por el nodo cliente con los ángulos que se quieren establecer en las articulaciones. En el caso de la sección *response*, serán retornados por el nodo servidor (*bhand_server*) con el ángulo final con el que han quedado las articulaciones una vez se ha ejecutado el movimiento.

- **set_finger**

El servicio permite fijar el ángulo de una única articulación manteniendo la configuración del resto de articulaciones. Para su uso se utiliza el tipo ***Set_Finger_Angle*** que es muy similar al utilizado en el servicio *set_fingers*.

Tabla 3.5: Campos de la variable compartida del servicio set_finger

bhand::Set_Finger_Angle	
request	**response**
string FingerSelect	float32 ThetaF
float32 ThetaF	

- **set_fingers_speed**

Permite configurar la velocidad de movimiento de las articulaciones. Se utiliza una variable compartida del tipo ***Set_Fingers_Speed***.

Tabla 3.6: Campos variable compartida del servicio set_fingers_speed

bhand::Set_Fingers_Speed	
request	**response**
uint16 SpeedF1 [16-4080]	
uint16 SpeedF2 [16-4080]	
uint16 SpeedF3 [16-4080]	
uint16 SpeedF4 [16-4080]	

En este caso el nodo *bhand*_server no tiene que retornar ningún valor al cliente que haga uso del servicio por lo que no hay campos en la sección response.
Los campos de la sección *request* se rellenan con las velocidades que se quiere establecer en cada articulación, siendo 16 la velocidad mínima y 4080 la velocidad máxima.

3.3 Nodo bhand_client

Se trata de un nodo de ejemplo creado para servir como referencia para utilizar los servicios del nodo *bhand_server*. En su código fuente aparece como utilizar los servicios *open* y *set_fingers*.

CAPÍTULO 4

APLICACIÓN DE CAPTURA DE UNA PELOTA EN MOVIMIENTO

4. APLICACIÓN DE CAPTURA DE UNA PELOTA EN MOVIMIENTO

4.1 Introducción

En este capítulo se pretende realizar una aplicación que haga uso de la BarrettHand mediante el paquete *bhand*.

La aplicación consiste en utilizar el brazo robot IRB 120 equipado con la BarrettHand como herramienta, para coger una pelota que realiza un movimiento pendular. Para realizar el seguimiento de la pelota, se utiliza una cámara Kinect.

La comunicación entre el brazo robot y el PC con ROS se hace mediante área local con el uso de un *socket*.

4.2 Arquitectura general de la aplicación

Una vez se libera la pelota en un extremo, ésta comenzará a oscilar. La Kinect permite realizar el seguimiento de la pelota que se encuentra en todo momento en el área de visión del sensor. El programa encargado del seguimiento es un paquete de ROS denominado ***vision*** y realizará las tareas de seguimiento, envío de órdenes al controlador IRC5 del IRB 120 y control de la BarrettHand.

En la figura 4.1 aparece un esquema general de los elementos software implicados en esta aplicación.

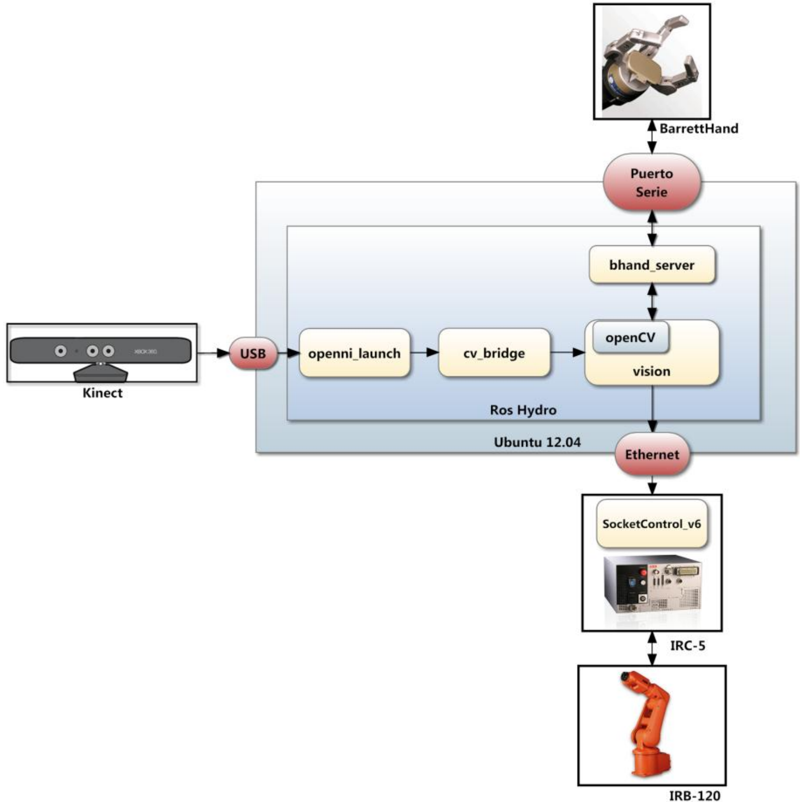


Figura 4.1: Arquitectura Aplicación de captura

A continuación se explica brevemente la función de los distintos elementos de software utilizados (bloques amarillos de la figura 4.1).

- **openni_launch**: Se encarga de comunicarse con la Kinect y obtener las imágenes. Para transferir las imágenes a otros nodos de ROS, utiliza una comunicación por medio de *Topic* (véase apartado 2.1.4). El formato de salida de las imágenes es ROS *Image*.
- **ros_bridge:** Realiza la conversión de imágenes en formato ROS a formato OpenCV.
- **vision**: Nodo principal de la aplicación.
- **bhand_server**: Encargado de la comunicación con la BarrettHand. Sus características y funcionamiento se encuentra detallado en el apartado 3.2.
- **SocketControl_v6**: Aplicación RAPID encargada de la comunicación entre el robot y un PC. Basado en el TFG "Socket based communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station" por Marek Jerzy Frydrysiak.

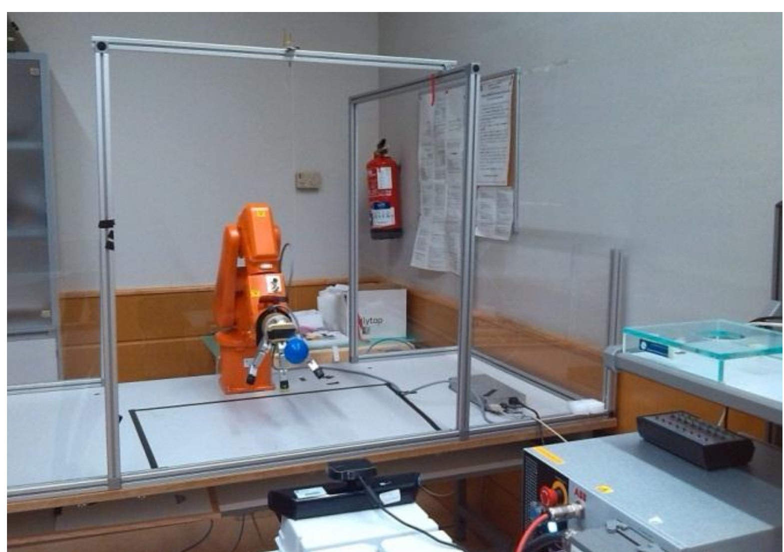


Figura 4.2: Aplicación de captura. Componentes

4.3 Estudio del movimiento pendular

Para poder llevar a cabo el desarrollo de la aplicación, en primer lugar se debe realizar un análisis del movimiento pendular, también llamado movimiento armónico con el fin de familiarizarse con este tipo de movimiento.

Este movimiento se da típicamente cuando se coloca una masa en el extremo de una cuerda inextensible o un eje rígido y su otro extremo queda fijado en un soporte. Al liberar la masa desde cierto ángulo se inicia un movimiento periódico de oscilación de un extremo a otro.

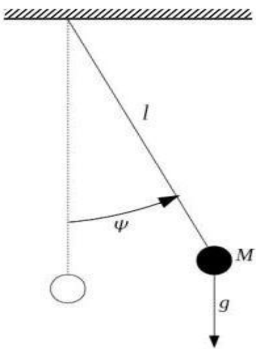


Figura 4.3: Péndulo

A su vez, el movimiento armónico puede ser de dos tipos:

- Simple
- Amortiguado

4.3.1 Movimiento armónico simple

Los parámetros fundamentales que caracterizan este tipo de movimiento son:

- Longitud de la cuerda o del eje (L)
- Aceleración de la gravedad (g)
- Ángulo inicial (ψ)

A partir de estos parámetros y mediante un análisis de las fuerzas presentes [11] se pueden obtener las ecuaciones de posición y velocidad del objeto en función del tiempo:

Periodo (T):
Es el tiempo transcurrido desde que se libera la masa en una posición, hasta que nuevamente vuelve a su posición inicial.
La ecuación 4.1 permite determinar el periodo en función de varios parámetros.

$$T = 2\pi \cdot \sqrt{\frac{L}{g}} \quad (4.1)$$

Posición (θ):

$$\theta(t) = \psi \cdot \cos(\omega \cdot t) \; (4.2)$$

Sustituyendo la expresión 4.3 en la ecuación 4.2 se obtiene la ecuación 4.4 con la que se puede conocer la posición angular de la masa respecto de la vertical, en cualquier instante de tiempo una vez que el movimiento se ha iniciado.

$$\omega = \frac{2\pi}{T} = \sqrt{\frac{g}{L}} \; (4.3)$$

$$\theta(t) = \psi \cdot \cos(\sqrt{\frac{g}{L}} \cdot t) \; (4.4)$$

4.3.2 Movimiento armónico amortiguado

El movimiento armónico amortiguado se presenta en situaciones en las que la amplitud de las oscilaciones se atenúa con el tiempo. Esta atenuación puede aparecer debido a rozamientos o fuerzas externas. En el caso del péndulo utilizado para esta aplicación, el rozamiento aparece en la unión del péndulo con su soporte y también debido a la fricción con el aire.
Esta atenuación dará lugar a que la amplitud de las oscilaciones se reduzca en cada ciclo como se muestra en la figura 4.4, pero la frecuencia de la oscilación se mantendrá aproximadamente constante.

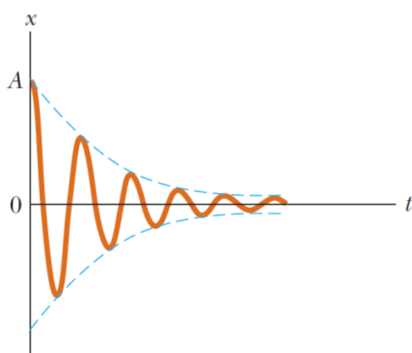


Figura 4.4: Movimiento amortiguado

Este tipo de movimiento está determinado por la ecuación 4.5.

$$\theta(t) = \psi \cdot e^{-\gamma \cdot t} \cdot \cos(w \cdot t) \quad (4.5)$$

Como se puede observar, la ecuación 4.5 solo difiere de la ecuación 4.2 en el término exponencial $e^{-\gamma \cdot t}$. Este término modela la atenuación que sufre la amplitud con el tiempo.
Los sistemas que siguen este comportamiento son habitualmente estudiados en *Ingeniería de Control* y se nombran como **sistemas subamortiguados**.
La constante γ se denomina **factor de amortiguamiento** y establece cómo se atenúa la amplitud conforme va transcurriendo el tiempo.
Particularizando el caso del péndulo, combinando la ecuación 4.5 y la ecuación 4.3 se obtiene la ecuación 4.6 que define el comportamiento de un péndulo con amortiguamiento.

$$\theta(t) = \psi \cdot e^{-\gamma \cdot t} \cdot \cos\left(\sqrt{\frac{g}{L}} \cdot t\right) \quad (4.6)$$

El uso de algunas de estas ecuaciones se justificará en el apartado 4.5.

4.4 Péndulo utilizado en la aplicación

4.4.1 Dimensiones

El péndulo utilizado es el mostrado en la figura 4.5 con una longitud de 0.8 m.
Utilizando la ecuación 4.1 se obtiene qué el periodo de este péndulo será de aproximadamente 1.8 segundos.

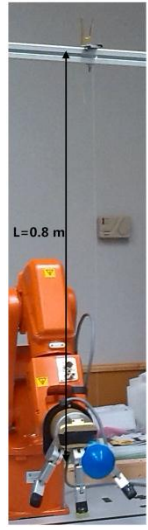


Figura 4.5: Péndulo real

4.4.2 Parámetros del péndulo real

Para conocer el comportamiento del péndulo real y si se aproxima al teórico, se ha realizado mediante la Kinect con el nodo *model* (véase apartado 4.6.3), una serie de medidas de posición mientras el péndulo se mueve libremente. Estas medidas consisten en registrar las posiciones X e Y de la pelota en la imagen (en píxel) con un periodo de muestreo aproximado de 33 ms. Las figuras 4.6 y 4.7 se muestran con la equivalencia entre píxel-distancia ya realizada (véase apartado 5.3.2).

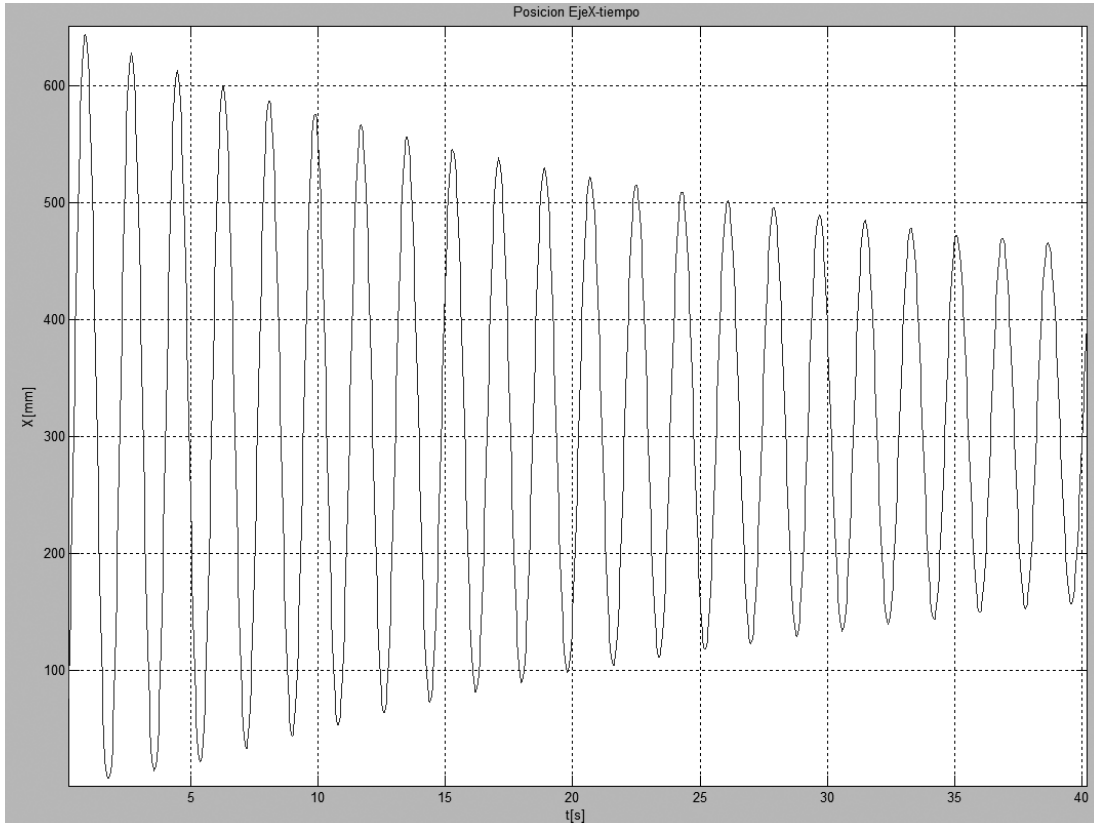


Figura 4.6: Péndulo. Posición eje X-Tiempo

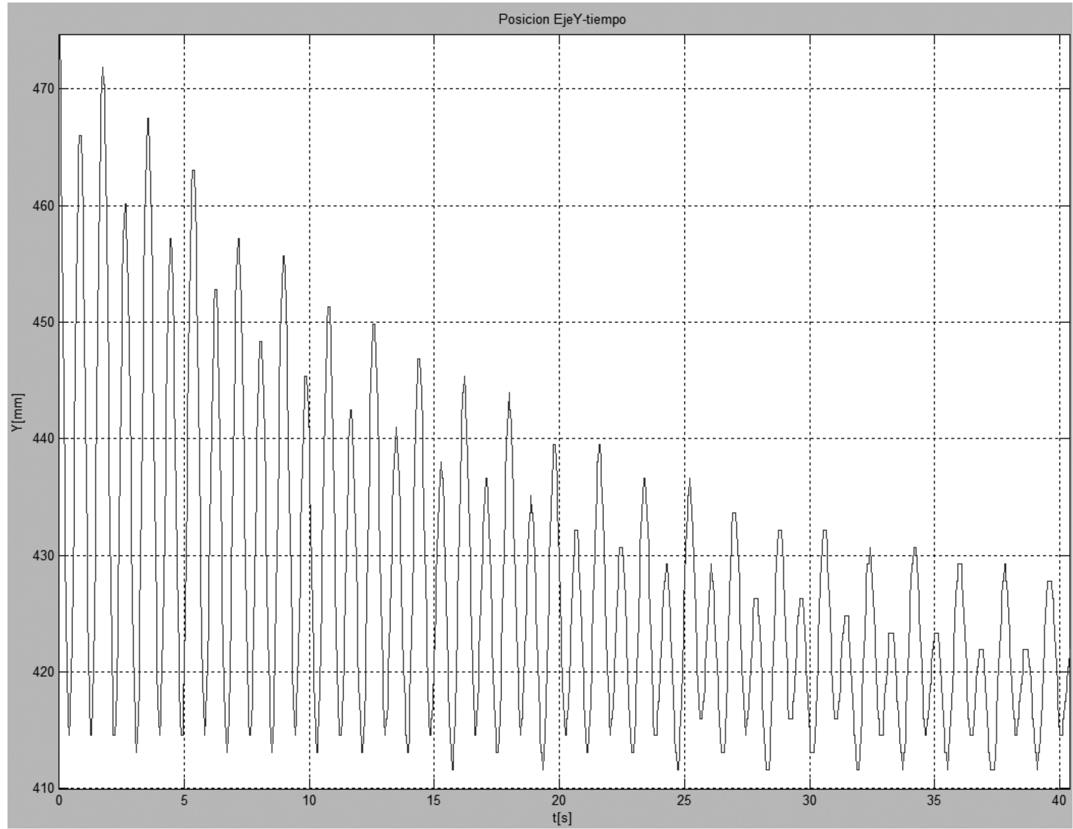


Figura 4.7: Péndulo. Posición eje Y-Tiempo

A partir de estas medidas se ha utilizado el software *Matlab/Simulink* para obtener un modelo del péndulo que permita hacer simulaciones que muestren el efecto de los retardos del sistema.

4.4.3 Modelo del péndulo real en simulink

Para generar un modelo del sistema que pueda ser utilizado en *simulink*, se ha hecho uso de la herramienta *ident* de *Matlab* (figura 4.8).
Esta herramienta permite entre otros, generar **modelos en variables de estado**.

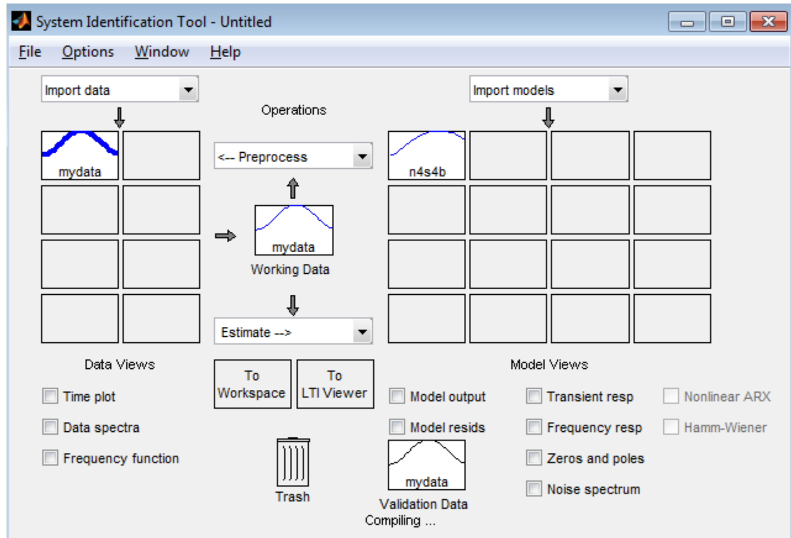


Figura 4.8: System Identification Tool

Para construir el modelo se realizan los siguientes pasos en *Matlab*:

- **Importar el archivo *Input_Output.mat*** que contiene las matrices *input* y *output*.
- **Iniciar la herramienta *ident*** desde el terminal de *Matlab* escribiendo: "init".
- **Cargar los datos de salida del sistema (posición X e Y)**
  - Seleccionar del desplegable "*Import data*" la opción "*Time Domain Data*". Aparecerá la ventana mostrada en la figura 4.9.

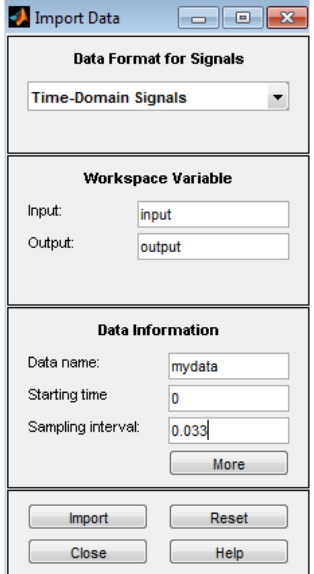


Figura 4.9: System Identification Tool. Importar datos

  - En el apartado "*Workspace Variable*", en la sección "Input" se introduce el nombre de la matriz que ha sido importada anteriormente: "input". En la sección "Output" se introduce la matriz "output".
  - En el apartado "*Data Information*", en la sección "*Sampling Interval*" se introduce el periodo de muestreo con el que se han realizado las medidas. En este caso 0.033s. El tiempo de inicio "*Starting time*" se fija a 0.
  - Para finalizar se presiona "*Import*" cargando los datos en la pantalla principal de *ident*.

- **Generar el modelo en Variables de Estado**
    - Seleccionar del desplegable "*Estimate*" la opción "*Linear Parametric Models*". Aparecerá una ventana como la mostrada en la figura 4.10.

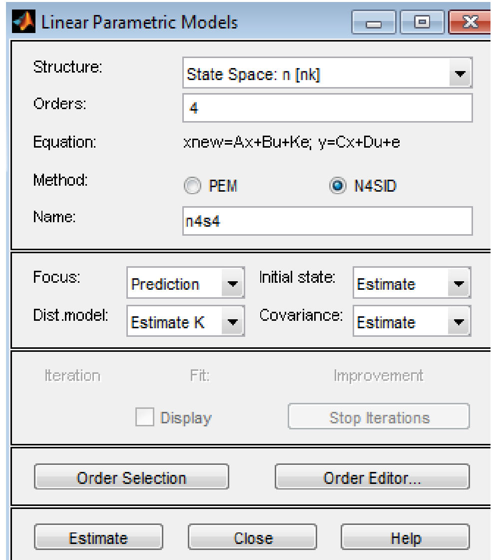


Figura 4.10: System Identification Tool. Parámetros del modelo

    - Modificar la opción *"Initial state"* con el valor "*Estimate*". Mantener el resto de opciones por defecto.
    - Presionar "*Estimate*" para generar el modelo en Variables de Estado.
- **Exportar el modelo generado a Matlab**
    - En la pantalla principal de *ident* aparecerá un modelo en la sección *"Model Views"* con el nombre "n4s4" (por defecto).
      Para poder trabajar con el modelo generado en *Matlab* o *Simulink*, se debe exportar el modelo al espacio de trabajo.
      Arrastrar la figura del modelo al recuadro "*To Workspace*".
      El modelo aparecerá en el espacio de trabajo con un formato *idss*.

Una vez se ha obtenido el modelo se crea el diagrama de *simulink* con el que puede ser utilizado (figura 4.11).

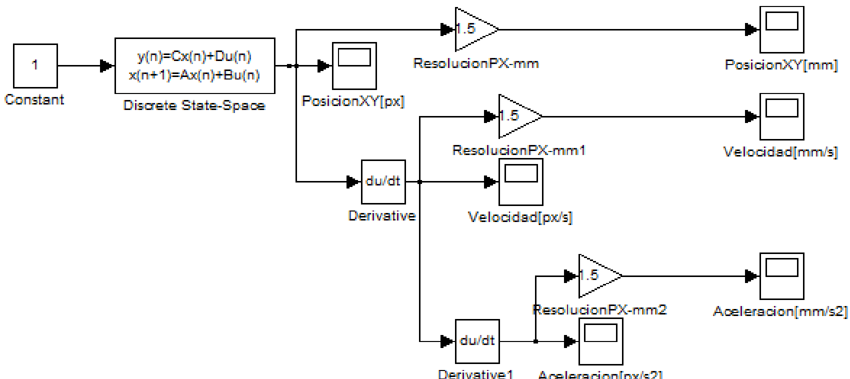


Figura 4.11: Diagrama de bloques Simulink

Este diagrama de bloques permite obtener parámetros importantes del movimiento del péndulo:

- Posición en ejes X e Y

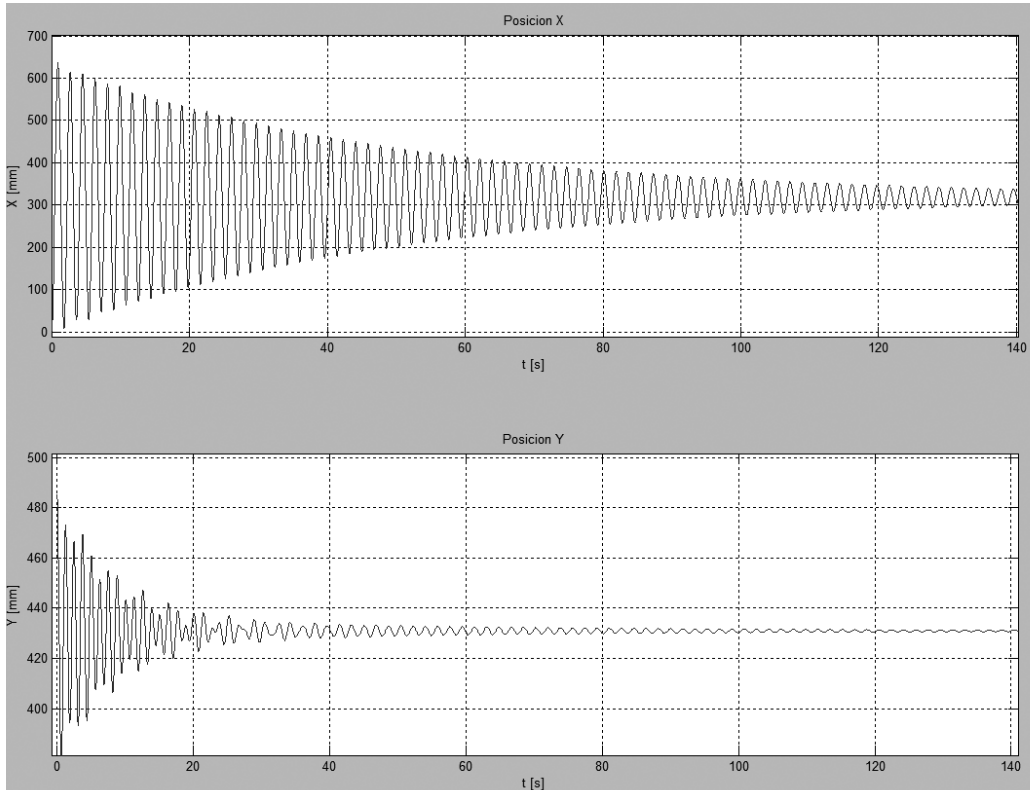


Figura 4.12: Modelo. Posición eje X e Y-Tiempo

- Velocidad en ejes X e Y

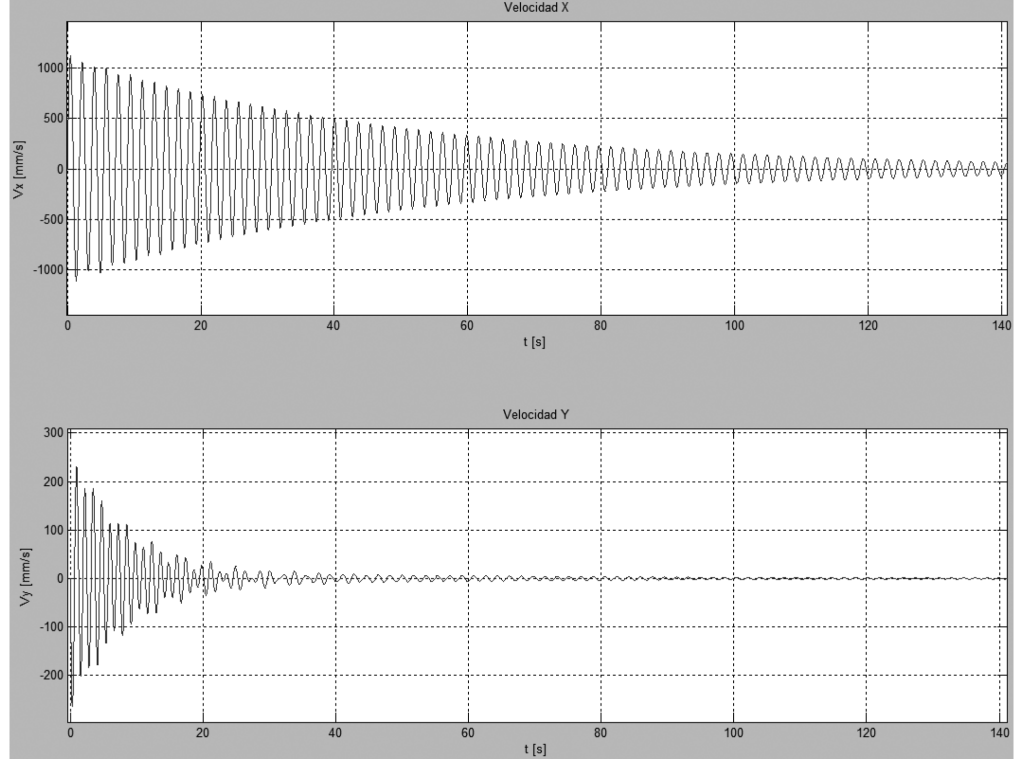


Figura 4.13: Modelo. Velocidades eje X e Y

- Aceleración en ejes X e Y

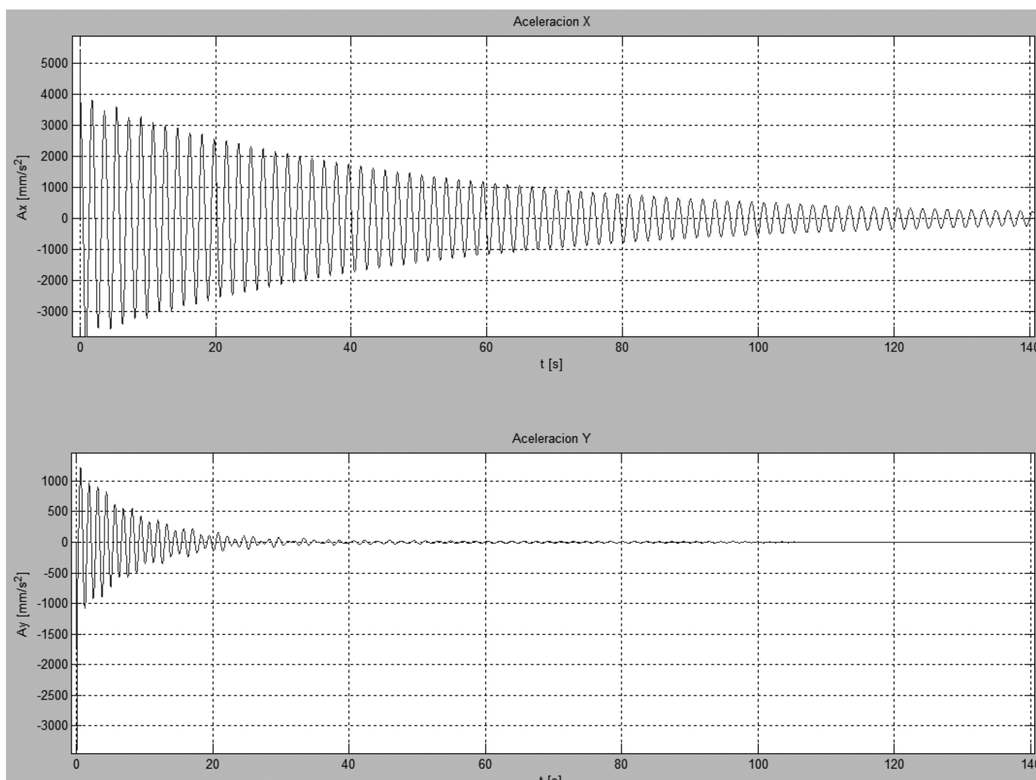


Figura 4.14: Modelo. Aceleraciones eje X e Y

Tabla 4.1: Velocidades y aceleraciones máximas en el péndulo

Velocidad máxima eje X [mm/s]	Aceleración máxima eje X [mm/s^2]
1120	3820

Con estos parámetros se conocen algunas de las características más importantes del movimiento del péndulo.

4.5 Limitaciones del sistema

En este apartado se van a describir las diferentes limitaciones que tienen los diferentes equipos utilizados a la hora de realizar esta aplicación. Estas limitaciones serán fundamentalmente temporales (retardos) y delimitarán las funciones que se pueden llevar a cabo.

El análisis se centrará en los siguientes elementos:

- Brazo robot IRB 120 y controlador IRC5
- Comunicación Ethernet
- PC de control
- Resolución Kinect
- Retardo Kinect

4.5.1 Brazo robot IRB 120 y controlador IRC5

Uno de los factores limitantes en aplicaciones que hacen uso de sistemas mecánicos, como pueden ser brazos robóticos, es el retardo que se da en actuadores eléctrico-mecánicos. Desde que se da una orden al brazo robot (típicamente de posicionamiento) hasta que éste la realiza, transcurre un tiempo que dependerá, entre otros, de la velocidad que permitan los motores, aceleraciones, cálculos de la unidad de control, etc.
Para realizar esta aplicación se ha medido el tiempo que requiere el IRB 120 en posicionarse desde el punto de reposo hasta el punto de recogida de la pelota.
Para ello se han realizado ocho medidas del tiempo que tarda el IRB 120 en ejecutar una instrucción MoveJ [12] moviéndose desde el punto (600, 0, 360)mm hasta el punto (750, 0, 360)mm (**desplazamiento de 150mm**) con una **velocidad de 1650 mm/s**. Las medidas se han hecho desde el programa *RAPID* que se encuentra cargado en el controlador IRC5 utilizando los temporizadores disponibles (precisión de 1ms).
Estas medidas se encuentran en la tabla 4.2.

Tabla 4.2: Retardo MoveJ

	Tiempo de ejecución MoveJ [ms]							
	269	248	269	344	344	344	269	318
Media	300							
σ	41							
Máximo	344							
Mínimo	248							

Hay que tener en cuenta que las medidas realizadas en la tabla 4.2 son válidas para una configuración y un movimiento concreto, por lo que no serían válidas si se modifica la configuración del robot o se utiliza una instrucción de movimiento diferente. Para conocer los detalles de la configuración utilizada en el robot y los comandos utilizados se deberá consultar el código fuente del programa que se ejecuta en el controlador IRC5 (*SocketControl_v6*) y en el PC (vision.cpp).

4.5.2 Comunicación Ethernet

Debido a que la aplicación que se encarga de hacer el seguimiento de la pelota se encuentra en el PC, las órdenes de movimiento que se quieren ejecutar en el brazo robot deben ser enviadas al controlador IRC5 mediante un canal de comunicación. En este caso, mediante Ethernet.
Para comprobar cuál es el retardo que aparece desde que se envía un comando de movimiento vía Ethernet hasta que el robot comienza a moverse se utiliza el siguiente procedimiento:

- Fijar una pelota azul en el efector final del robot como si se tratase de una herramienta.
- Enviar un comando de movimiento al robot vía Ethernet.
- La Kinect registra la imagen hasta que se produce un movimiento de la pelota (cambio de posición).

- Se mide el tiempo transcurrido desde el envío del comando hasta que se detecta el movimiento mediante la Kinect.

En la tabla 4.3 se muestran las 8 medidas de este tiempo de respuesta.

Tabla 4.3: Tiempo de respuesta

Tiempo de respuesta, T_{res} [ms]								
	190	200	180	170	200	190	160	160
Media	181							
σ	16							
Máximo	200							
Mínimo	160							

4.5.3 PC con ROS

Un gran inconveniente de utilizar como sistema operativo Linux con la distribución Ubuntu, es que **no se trata de un sistema operativo de tiempo real**. Debido a esta característica no se pueden asegurar tiempos de respuesta concretos en los diferentes procesos que se están ejecutando en el sistema. Ya que no es posible medir los tiempos de todas las aplicaciones que se ejecutan en el sistema, se han realizado una serie de medidas del tiempo de ejecución de la sección más crítica del nodo *vision* (véase apartado 4.6.2) encargado del seguimiento del objeto y envío de órdenes al robot.
Esta sección es la encargada de realizar la segmentación de color, cálculo de las posiciones de destino para el robot y envío de órdenes.
La sección se ejecuta periódicamente cada vez que la Kinect envía una imagen y es de gran importancia que el tiempo que se tarda en ejecutar la sección no supere el periodo de adquisición de imágenes (Ts en las figuras 4.15 y 4.16). Si se superase, se produciría pérdida de información (figura 4.15) o incluso retardos acumulables en el tiempo (se adquiere y trata una imagen que no es actual), en la figura 4.16.

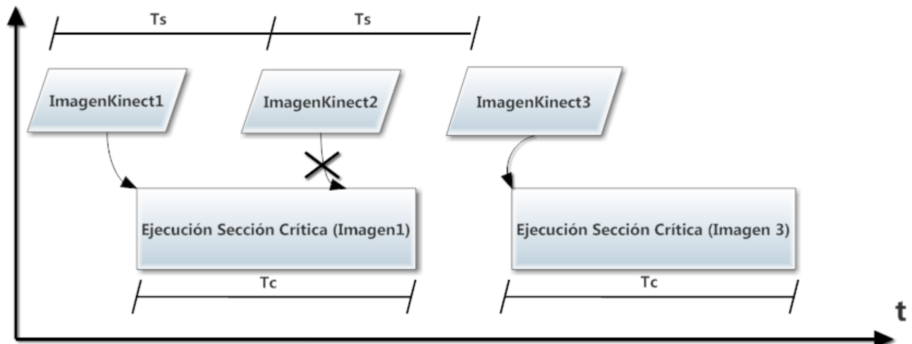


Figura 4.15: Sección crítica. Pérdida de información

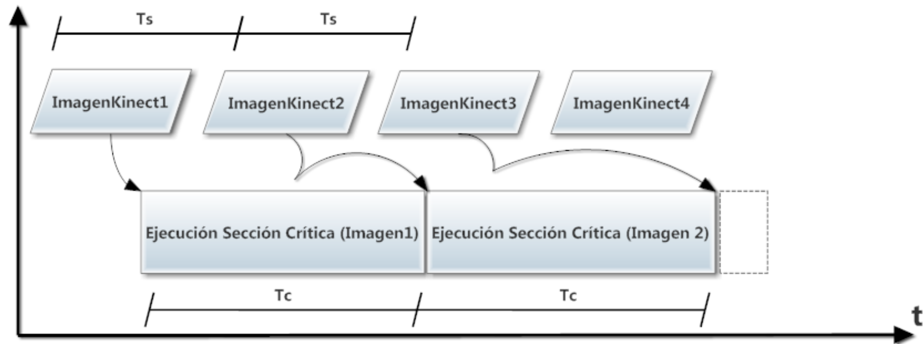


Figura 4.16: Sección crítica. Retardos acumulables

En primer lugar se mide el periodo de adquisición de imágenes para conocer qué tiempo de ejecución no puede superar la sección crítica. La adquisición de imágenes de la Kinect se realiza a 30 imágenes por segundo, dando un Ts de 33.33 ms. Este valor es propio de la Kinect, pero como la adquisición de la imagen es realizada por el nodo *openni_launch* que sirve de intermediario con el nodo *vision*, la temporización se puede ver afectada.

En la tabla 4.4 se encuentran algunas de las muestras de una serie de **300 medidas** del periodo de adquisición de imágenes sobre el nodo *vision.* Se adjuntan su media, desviación típica y valores máximos/mínimos.

Tabla 4.4: Periodo adquisición de imágenes

Periodo de adquisición de imágenes, Ts [ms]									
	52	39	34	37	35	33	32	…	36
Media	33.34								
σ	2.15								
Máximo	52								
Mínimo	29								

Se observa que a pesar de que el tiempo de adquisición medio coincide con el específico de la Kinect, en ocasiones aparecen periodos notablemente más elevados o ligeramente inferiores.

Por último, se debe medir el tiempo de ejecución (Tc en la figura 4.15 y figura 4.16) de la sección crítica para comprobar que sea inferior al periodo de adquisición y de esta forma asegurar que no se den problemas como los mostrados en la figura 4.15 y figura 4.16.
En la tabla 4.5 aparecen varias muestras de las **300 medidas** realizadas junto a sus principales parámetros estadísticos.

Tabla 4.5: Tiempo sección crítica

Tiempo Sección Crítica, Tc [ms]									
	40	34	38	30	33	32	32	…	26
Media	27.33								
σ	2.56								
Máximo	40								
Mínimo	24								

A pesar de que el valor medio de Tc es inferior al valor medio de Ts, se dan casos en los que las muestras de Tc presentan tiempos superiores a las de Ts. Este hecho puede dar lugar a los errores mostrados en las figuras 4.15 y 4.16. Se deben asumir estos problemas debido a la dificultad de conseguir una correcta temporización en un sistema operativo que no es de tiempo real, lo que supondrá una fuente de error en la aplicación.
Además de estos problemas, esta sección genera un retardo en el sistema, ya que hasta que no se ejecute por completo no se enviarán órdenes a la BarrettHand o al IRC5.

4.5.4 Resolución Kinect

A la hora de conocer la posición de un objeto a partir de una imagen capturada por la Kinect, existe cierta relación entre las distancias de la imagen medidas en píxeles y las distancias del mundo real. Esta relación dependerá de diversos factores como pueden ser: tamaño de píxel del sensor de la Kinect, óptica, distancia al objeto, etc. En este caso, para encontrar la equivalencia se ha realizado una serie de medidas experimentales obteniendo la resolución de cada píxel.

$$Resolución = 1.47\ mm/px$$

Este valor de resolución tan solo será válido para las condiciones concretas de esta aplicación y se vería modificado si varía la distancia entre la cámara y el péndulo.

4.5.5 Retardo Kinect

Como cualquier sistema de adquisición de imágenes, la Kinect presenta un retardo en la captura de la imagen. Este retardo es función de gran cantidad de parámetros: tiempo de exposición, retardos de la electrónica del sensor, tratamiento de la imagen, retardo en la transferencia vía USB, etc. El retardo da lugar a que la imagen que disponemos en el PC no se corresponda con la realidad. En la figura 4.17 se ilustra el efecto de forma gráfica.

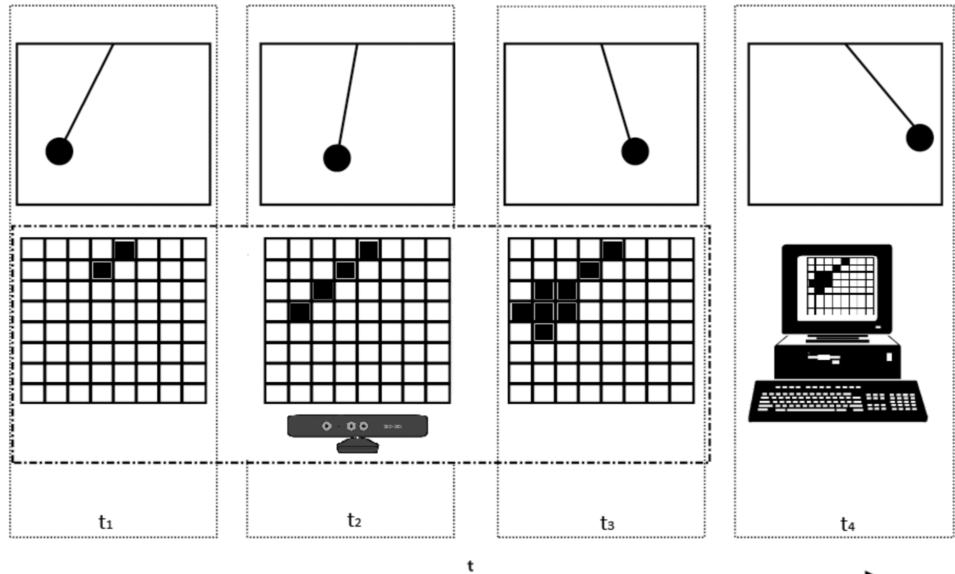


Figura 4.17: Retardo Kinect

Este retardo es desconocido y tan solo se puede asegurar que su valor es inferior a 33.33 ms (periodo de muestreo de la Kinect). Debido a la dificultad en su estimación, este retardo no se tendrá en cuenta en el análisis, pero producirá cierto error en esta aplicación.

4.5.6 Retardo general del sistema

Teniendo en cuenta los retardos individuales mencionados anteriormente, se establece que el retardo general medio en esta aplicación es:

$$T_{General} \approx T_{res} + T_{MOVEJ} + T_c = 181 + 300 + 27 = 508\, ms$$

4.6 Paquete vision

4.6.1 Introducción

Como se ha explicado anteriormente, el paquete ***vision*** contiene los elementos necesarios que permiten realizar una aplicación que consiste en capturar una pelota que realiza un movimiento pendular. Está compuesto por dos nodos:

- *vision*
- *model*

4.6.2 Nodo vision

Es el nodo encargado de realizar las diferentes acciones necesarias y coordinar los distintos elementos hardware que forman parte de la aplicación.
En la figura 4.18 se muestran las acciones que se realizan.

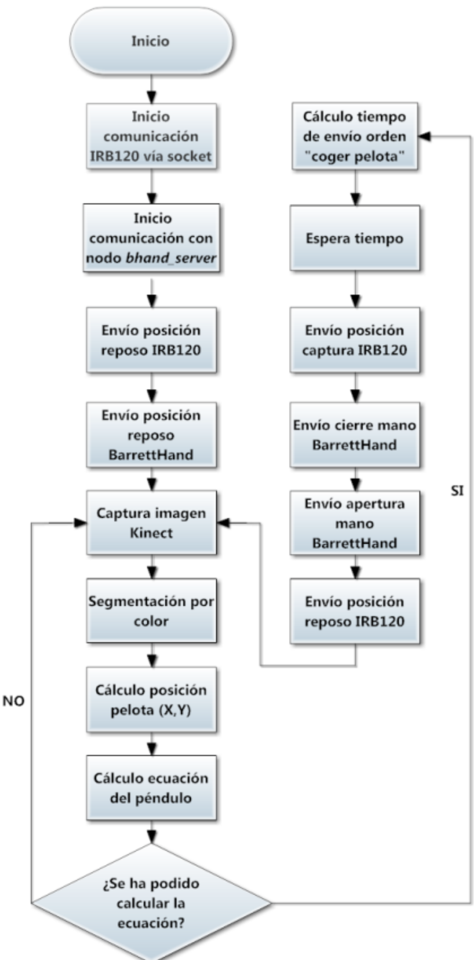


Figura 4.18: Aplicación de captura. Diagrama de flujo

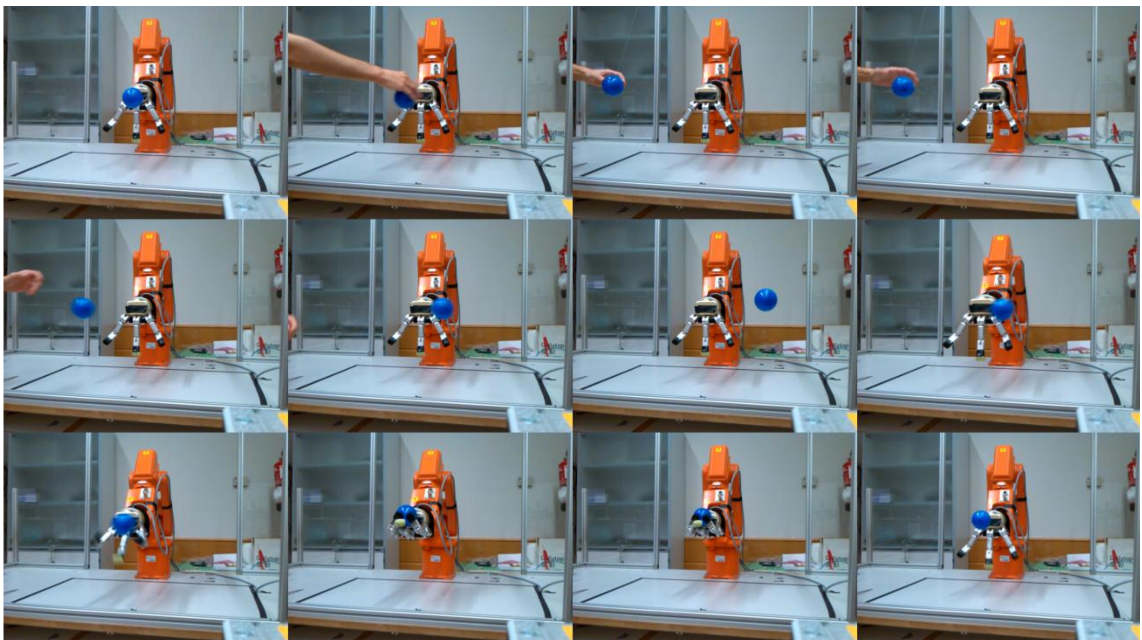


Figura 4.19: Aplicación de captura. Secuencia

A continuación se detallan los diferentes módulos de este nodo.

A) Comunicación por socket

El término *socket* referido a comunicación, hace referencia a un mecanismo de comunicación que generalmente se utiliza para poner en contacto dos o más ordenadores que se encuentran en una red.
Fundamentalmente existen dos tipos diferentes de *socket:*

- **UDP** (User Datagram Protocol)
- **TCP** (Transmission Control Protocol)

Estos dos protocolos permiten transmitir paquetes de datos entre dos o más equipos. La principal diferencia entre estos dos tipos reside en que en el protocolo TCP se asegura que la trasmisión de un paquete de datos ha llegado de forma correcta, en el caso de no se reciba correctamente se retransmitirá tantas veces como sea necesario, mientras que en el UDP no se asegura que el paquete de datos llegue sin errores.
A pesar de que a primera vista puede parecer que el protocolo TCP es el más adecuado para un sistema de tiempo real, presenta ciertas desventajas sobre el UDP:

- **Mayor latencia**: Generando un incremento del retado en la comunicación.
- **Empaquetado de datos**: El protocolo TCP tiende a agrupar varios paquetes de datos antes de realizar el envío de éstos. Con esto se consigue reducir el tráfico de datos en la red, pero incrementa el tiempo de envío.

La principal ventaja del TCP es que se asegura de que el paquete ha llegado de forma correcta al destino, lo cual puede ser necesario en ciertas aplicaciones de tiempo real en las que prima más que el dato llegue al destino frente al tiempo requerido en el envío.

A la hora de **comunicarse con el IRC5**, se utiliza una comunicación por red de área local. En el IRC5 se ejecutará el programa *RAPID SocketControl_v6* basado en el TFG "Socket based communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station" por Marek Jerzy Frydrysiak.
Este programa utiliza un protocolo de comunicación por medio de un socket TCP actuando como un servidor.
El envío de la información se tiene que hacer con paquetes que contienen las órdenes y configuraciones que deben ser utilizadas por el robot. El PC que quiera comunicarse con el robot deberá utilizar la estructura de estos paquetes para que el programa que se está ejecutando en el IRC5 sea capaz de interpretarlos.
En la figura 4.20 hay un ejemplo de uno de estos tipos de paquetes de datos. En este caso corresponde a una instrucción de orientación de la herramienta.

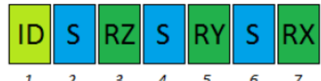


FIGURE 2.13 TCP orientation datagram structure

ID	**Name of frame:** **Data type:** **Value:** **Remarks:**	Identifier of datagram Byte 2 Contains the identifier of the system orientation datagram.
S	**Name of frame:** **Data type:** **Values:** **Remarks:**	Sign of value Byte 0 or 1 $sgn(R_{n+1}) = \begin{cases} -R_{n+1} \ if \ S_n = 0 \\ R_{n+1} \ if \ S_n = 1 \end{cases}$ Carries a sign identifier of the next rotation frame.
R_{zyx}	**Name of frame:** **Data type:** **Values:** **Remarks:**	Absolute rotation value Byte 0 – 255 An absolute value of the EulerZYX rotation around the chosen axis in reference to the local TCP Cartesian system. Nevertheless, the unrotated local system corresponds to the global one (the parallel axes). Thus, the robot in the default position (the joints' values set to 0) has the TCP's local system orientation equals to (0, 90, 0). Note: the order of the rotation is not negligible.

TABLE 2.4 Description of the TCP orientation datagram

Figura 4.20: Paquetes socket. Extracto de [13]

En el nodo *vision* se ha configurado la conexión vía *socket* para estar en consonancia con la configuración del *socket* del IRC5. Los principales campos configurados son:

- Dirección IP y puerto del servidor (IRC5)
- Tipo de protocolo: TCP
- **Desactivación del algoritmo Nagle**: Como se ha comentado en líneas anteriores, en la comunicación por TCP se produce un empaquetamiento de los datos antes de realizar un envío, enviando datos agrupados. Para ello se hace uso de este algoritmo. En esta aplicación se busca que el envío de los datos se haga de la forma más rápida posible, por lo que este algoritmo resulta perjudicial y se desactiva.

B) Seguimiento de objetos

Para realizar el seguimiento de la pelota se utilizan las diferentes funciones de tratamiento de imágenes que facilita la librería OpenCV. Se dan dos etapas:

- **Segmentación por color**: El objetivo de esta etapa es el de, a partir de una imagen en color, obtener una nueva imagen en la que tan solo aparezcan aquellos objetos que tienen un color determinado. De esta forma se puede separar el objeto que se quiere seguir (la pelota) del resto de objetos del entorno.
  El color de la pelota que se ha utilizado es el azul debido a que hay pocos objetos en el área de trabajo que contengan este color.
  Para el filtrado de la imagen se fijan los siguientes umbrales para cada uno de los tres canales en el espacio de color HSV.

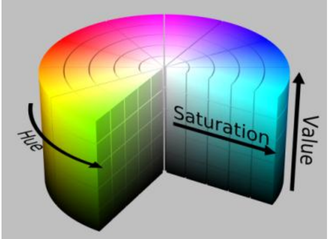


Figura 4.21: Espacio de color HSV

$$110 < \boldsymbol{hue} < 126$$
$$104 < \boldsymbol{saturation} < 255$$
$$0 < \boldsymbol{value} < 255$$

Los píxeles cuyos canales se encuentren en estos umbrales serán de color azul y por lo tanto se les asigna un valor de 255 (píxel blanco) mientras que los que se encuentren fuera de los umbrales serán de otros colores y se asignarán con valor 0 (píxel negro).
En la figura 4.22 aparece la secuencia completa utilizada para pasar de la imagen original obtenida por la Kinect (RGB) a una imagen binaria en la que tan solo aparece la pelota.

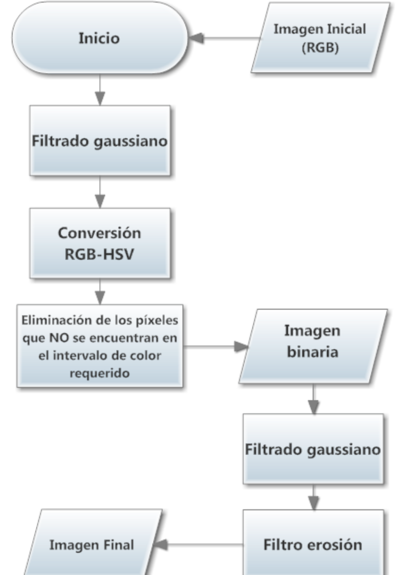


Figura 4.22: Segmentación. Diagrama de flujo

Los filtrados gaussianos y de erosión son utilizados para eliminar ruido de la imagen.
El resultado de esta segmentación se puede observar en la figura 4.23.

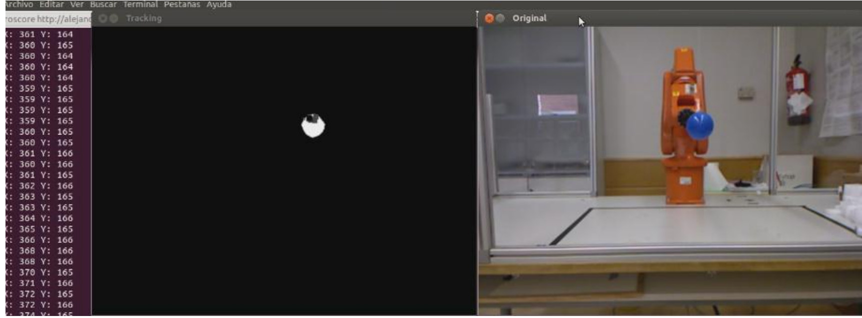


Figura 4.23: Segmentación. Ejemplo

- **Localización:** Una vez que en la etapa anterior se ha obtenido la imagen binaria en la que solo aparece la pelota, se debe obtener las coordenadas (X,Y) que identifican en qué lugar de la imagen se encuentra.
  El método utilizado se basa en el cálculo del **primer momento de área**.
  El primer momento de área es una magnitud que se puede calcular sobre superficies planas y permite localizar el **centroide** de un área respecto a un eje. La ecuación 4.7

calcula el primer momento de área de un área A respecto al eje X y la ecuación 4.8 lo hace respecto al eje Y.

$$S_x = \int_A y \cdot dA \quad (4.7)$$

$$S_Y = \int_A x \cdot dA \quad (4.8)$$

Siendo ‘x’ e ‘y’ las distancias del elemento de área al eje Y y X respectivamente.

Para aclarar el concepto se propone un ejemplo en el que se busca identificar el centroide de una figura sencilla compuesta por cinco píxeles.

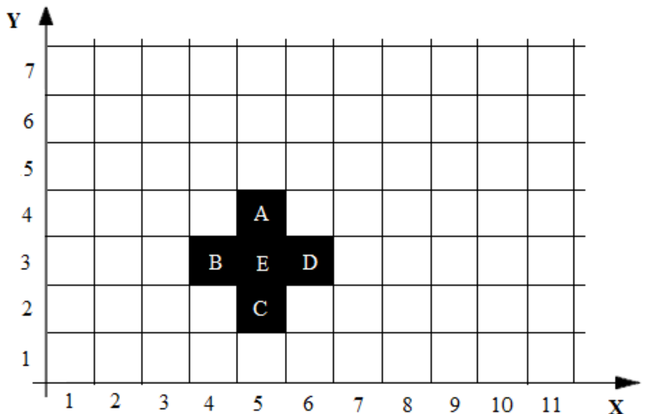


Figura 4.24: Momentos de área. Ejemplo

Dada la figura 4.24 se busca calcular los momentos de área respecto al eje X y el eje Y. Aplicando las ecuaciones 4.7 y 4.8 se obtiene:

$$S_x = A_A \cdot d_{x-A} + A_B \cdot d_{x-B} + A_C \cdot d_{x-C} + A_D \cdot d_{x-D} + A_E \cdot d_{x-E}$$
$$S_y = A_A \cdot d_{y-A} + A_B \cdot d_{y-B} + A_C \cdot d_{y-C} + A_D \cdot d_{y-D} + A_E \cdot d_{y-E}$$

$$S_x = \overbrace{1 \cdot 4}^{A} + \underbrace{1 \cdot 3}_{B} + \overbrace{1 \cdot 2}^{C} + \underbrace{1 \cdot 3}_{D} + \overbrace{1 \cdot 3}^{E} = 15$$

$$S_y = \overbrace{1 \cdot 5}^{A} + \underbrace{1 \cdot 4}_{B} + \overbrace{1 \cdot 5}^{C} + \underbrace{1 \cdot 6}_{D} + \overbrace{1 \cdot 5}^{E} = 25$$

La librería OpenCV contiene una serie de funciones para el cálculo del momento de área y del área total de la superficie, por lo que en el programa *vision.cpp* no es necesario hacerlo manualmente.

Una vez se conocen los momentos de área, el centroide del área se puede calcular aplicando las ecuaciones 4.9 y 4.10.

$$Xc = \frac{S_y}{A} \quad (4.9)$$

$$Yc = \frac{S_x}{A} \quad (4.10)$$

Siendo “Xc” e “Yc” las coordenadas del centroide y “A” el área total de la superficie.

Continuando con el ejemplo anterior, si se aplican las ecuaciones 4.9 y 4.10, se calcula el centroide de la superficie que en este caso tiene un área A de 5 px^2.

$$Xc = \frac{S_y}{A} = \frac{25}{5} = 5$$

$$Yc = \frac{S_x}{A} = \frac{15}{5} = 3$$

Centroide = (5,3)

De esta forma, el nodo *vision* calcula las coordenadas del centroide. Hay que tener en cuenta que para que el proceso se realice correctamente, tan solo debe aparecer un objeto del color elegido (azul) en la escena, ya que si hay varios objetos, el procedimiento calcularía el centroide suponiendo que los dos objetos son dos partes distintas de un mismo objeto. Por esta razón cobra gran importancia que en la etapa de segmentación se separe el resto de objetos de la pelota.

C) Captura de la pelota

El paso final de la aplicación consiste en coordinar el brazo robot junto a la BarrettHand para capturar la pelota.

El brazo robot se encuentra inicialmente en una posición de reposo en la que permite el paso de la pelota y una vez la pelota pasa frente a esta posición, se produce el avance del brazo y el cierre de la mano (figura 4.25).

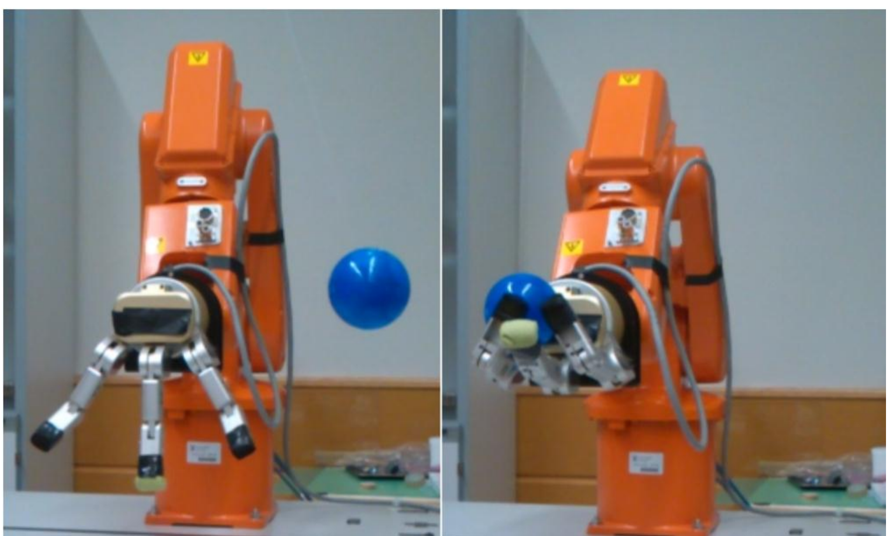


Figura 4.25: Secuencia captura de pelota

Las posiciones del eslabón final del robot mostradas en la figura 4.25 son las siguientes:

Reposo: (600, 0, 360) mm
Captura: (750, 0, 360) mm

En esta etapa un factor muy importante es el retardo del sistema. Ya que la pelota posee una velocidad, el envío de las órdenes de captura (tanto a la BarrettHand como al IRC5) se debe realizar con antelación, antes de que la pelota pase frente a la mano.
Para poderse anticipar al paso de la pelota, se deben tener en cuenta dos tiempos: tiempo de tránsito de la pelota y el retardo del sistema.

- **Tiempo de tránsito de la pelota**: El tiempo de tránsito de la pelota es el **tiempo que transcurre desde que se libera la pelota hasta que pasa frente la posición de reposo** del robot por **segunda vez consecutiva**.
  Es necesario conocer este tiempo debido a que la orden de captura se ha de enviar antes de que transcurra el tiempo. Para calcularlo, mientras el nodo se está ejecutando, se capturan imágenes de la pelota analizando el movimiento y obteniendo algunos de los parámetros característicos que definen el péndulo (amplitud máxima e instante en el que se inicia el movimiento).
- **Retardo del sistema:** El retardo del sistema es el tiempo desde que se envía la orden de captura hasta que el brazo robot se coloca en la posición de captura y se ejecuta el movimiento de cierre de la mano.

Como se muestra en la figura 4.19 la captura de la pelota no se realiza en el primer instante en el que la pelota pasa por el punto de captura, si no que se hace en el retorno. Esto se debe a que el retardo del sistema imposibilita cogerla en el primer cruce.

Para el cálculo del tiempo de tránsito de la pelota, se utiliza la ecuación 4.4 despejando la variable tiempo ‘t’, obteniéndose la ecuación 4.11.

$$t = \pi \cdot \sqrt{\frac{L}{g}} + \frac{\cos^{-1}\left(-\frac{x-c}{A}\right)}{\sqrt{\frac{g}{L}}} \quad (4.11)$$

Siendo:
x: Posición sobre el eje X en la que se quiere coger la pelota
c: Posición central respecto donde se produce la oscilación
A: Amplitud de la oscilación.
g: Aceleración de la gravedad
L: Longitud del péndulo

Los parámetros ‘c’ y ‘L’ se obtienen por medio de calibración; ‘A’ es obtenido en tiempo de ejecución y ‘x’ es configurable, pudiendo elegir la posición sobre el eje X dónde se quiere coger la pelota.

En la figura 4.26 se resume el procedimiento de captura mediante un diagrama de flujo.

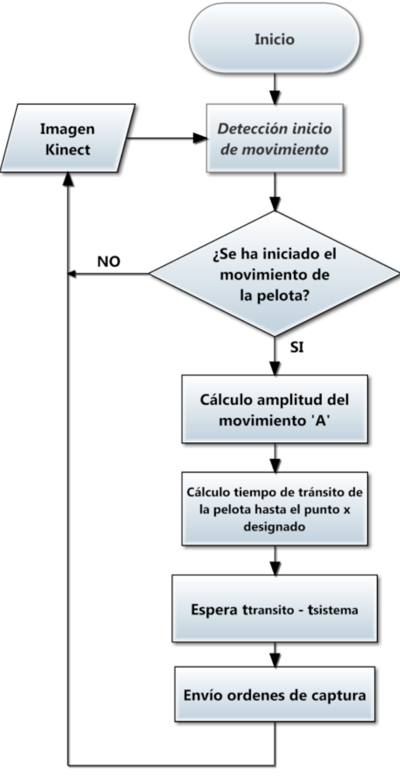


Figura 4.26: Predicción

4.6.3 Nodo model

Este nodo se trata de un derivado del nodo *vision* y su objetivo es registrar de forma continuada los tiempos y las posiciones en las que se localiza la pelota en cada imagen obtenida por la Kinect. Su funcionamiento es similar al nodo *vision* pero se elimina la sección de captura.

Con este nodo se ha registrado la serie de datos utilizados en el apartado 6.4 "Péndulo utilizado en la aplicación" para la creación del modelo y se han obtenido posiciones de calibración.

CAPÍTULO 5

APLICACIÓN DE SEGUIMIENTO

5. APLICACIÓN DE SEGUIMIENTO

5.1 Introducción

En esta aplicación se busca obtener un sistema que permita al IRB 120 hacer un seguimiento en tiempo real de un objeto que se mueva de forma arbitraria (el efector del robot seguirá al objeto). Para ello se utilizará, además del IRB 120, la cámara Kinect que permite la detección del objeto y un PC que será el encargado del procesamiento de las imágenes y la comunicación con el IRB 120.

Esta aplicación no es válida para el seguimiento de objetos que se mueven a velocidades elevadas como en el caso del péndulo utilizado en la aplicación anterior (véase capítulo 4) debido a las limitaciones del sistema (véase apartado 4.5).

5.2 Arquitectura general de la aplicación

Para esta aplicación, se ha diseñado un paquete de ROS llamado ***tracking*** que se encargará del procesado de las imágenes obtenidas por la Kinect y del envío de órdenes de movimiento al robot.

En la figura 5.1 se muestra un esquema general de los programas utilizados en la aplicación.

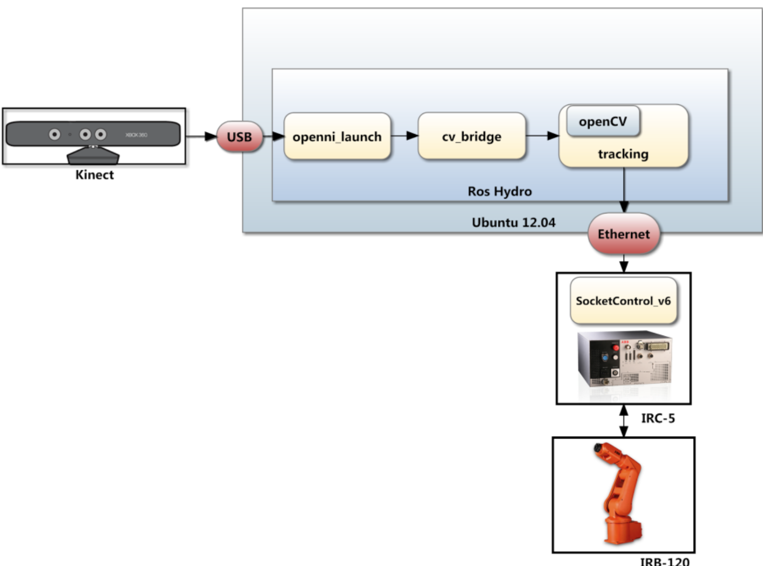


Figura 5.1: Arquitectura aplicación seguimiento

Como se observa en la figura 5.1 la arquitectura del sistema utilizada para esta aplicación es muy similar a la del capítulo 4 y se reutilizarán algunos de los módulos diseñados anteriormente (comunicación por socket, segmentación por color, etc.).

5.3 Paquete tracking

5.3.1 Introducción

El paquete tracking está formado por un nodo con el mismo nombre ***tracking***.

5.3.2 Nodo tracking

Este nodo llevará a cabo las acciones necesarias para esta aplicación. En la figura 5.2 se muestran la serie de acciones que realiza.

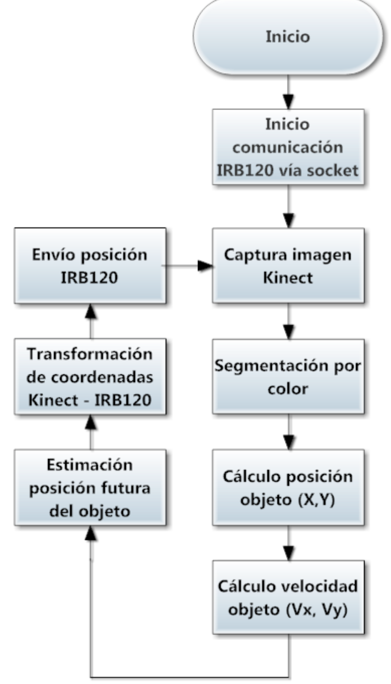


Figura 5.2: Aplicación seguimiento. Diagrama de flujo

Figura 5.3: Aplicación seguimiento. Secuencia

Ya que gran parte del código utilizado en este nodo es idéntico al del nodo *vision* tratado en el capítulo 4, se omitirán los módulos de "Comunicación por socket" y "Seguimiento de objetos". A continuación se detallan dos módulos nuevos utilizados para esta aplicación.

A) Predicción de la posición futura del objeto

Como se expuso en el capítulo 4, existe un retardo no despreciable en la comunicación entre el PC y el robot (T_{RES}). Debido a este retardo, se debe buscar una solución que minimice en la medida de lo posible el error de posición que sufre el robot al hacer el seguimiento.

Como solución se ha optado por realizar un **predictor de posición** que estime dónde se va a encontrar el objeto en un instante futuro, para así, enviar la posición futura del objeto en lugar de la actual. De esta forma se evita enviar una posición al controlador IRC5, que en el momento de recibirse se encontraría desactualizada, ya que en el tiempo de envío de la posición el objeto ha continuado moviéndose.

El predictor diseñado utiliza la velocidad del objeto (V_{Actual}) y su posición (X/Y_{Actual}) y aplicando las ecuaciones del MRU (ecuaciones 5.1 y 5.2), calcula la posición futura en la que se podría encontrar el objeto.
Las velocidades son calculadas a partir de la diferencia de posición del objeto en dos imágenes consecutivas de la Kinect.

$$X_{Futura} = X_{Actual} + Vx_{Actual} \cdot T_{RES} \quad (5.1)$$

$$Y_{Futura} = Y_{Actual} + Vy_{Actual} \cdot T_{RES} \quad (5.2)$$

En la figura 5.4 se muestra un ejemplo gráfico del funcionamiento del predictor.

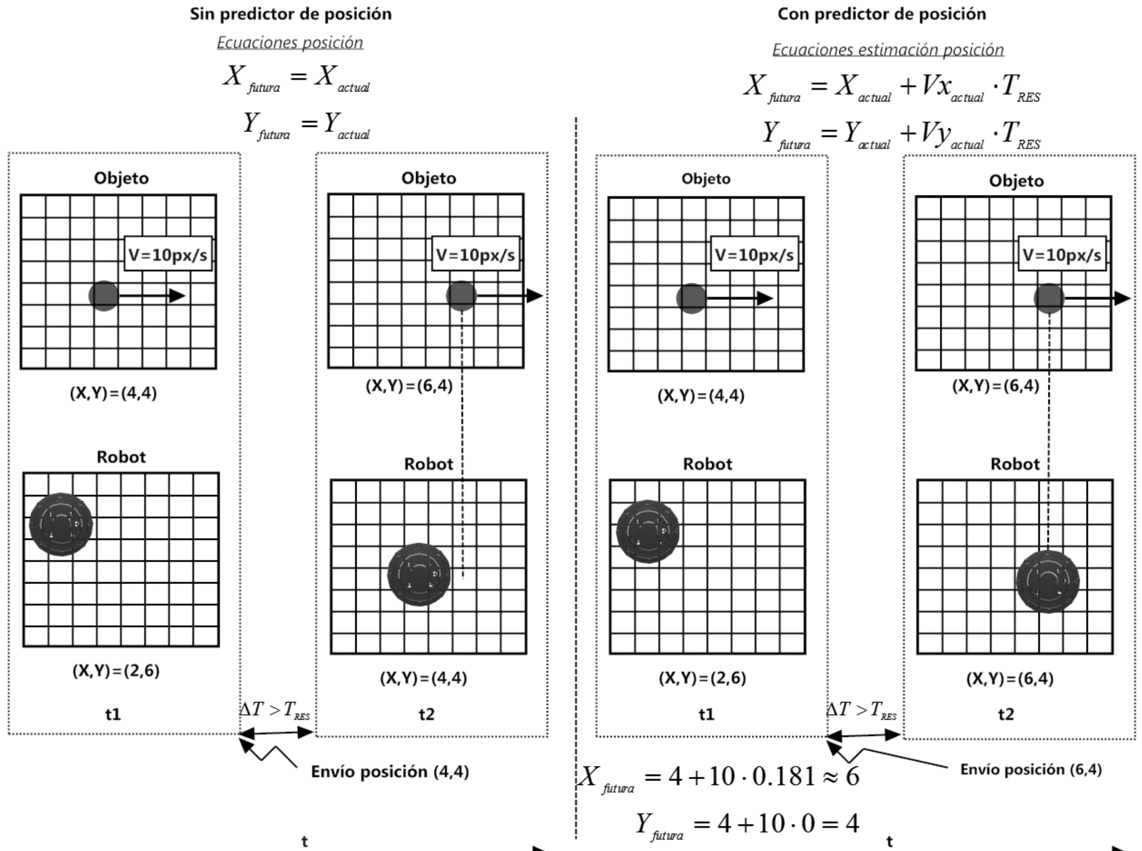


Figura 5.4: Predictor de posición

Esta predicción de la posición futura del objeto también presenta ciertos inconvenientes:

- No tiene en cuenta el tiempo de traslación del robot. Esto produce errores mayores conforme aumenta la diferencia en la distancia que hay entre el robot y el objeto (a mayor distancia el robot requerirá más tiempo en trasladarse).
- El movimiento del objeto no es uniforme. En el caso de que el movimiento del objeto no siga las ecuaciones del MRU la predicción de la posición futura también contendrá un error. Este error variará en función de las características del movimiento.
- Los retardos en la comunicación no son constantes. En las ecuaciones de predicción se ha supuesto que el tiempo de respuesta T_{RES} (aproximadamente el tiempo de comunicación) es constante. Como se muestra en la tabla 4.3 estos tiempos presentan variaciones dando lugar a errores de posición durante el seguimiento.

A estos inconvenientes hay que añadir el hecho de que no se pueden enviar las posiciones del objeto en cada imagen que captura la Kinect ya que se saturaría la aplicación RAPID que se ejecuta en el IRC-5. Por ello, **se enviará la posición de una de cada seis imágenes** (muestreo de posición de aproximadamente 0.2 segundos).

B) Transformación de coordenadas Kinect – IRB 120

En esta aplicación el seguimiento del objeto se realiza detectando su posición sobre la imagen obtenida por la Kinect. Al utilizar el sensor de imagen de la Kinect y no el de profundidad, el seguimiento se hará en dos dimensiones. Para el envío de las órdenes de movimiento al robot se debe conocer cómo se relacionan las coordenadas de la Kinect y las del robot.
Como se muestra en la figura 5.5 los sistemas de referencias no coinciden.

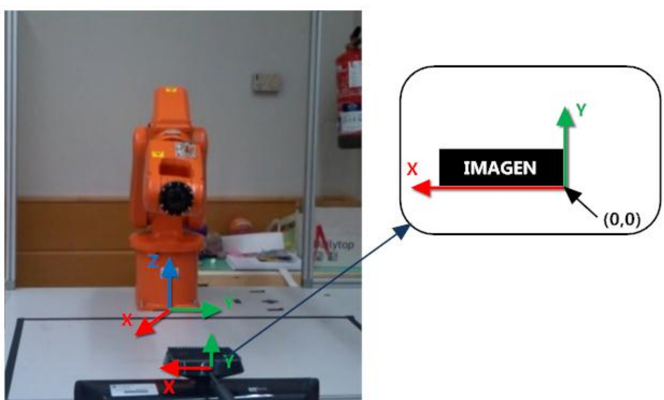


Figura 5.5: Sistemas de coordenadas IRB 120 y Kinect

Para buscar la relación entre las coordenadas del robot y la Kinect hay que tener en cuenta que el eje X de la Kinect se corresponde al eje Y del robot (con distinto sentido), mientras que el eje Y de la Kinect se corresponderá con el eje Z del robot. La posición sobre el eje X del robot se mantendrá fija.
Las ecuaciones que relacionan las coordenadas se han obtenido mediante un proceso de calibración. El proceso consiste en fijar una pelota azul (objeto detectable) al eslabón final del robot y realizar la siguiente secuencia:

- Mover manualmente el robot a una posición
- Registrar las posiciones (Y,Z) del robot y las posiciones (X,Y) del objeto detectado por la Kinect
- Repetir la secuencia

Con la serie de parejas de posiciones $(Y,Z)_{robot}$ y $(X,Y)_{kinect}$ se aplica un ajuste por mínimos cuadrados obteniendo las ecuaciones 5.3, 5.4 y 5.5.

$$X_{robot} = 790 \quad (5.3)$$

$$Y_{robot} = -0.829 \cdot X_{kinect} + 181.17 \quad (5.4)$$

$$Z_{robot} = 0.779 \cdot Y_{kinect} + 5.87 \quad (5.5)$$

CAPÍTULO 6

APLICACIÓN DE CLASIFICACIÓN DE OBJETOS

6. APLICACIÓN DE CLASIFICACIÓN DE OBJETOS

6.1 Introducción

En este capítulo se describirá el diseño e implementación de una aplicación encargada de clasificar y manipular objetos. Para este fin, se ha utilizado un sistema de visión compuesto por una Kinect, la cual servirá para capturar imágenes de la zona de trabajo. La información recibida por parte de la Kinect es utilizada para mover y posicionar el brazo robot ABB IRB 120, que será capaz mediante el uso de la BarrettHand BH8-262, de coger diferentes objetos que clasificará en función de sus características.

El diseño y puesta en marcha de esta aplicación se ha realizado con la colaboración de varios estudiantes de Grado. Si es necesario obtener información más detallada de cada parte se debe consultar los TFG: "**3D recognition and pose estimation using VFH descriptors**" de **Alberto Lázaro Enguita** [14]; "**Trajectory planning for IRB 120 robotic arm using ROS**" de **Javier Moriano Martín** [15] y el presente TFG "**Control de la mano robótica BarrettHand BH8-262 en entorno ROS**" de **Alejandro Gómez Rubio**.

6.2 Sistema de reconocimiento de objetos y estimación de posición

El sistema de visión es el encargado de capturar el entorno utilizando un filtro de posición para evitar la detección de objetos que se encuentran fuera del área de trabajo. Los datos obtenidos por medio de la Kinect son utilizados para construir la PCL de la región de interés. Una vez la PCL ha sido filtrada, se aplica una extracción por agrupamiento para obtener los diferentes objetos. A partir de cada objeto obtenido mediante la extracción, se aplica un algoritmo de reconocimiento de objetos para comprobar si el objeto pertenece a las clases de objetos predefinidas.

Este sistema de clasificación se basa en comparar los coeficientes obtenidos con un patrón con el fin de discernir el tipo de objeto.

Además de la clasificación de objetos, el sistema debe proporcionar las dimensiones de los objetos, su posición y su orientación (en el caso de ser necesario) para poder cogerlos correctamente.

La posición de los objetos se encuentra definida en un sistema de coordenadas cartesianas. El cálculo de la orientación se realiza para los objetos que son cilíndricos o paralelepípedos, mientras que para las esferas no es necesario.

El número de parámetros que deben ser calculados para definir los distintos tipos de objetos varía según su clase. Por ejemplo, las esferas solo necesitan el radio, mientras que los cilindros deben ser definidos por el radio y su altura.

La estimación de las posiciones de los objetos es llevada a cabo mediante la Kinect y para poder proporcionar las coordenadas pertinentes al robot IRB 120 es necesario aplicar un cambio del sistema de referencia que se realiza por medio de traslación y rotación.

Para realizar el control del sistema robótico se utiliza la clase de mensaje *moveit_msgs::CollisionObject* que contendrá los coeficientes obtenidos de los diferentes

objetos presentes en la escena. Estos mensajes serán publicados en un topic de ROS previamente creado.

6.3 Control de la BarrettHand BH8-262

Para coger los diferentes objetos se emplea la BarrettHand BH8-262. Se trata de una mano robótica de propósito general compuesta por cuatro articulaciones independientes. Se encontrará conectada al último eslabón del robot permitiendo coger los diferentes objetos una vez el robot se encuentre en la posición adecuada.

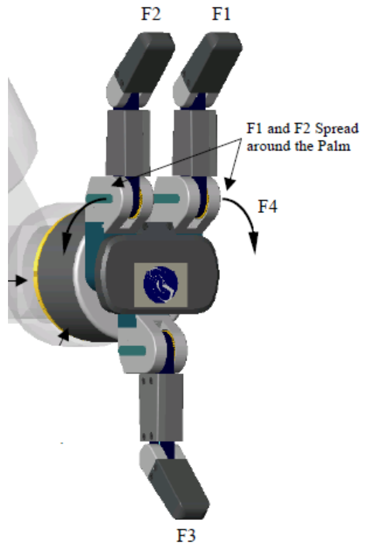


Figura 6.1: Articulaciones BarrettHand

Para el uso de la BarrettHand se emplean los servicios de ROS. Se trata de un método de comunicación entre aplicaciones que funcionan bajo ROS. La estructura y fundamento de este tipo de comunicación se explica a continuación.
Una aplicación denominada habitualmente como servidor, proporciona una serie de servicios a terceras aplicaciones (clientes). Estos servicios consisten en un conjunto de acciones que realizará el servidor. Para hacer uso de estos servicios, el cliente puede utilizar una variable compartida con el servidor. Con esta variable se proporciona una comunicación bidireccional entre el cliente y el servidor. La variable está compuesta por dos campos: request y response.

Tabla 6.1: Tipos de campos en variables compartidas

request	response
float32 C_Variable1	string S_Variable1
int8 C_Variable2	float32 S_Variable2
uint32 C_Variable3	float32 S_Variable3
…	…

La sección request es escrita por la aplicación cliente para transmitir datos a la aplicación servidor. Por otro lado, la aplicación servidor utiliza estos datos para realizar sus acciones y escribe los resultados en la sección response en el caso de que sea necesario entregar una respuesta al cliente. Finalmente el cliente puede leer el campo response para comprobar los resultados retornados por el servidor.

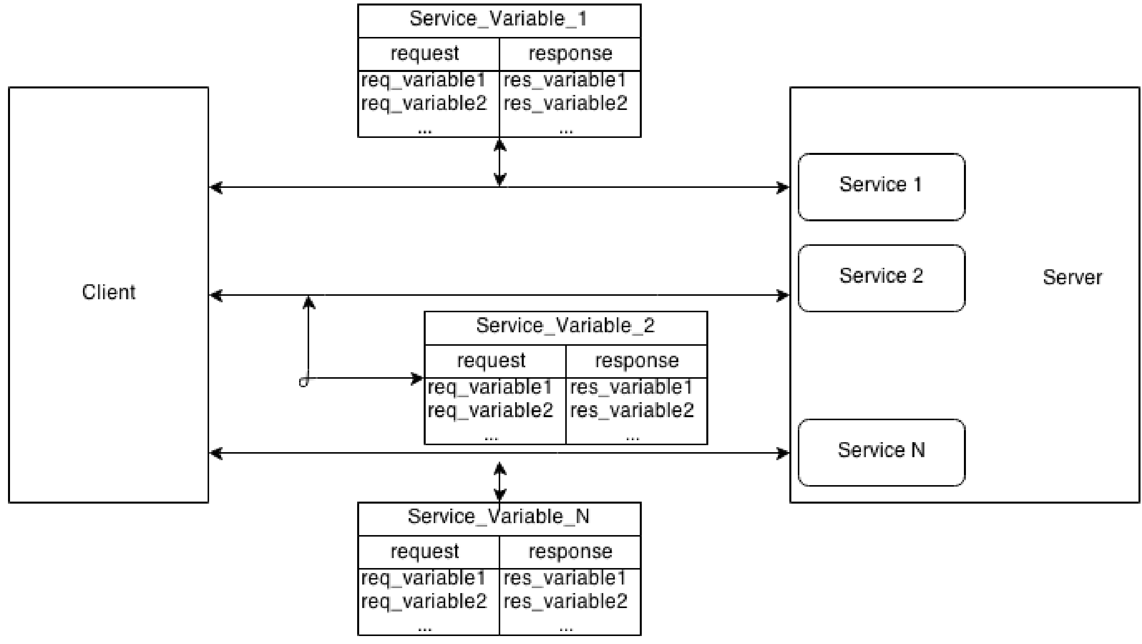


Figura 6.2: Estructura servicios

El control de la BarrettHand por medio de ROS se realiza mediante un paquete denominado *bhand*. Este paquete contiene un nodo de ROS (*bhand_server*) encargado de realizar el control de la mano robótica a bajo nivel: enviar comandos de inicialización, establecer velocidades, fijar ángulos de las articulaciones, etc.
La aplicación se ejecuta en el sistema y provee a otras aplicaciones de ROS de una serie de servicios con los que se puede realizar un control a alto nivel de la BarrettHand. La lista de servicios es la siguiente:

- OpenHand
- CloseHand
- Set_Fingers_Angles
- Set_Finger_Angle
- Set_Fingers_Speed

Los servicios utilizados para agarrar objetos son *set_fingers* y *set_fingers_speed*.

6.3.1 Servicio set_fingers

El servicio ***set_fingers*** permite fijar los ángulos de las cuatro articulaciones disponibles en la mano robótica, para ello se utiliza el miembro de la clase *bhand* llamado *Set_Fingers_Angles*. Este miembro está compuesto por la siguiente estructura:

Tabla 6.2: Variable compartida del servicio set_fingers y rango de articulaciones BarrettHand

bhand::Set_Fingers_Angles	
request	**response**
float32 ThetaF1	float32 ThetaF1
float32 ThetaF2	float32 ThetaF2
float32 ThetaF3	float32 ThetaF3
float32 ThetaF4	float32 ThetaF4
Rango ThetaF1_2_3: [0-140]	Rango ThetaF4: [0-180]

Como se observa en la tabla 6.2, el miembro tiene una estructura típica de servicios, compuesta por una sección *request* y otra *response*.

La sección *request* será rellenada por la aplicación cliente (*attach*) con los ángulos que se quieren establecer en las articulaciones. Una vez el cliente envía la petición de servicio la aplicación servidor se comunicará mediante el puerto serie con la mano robótica ejecutando las instrucciones.

Hay que destacar que no se puede asegurar que las articulaciones alcancen el ángulo deseado, ya que, por ejemplo, si se produce una colisión de algún dedo con un objeto se producirá una parada de dicha articulación de forma automática.

Para poder conocer el ángulo final de las articulaciones una vez se ha ejecutado la instrucción de movimiento, la aplicación servidor escribirá en el campo *response* los ángulos finales que han alcanzado las cuatro articulaciones. Estos ángulos son leídos por la aplicación cliente y son utilizados para representar la posición de los dedos en el entorno gráfico RVIZ.

6.3.2 Servicio set_fingers_speed

El servicio ***set_fingers_speed*** es utilizado para configurar la velocidad de desplazamiento de las articulaciones. La estructura utilizada tan solo tiene campos *request* donde se configura la velocidad de movimiento de cada articulación (tanto para la apertura como para el cierre).

Tabla 6.3: Variable compartida del servicio set_fingers_speed y rango de velocidades BarrettHand

bhand::Set_Fingers_Speed	
request	**response**
uint16 SpeedF1	
uint16 SpeedF2	
uint16 SpeedF3	
uint16 SpeedF4	
Rango SpeedF1_2_3_4: [16-4080]	

6.4 Control del IRB 120

El control del IRB 120 se lleva a cabo por medio de las librerías MoveIT! y el framework ROS. Para realizar este control se utilizan dos nodos: *attach* encargado de la comunicación con el robot y la BarrettHand y *object_attach* encargado de recibir la información de los diferentes objetos para poderlos representar.

De forma añadida, se crea y representa un modelo 3D del robot IRB 120 y de la BarrettHand en el simulador RVIZ. La configuración de la mano robótica se actualizará cada vez que el nodo de la BarrettHand retorne los ángulos de las articulaciones.

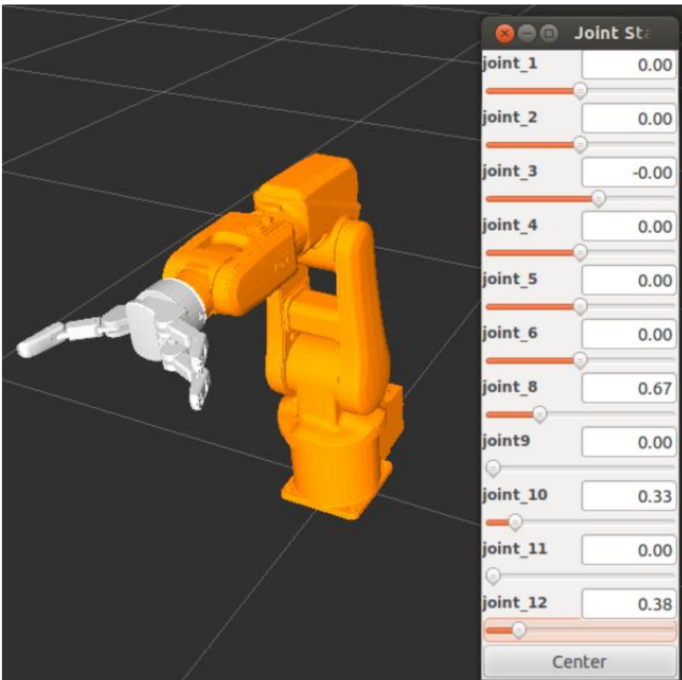


Figura 6.3: Modelo IRB 120 con BarrettHand

6.4.1 Nodo 1: object_attach

El nodo recibe la información obtenida por el sistema de visión. El objeto de mayor tamaño es representado en color azul. La información de este objeto se envía al nodo *attach*, el cual se encarga de que el robot capture el objeto y lo deposite en la posición correcta de acuerdo con su clase. El resto de los objetos se representarán en color verde añadiéndole una zona de seguridad. No se permiten colisiones entre el robot y los objetos verdes.

6.4.2 Nodo 2: attach

Como se ha explicado en el apartado 6.4.1, el nodo *attach* controla el movimiento del robot y envía las órdenes al nodo de control de la BarrettHand *bhand_server*. El programa realiza los siguientes pasos:

- Se mueve el robot a una posición inicial para permitir al sistema de visión capturar adecuadamente toda la escena. Posteriormente el nodo *object_attach* actualizará la escena simulada en función de la información que reciba.
- Una vez el nodo *attach* recibe la posición del objeto de mayor tamaño, el brazo robot se posicionará en esta posición. Mientras que el robot realice el movimiento, el nodo *object_attach* detendrá la publicación. De esta forma se consigue que la simulación no se actualice mientras el robot se encuentra en el campo de visión de la *Kinect*.
- Finalmente, el robot depositará el objeto que se ha cogido en la posición designada que corresponda a la clase particular del objeto. Una vez el robot se localice en el punto en el que se tiene que depositar el objeto, se enviará la orden de soltar el objeto a la mano robótica y el nodo *object_attach* volverá a actualizar la escena.

La forma de coger los objetos dependerá de su clase:

- Paralelepípedo: Para coger un paralelepípedo se mueve el robot sobre el objeto. Una vez se encuentra encima del objeto, se configura el sexto eje del robot para posicionar la mano de forma perpendicular respecto al paralelepípedo. A continuación el robot desciende de forma lineal hasta alcanzar la posición definida para coger el objeto. Alcanzada esta posición se envía la orden de cerrar los dedos de la mano robótica.
- Cilindro: Para coger cilindros se posiciona el robot sobre un objeto y se desciende verticalmente. Se configura el ángulo entre los dedos uno y dos en 30º una vez el robot alcanza la mejor posición para agarrar el cilindro. Para finalizar, la mano robótica cierra los dedos agarrando el objeto.
- Esfera: Para agarrar una esfera el procedimiento es el mismo que en el caso del cilindro.

Figura 6.4: Aplicación de clasificación. Situación inicial y final

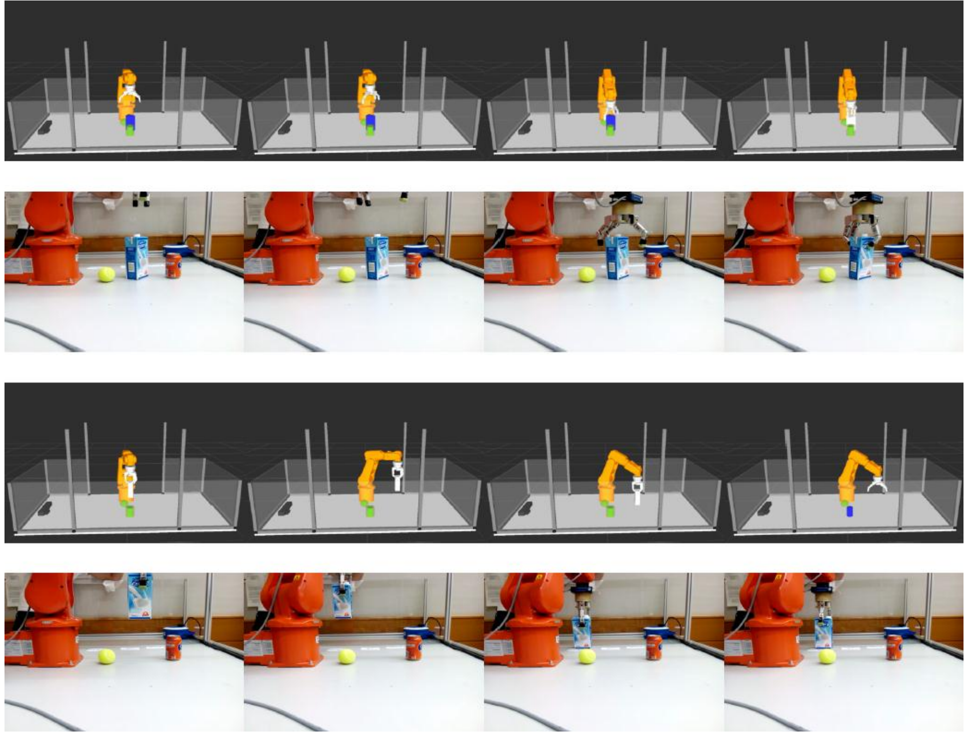


Figura 6.5: Robot posicionando una caja. La caja es detectada cómo el objeto más grande y se selecciona para que el robot la coja. La mano se orienta de forma perpendicular a la caja y coge la caja depositándola en la posición definida. Simulación con RVIZ (primera y tercera fila). Imagen desde la posición de la Kinect (segunda y cuarta fila)

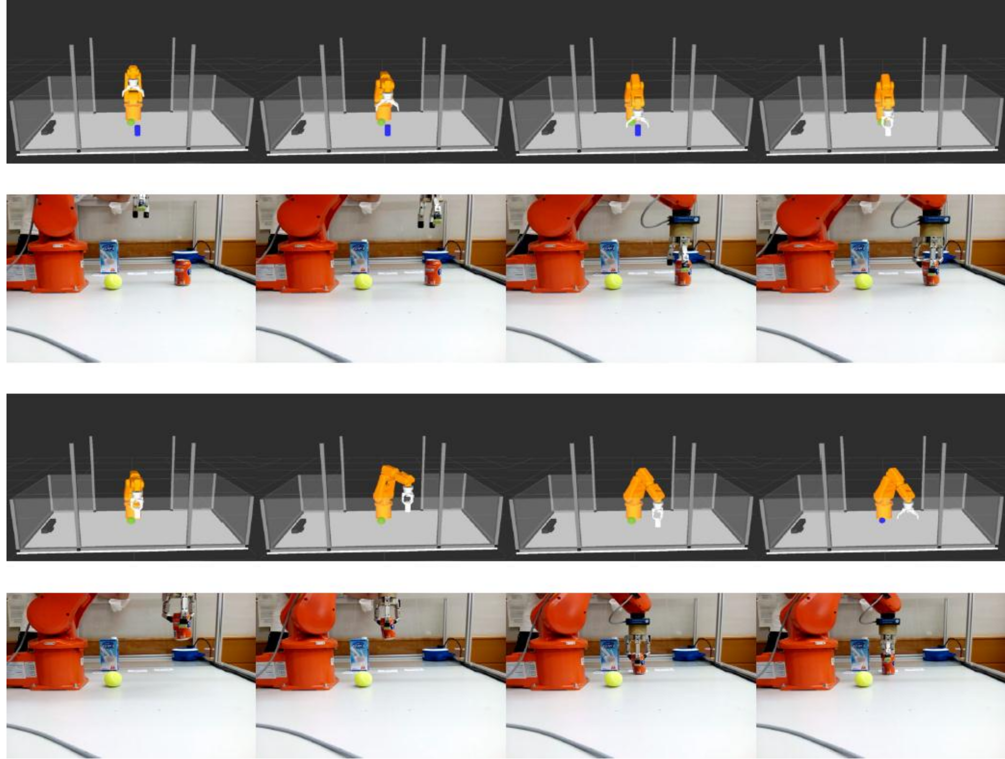


Figura 6.6: Robot posicionando una lata. La lata es detectada cómo el objeto más grande y se selecciona para que el robot la coja. La mano rota los dedos 1 y 2 para agarrar la lata. Simulación con RVIZ (primera y tercera fila). Imagen desde la posición de la Kinect (segunda y cuarta fila)

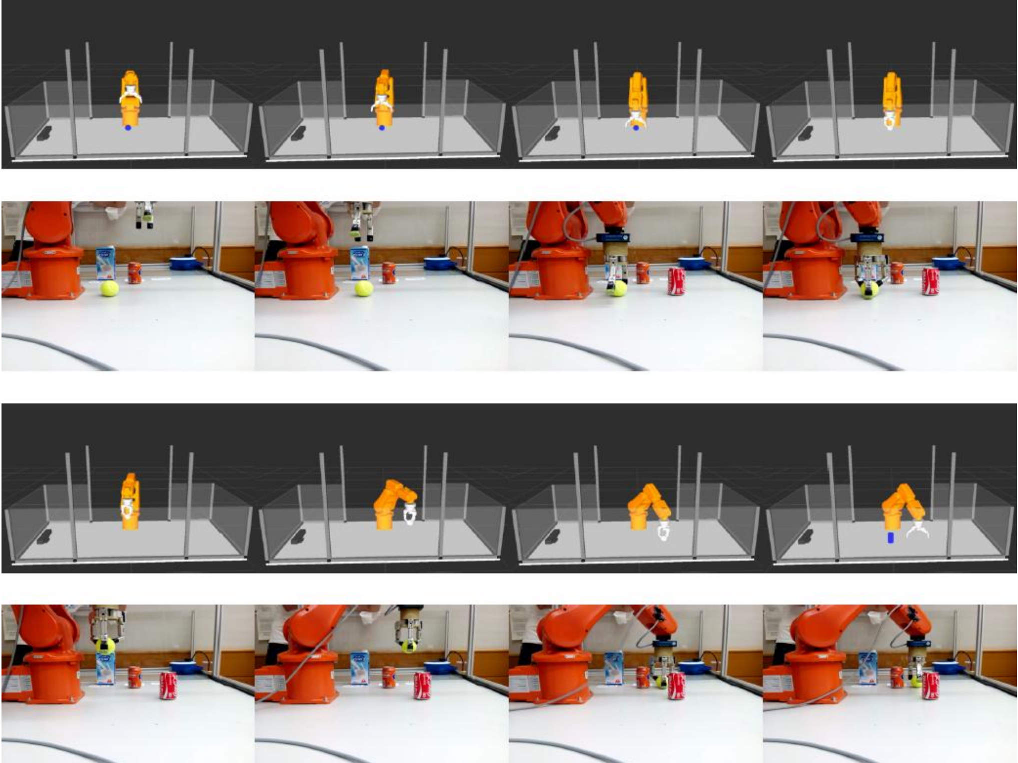


Figura 6.7: Robot posicionando una pelota. La pelota es detectada como el objeto más grande (el último restante). La mano rota los dedos 1 y 2 para agarrar la pelota. Simulación con RVIZ (primera y tercera fila). Imagen desde la posición de la Kinect (segunda y cuarta fila)

CAPÍTULO 7

INTRODUCCIÓN A GRASPIT!

7. INTRODUCCIÓN A GRASPIT!

7.1 Introducción

En las aplicaciones tratadas anteriormente en las que se ha hecho uso de la BarrettHand, es tarea del diseñador realizar la configuración de las articulaciones de la mano robótica en función de las características del objeto que se va a coger (forma, dimensiones, etc.). Este proceso tiene un carácter experimental en el que hay que comprobar cuál de las múltiples configuraciones posibles es la más adecuada para agarrar el objeto.
Actualmente puede resultar muy interesante que este procedimiento sea realizado de forma automática mediante un ordenador por medio de simulación. Una de las aplicaciones que existen para este fin es ***GraspIt!***.

Graspit! Es una herramienta creada por el grupo *"Robotics Group"* perteneciente al departamento de Ciencias de la Computación de la Universidad de Columbia. Uno de sus principales propósitos es permitir a las diferentes manos robóticas coger objetos de forma automática calculando las posibles configuraciones articulares.
En este capítulo se presentarán algunos ejemplos del funcionamiento de esta aplicación mediante el uso de las herramientas de GraspIt! para el fin que se ha descrito anteriormente.

7.2 Archivos de configuración de objetos y robots

Para trabajar con GraspIt! contamos con una interfaz visual que nos permite visualizar el entorno de trabajo. En este entorno se podrán situar varios tipos de objetos:

- **Robots**: Dispositivos equipados con articulaciones y elementos móviles. Pueden ser desde brazos robóticos completos hasta grupos de eslabones sencillos. En este caso el modelo de robot que va a ser utilizado es la BarrettHand.
- **Objetos**: Son aquellos elementos que van a ser manipulados por los robots.
- **Obstáculos**: Son objetos que se encuentran en el área de trabajo (una mesa, paredes, una lámpara, etc.).

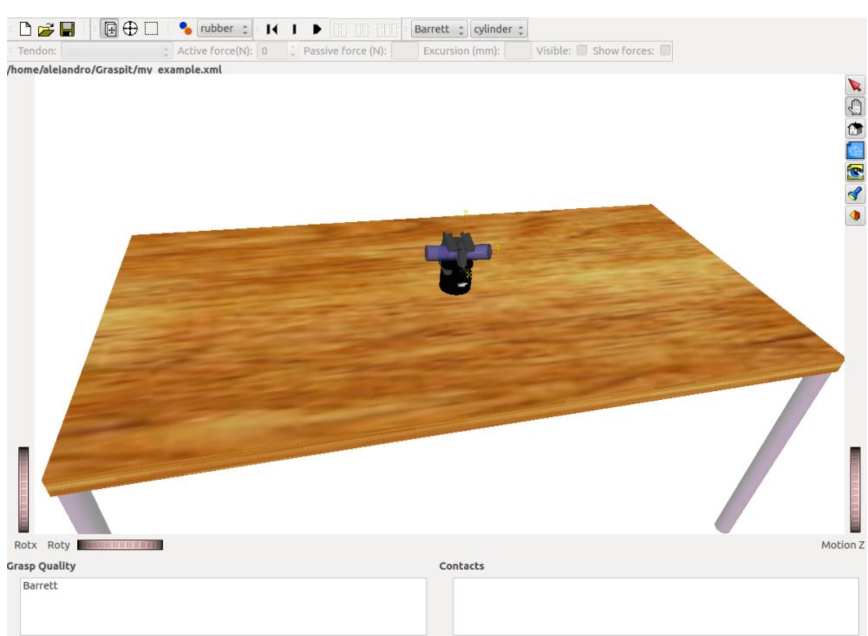


Figura 7.1: Entorno de GraspIt!

Para definir los diferentes elementos indicados anteriormente que pueden aparecer en la escena, se utiliza un **fichero de definición** con formato ***.xml*** en el que se configuran parámetros como: masa, material, coeficientes de fricción, tipo de articulaciones, rangos de las articulaciones, etc. Estos ficheros se encuentran en la carpeta *models* del directorio de GraspIt! y hacen uso de archivos de geometría (extensión *.iv*, Open Inventor) para definir su aspecto visual.
Para más información acerca de este tipo de ficheros se debe consultar el capítulo 4 del manual de usuario de GraspIt! [16].

Otro de los archivos más importantes es el **fichero de mundo**. Este archivo que también tiene un formato .xml, contiene los elementos que se han cargado en el entorno. En la figura 7.1 aparece la BarrettHand (robot), un cilindro (objeto) y la mesa (obstáculo). En su contenido se definen las posiciones de los diferentes elementos, sus orientaciones, los ángulos de las articulaciones de los robots, etc.

A continuación se explicaran los principales campos de este tipo de archivo ya que serán necesarios para planificar la captura de un objeto. Para ello se parte del archivo de mundo que corresponde a la figura 7.1.

```
<?xml version="1.0" ?>
<world>
        <obstacle>
                <filename>///home/alejandro/Graspit/models/obstacles/table.xml</filename>
                <transform>
                        <fullTransform>(+1 +0 +0 +0)[+4.79625 +90.3154 +0]</fullTransform>
                </transform>
        </obstacle>
        <graspableBody>
                <filename>/home/alejandro/Graspit/models/objects/cylinder.xml</filename>
                <transform>
                        <fullTransform>(+0.693968 +0.0264789 +0.0548481 -0.717425)[+0.643532 +677.485 -405.468]</fullTransform>
                </transform>
        </graspableBody>
        <robot>
                <filename>///home/alejandro/Graspit/models/robots/Barrett/Barrett.xml</filename>
                <dofValues>+0 +1.95938 +1 +1.86438 +0 +1.97938 +1 +1.8175 +0 +1.85813 +0 +0 </dofValues>
                <transform>
                        <fullTransform>(+1 +0 +0 +0)[+0 +676.516 -432.999]</fullTransform>
                </transform>
        </robot>
        <camera>
                <position>-350.12 -591.72 +375.133</position>
                <orientation>+0.449239 -0.0633683 -0.0858087 +0.887021</orientation>
                <focalDistance>+1564.4</focalDistance>
        </camera>
</world>
```

Figura 7.2: Fichero de mundo .xml

En la figura 7.2 aparece el contenido de un fichero de mundo. Los campos más importantes son:

- **fullTransform**: Este campo aparece en todos los elementos (robots, objetos y obstáculos) y sirve para determinar la posición y orientación. La posición se fija mediante coordenadas cartesianas y la orientación por medio de cuaternios.

Posición

<fullTransform> $(q_0, q_1, q_2, q_3)[X, Y, Z]$ </fullTransform>

Orientación

Figura 7.3: Definición de posición y orientación en GraspIt!

- **dofValues**: El campo *Degrees of Freedom* es característico de los robots. Contiene la configuración articular del robot que se está representando en la escena. La estructura de este parámetro es algo compleja y varía en función de los tipos de articulaciones que dispone el robot. En la figura 7.3 se muestra la estructura para la BarrettHand.

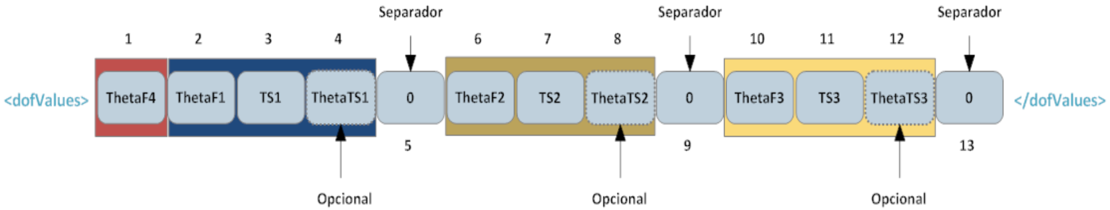


Figura 7.4: Campos de articulaciones de la BarrettHand en GraspIt!

La nomenclatura utilizada para enumerar las articulaciones de la BarrettHand es la misma que la indicada en el apartado 3.2.
Los campos que se muestran en la figura 7.4 se detallan en la tabla 7.1.

Tabla 7.1: Parámetros de articulaciones de la BarrettHand en GraspIt!

ThetaFx	Ángulo de la articulación *x*. Expresado en radianes.
TSx	Indica si se ha activado el mecanismo TorqueSwitch (véase apartado 2.2.2). 0: No se ha activado 1: Sí se ha activado
ThetaTSx	Indica el ángulo de la articulación *x* en el que se ha producido la activación del TorqueSwitch. Expresado en radianes. Este campo solo aparece si el mecanismo TorqueSwitch ha sido activado.

Hay que prestar atención a que la estructura anterior variará de tamaño en función de si se activa o no, el mecanismo TorqueSwitch. Por lo tanto, el campo de los ángulos de los dedos 2 y 3 puede cambiar de posición respecto al inicio (la estructura variará entre diez y trece campos). Otro factor a tener en cuenta es que el valor de ThetaFx indica el ángulo de la articulación 1 (véase figura 2.10) siempre y cuando no se ejecute el mecanismo TorqueSwitch. En el caso de que se ejecute, el valor contendrá la suma de los ángulos de las articulaciones 1 y 2 (véase figura 2.10). A consecuencia de esto, para conocer los ángulos de las articulaciones de forma independiente **una vez se activa el TorqueSwitch** (TSx=1) se deben aplicar las ecuaciones 7.1 y 7.2.

$$\theta_{Articulación\ 1} = ThetaTSx \quad (7.1)$$

$$\theta_{Articulación\ 2} = ThetaFx - ThetaTSx \quad (7.2)$$

7.3 Planificadores

GraspIt! cuenta con tres modelos de planificadores encargados de obtener las configuraciones articulares adecuadas para coger un objeto. Los planificadores son los siguientes:

- **Primitive-base Planner**: Su funcionamiento se basa en aproximar el objeto que se quiere agarrar a figuras primitivas (cilindros, conos, cubos o esferas) para establecer la estrategia en el agarre mediante repetidas simulaciones.
- **Eigengrasp Planner:** Basado en la búsqueda de configuraciones utilizando espacios Eigengrasp [17] mediante los que se utilizan posturas predefinidas (ver figura 7.5).
- **Database Planner:** Utiliza una base de datos de objetos y las correspondientes configuraciones para el agarre, en la que busca similitudes para aproximar la configuración.

El planificador que se mostrará en este capítulo es el *Primitive-base Planner*.
Este tipo de planificador realizará una serie de simulaciones calculando las diferentes configuraciones posibles. Finalmente a cada configuración válida se le asignará un factor de calidad (***Quality Measure***) que dependerá de varios parámetros como por ejemplo la superficie en contacto con el objeto, las fuerzas en los puntos de contacto, etc. Este parámetro

tendrá **valores comprendidos entre 0 y 1**, siendo los valores más altos los que proporcionan un mejor agarre.

Model	DOFs	Eigen-grasp	Description	Min	Max
Barrett	4	1	Spread angle opening		
		2	Finger flexion		
DLR	12	1	Prox. joints flexion Finger abduction Thumb flexion		
		2	Dist. joints flexion Prox. joints extension Thumb flexion		

Figura 7.5: Eigengrasp en diferentes manos robóticas. Extracto de [17]

7.4 Ejemplo práctico

Para finalizar el capítulo se presenta un ejemplo de planificación de un agarre utilizando GraspIt!. Se utilizará la mano robótica BarrettHand y el objeto que se quiere agarrar será un cilindro.
A continuación se definen los pasos que se deben seguir.

1. **Importar los elementos que van a ser utilizados**
   - *File → Import Robot →* Seleccionar fichero .xml (Graspit/models/robots/Barrett/Barrett.xml)
   - Se desplaza la BarrettHand ligeramente para que el siguiente objeto a cargar no se superponga.
   - *File → Import Object →* Seleccionar fichero .xml (Graspit/models/objects/cylinder.xml)

2. **Guardar el fichero de mundo**
   File → Save as

3. **Iniciar el planificador**
   Grasp → Planner
   Aparecerá una ventana como la mostrada en la figura 7.6.

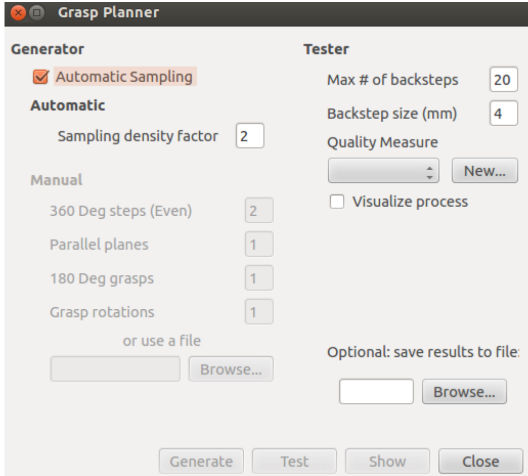


Figura 7.6: Menú de Grasp Planner

Para conocer la utilidad de las diferentes opciones se debe consultar el manual de usuario de GraspIt! [16].

4. Crear una nueva variable de factor de calidad

Al presionar en la opción *New...* aparecerá una ventana como la mostrada en la figura 7.7. En esta ventana se puede elegir una serie de parámetros que determinarán cómo se medirá el factor de calidad. Se pulsará *Add/Edit* y a continuación *OK*.

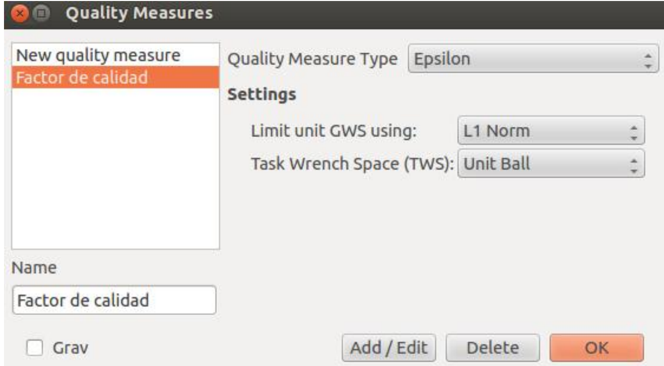


Figura 7.7: Factor de calidad

5. Ejecutar el planificador

- Marcar la opción *Visualize process* para ver cómo se van haciendo las simulaciones.
- Pulsar *Generate.*
- Pulsar *Test*.

Comenzará la ejecución del planificador mostrando múltiples intentos de captura.

6. Mostrar las configuraciones válidas

Una vez se haya finalizado las simulaciones se habilitará el botón *Show.* Cada vez que se pulse se mostrará una de las configuraciones válidas calculadas junto a su factor de calidad (figura 7.8).

Para comprobar cada configuración articular se deberá guardar el fichero de mundo y a continuación abrirlo con un editor de textos comprobando los campos pertinentes (véase apartado 7.2).

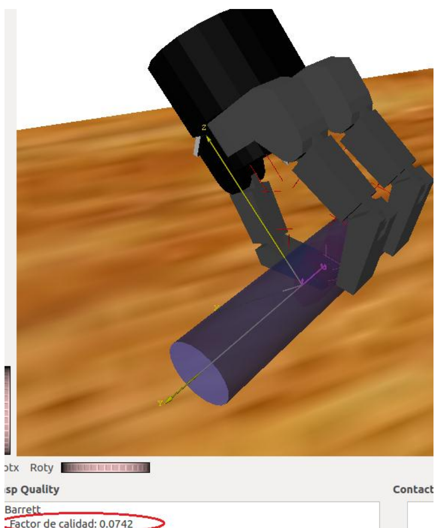


Figura 7.8: Factor de calidad de agarre

En la figura 7.9 se muestran algunos de los resultados obtenidos en la planificación del agarre. En la tabla 7.2 se encuentra un resumen de la configuración articular correspondiente a cada agarre junto a su factor de calidad.

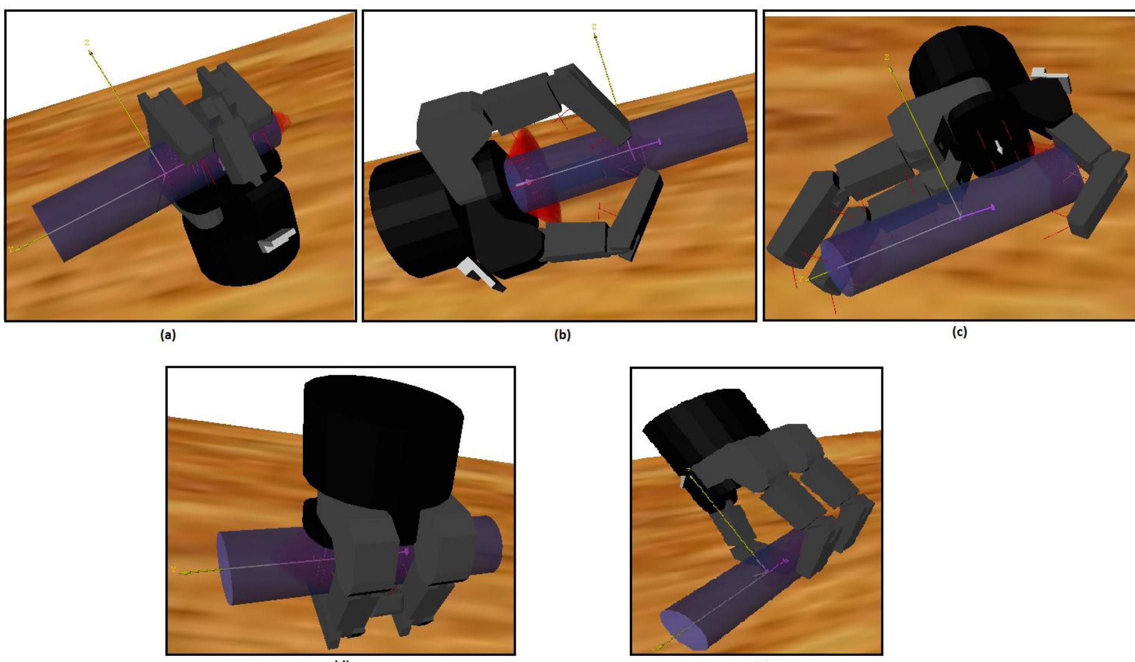


Figura 7.9: Distintas configuraciones de agarre

Tabla 7.2: Configuración articular de las distintos agarres

	a	b	c	d	e
ThetaF1	110°	81°	0°	110°	70°
ThetaF2	110°	81°	0°	110°	70°
ThetaF3	110°	69°	0°	110°	70°
ThetaF4	0°	60°	68°	0°	0°
Factor Calidad	0.27	0.23	0.159	0.27	0.0742

Se puede observar la importancia del factor de calidad en el agarre comparando la configuración *'a'* de la figura 7.9 con la *'e'*. En este último caso el agarre es mucho menos robusto que en el caso 'a', por lo que tiene un factor de calidad muy inferior (0.0742 frente a 0.27).

7.5 GraspIt! en ROS

Para utilizar Graspilt! Con ROS se dispone del paquete *graspit_simulator* que ofrece métodos para comunicar esta herramienta con las distintas aplicaciones de ROS. El paquete está construido con la herramienta *rosbuild* utilizada en las distribuciones más antiguas de ROS (como *electric* o *fuerte*) por lo que para poder ser utilizado en versiones actuales es necesario portar el paquete al sistema de construcción de paquetes *catkin.*

CAPÍTULO 8

RESULTADOS

8. RESULTADOS

8.1 Introducción

En este capítulo se expondrán los resultados obtenidos en la aplicación de seguimiento y captura. Estos resultados serán fundamentalmente la tasa de aciertos y fallos que se dan al coger la pelota. Finalmente se expondrán las razones por las que aparecen estos fallos.

8.2 Tasa de aciertos y fallos en la aplicación

La metodología utilizada para medir la tasa de aciertos y fallos en esta aplicación consiste en realizar varias series de **25 lanzamientos** de la pelota desde diferentes posiciones, es decir, con diferentes ángulos iniciales del péndulo respecto la vertical.
Los ángulos designados son: 12°, 16° y 20°.

Tabla 8.1: Tasa de aciertos. Lanzamiento 12°

Lanzamiento con ángulo de 12º	
Aciertos: 68%	Fallos: 32%

Tabla 8.2: Tasa de aciertos. Lanzamiento 16°

Lanzamiento con ángulo de 16º	
Aciertos: 76%	Fallos: 24%

Tabla 8.3: Tasa de aciertos. Lanzamiento 20°

Lanzamiento con ángulo de 20º	
Aciertos: 44%	Fallos: 66%

8.3 Errores

Tras realizar las medidas, se comprueba que la tasa de aciertos no alcanza el 100%. La principal causa de los fallos en la recogida de la pelota son las **incertidumbres en los tiempos de retardo**.

Como se vio en el apartado 4.5 en el sistema existen múltiples retardos. Estos retardos no son siempre constantes si no que varían de forma aleatoria. Estas variaciones pueden producir que los elementos por los que se encuentra formado el sistema no respondan de la misma forma en ocasiones distintas, dando lugar a problemas en la sincronización y haciendo que en algunos casos la ejecución del movimiento de captura se adelante o se retrase respecto el movimiento del péndulo.
El error producido por estas incertidumbres se acentúa debido a la gran velocidad de la pelota. Para tener una aproximación del margen de error que existe en la aplicación, se va a tener en cuenta las incertidumbres que aparecen en los diferentes retardos utilizando la desviación estándar calculada en las tablas 4.2 y 4.3.
En la tabla 8.4 se resumen las desviaciones estándar.

Tabla 8.4: Resumen desviaciones estándar

σ [ms]			
Retardo mecánico	**Retardo comunicación Ethernet**	**Retardo adquisición de imágenes**	**Retardo Sección crítica**
41	16	2.15	2.56
σ_{TOTAL}≈62ms			

Suponiendo que la incertidumbre del sistema es σ_{TOTAL} y que el error en el retardo se manifiesta de forma simétrica respecto al retardo general medio (apartado 4.5) $T_{general}$ que sufre el sistema (figura 8.1) se puede hacer una estimación del error cometido. Habrá casos en los que el retardo sea superior a $T_{general}$, en los cuales, la acción de coger la pelota se retrasará, por lo que la pelota pasará frente al robot sin que dé tiempo a capturarla. En los casos en los que el retardo sea inferior a $T_{general}$ la acción de coger la pelota se anticipará por lo que la pelota no llegará al punto ideal de captura.

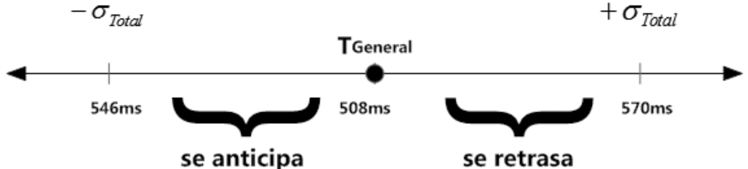


Figura 8.1: Desviación del retardo general

Para conocer cuál es el error de posición de la pelota debido a las incertidumbres en los retardos, se utiliza la figura 8.2 midiendo la distancia que puede recorrer sobre el eje X durante el transcurso de un tiempo σ_{TOTAL}. El error sobre el eje Y no se analizará ya que es muy inferior respecto al del eje X.
Hay que tener en cuenta que a mayor ángulo inicial desde donde se produce el lanzamiento, mayor velocidad adquiere la pelota, por lo que será mayor el error de posición.

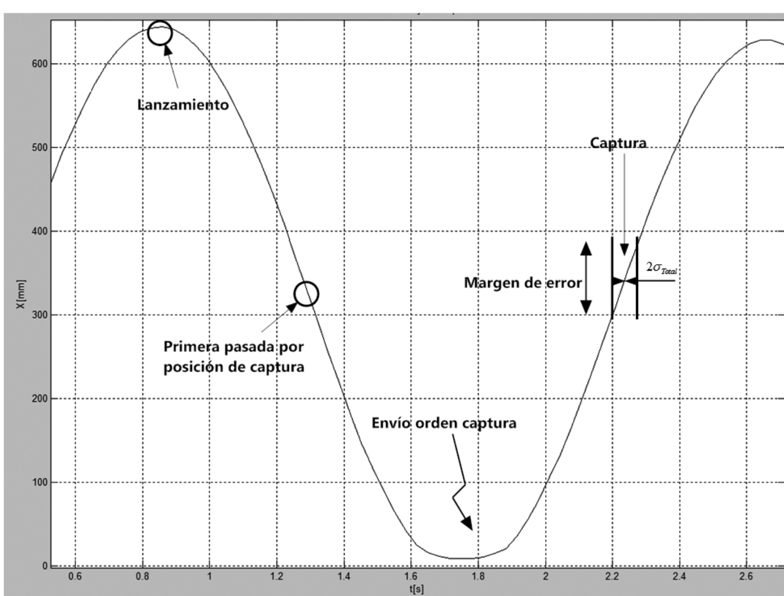


Figura 8.2: Error de posición

Los márgenes de error son:

- Ángulo inicial: 12º

  $\varepsilon_x = \pm 46$ mm

- Ángulo inicial: 16º

  $\varepsilon_x = \pm 59$ mm

- Ángulo inicial: 20º

  $\varepsilon_x = \pm 75$ mm

Hay que tener presente que estos valores tan solo son una aproximación para conocer el orden de magnitud del error, ya que el error real dependerá de más factores además de los analizados.

CAPÍTULO 9

CONCLUSIONES Y TRABAJO FUTURO

9. CONCLUSIONES Y TRABAJO FUTURO

9.1 Conclusiones

En este TFG se han abordado múltiples temas:
El **desarrollo del paquete de control de la BarrettHand** en ROS Hydro permite al usuario de ROS llevar el control de la BarrettHand de forma sencilla y rápida, reduciendo el tiempo de desarrollo de aplicaciones en las que sea necesaria. De forma añadida, se ha conseguido obtener un paquete con una buena compatibilidad entre diferentes equipos incluyendo versiones para 32 y 64 bits.

Por otro lado, se han llevado a cabo diferentes **aplicaciones prácticas** de temática diferente (Captura de una pelota en movimiento, clasificación de objetos) en las que se hace uso del paquete de control mencionado anteriormente. En la aplicación de captura de una pelota en movimiento se ha profundizado en las diferentes limitaciones del sistema utilizado. Con los resultados obtenidos se puede concluir que el sistema no es el más idóneo para manipular objetos que se mueven a altas velocidades, pero es válido en situaciones en los que los objetos a manipular se mueven con velocidades bajas o en caso de objetos estáticos, como se muestra en la aplicación de clasificación de objetos. Una de las limitaciones que ha condicionado el diseño de las aplicaciones es la velocidad máxima de trabajo del robot. A pesar de que permite velocidades de hasta 2000 mm/s, el soporte y anclaje del robot (la mesa) no se encuentra fijada al suelo y no presenta la suficiente robustez ante movimientos con elevadas velocidades y aceleraciones. Por ello no es recomendable utilizar velocidades superiores a 800mm/s debido a que se producen excesivas vibraciones.

Resaltar que las aplicaciones realizadas, tanto de forma individual como colaborativa, sientan una base para el diseño e implementación de sistemas de carácter industrial en aplicaciones relacionadas con la producción.
Finalmente, el estudio de las principales características de la aplicación GraspIt! permite llevar a cabo la planificación del agarre de objetos en un entorno simulado.

9.2 Trabajo futuro

A continuación se exponen algunas de las mejoras que podrían ser implementadas a partir de este TFG.

- Portar el paquete de control de la BarrettHand en un ordenador de pequeño tamaño basado en ARM, como el Raspberry Pi. Este ordenador permite instalar un sistema operativo Linux y cuenta con soporte de ROS. Las principales ventajas de este equipo son sus reducidas dimensiones y bajo consumo, lo que permitiría incorporarlo en el bloque de la fuente de alimentación de la BarrettHand. Con esta configuración y mediante comunicación Ethernet o mediante puerto serie, el Raspberry podría actuar de intermediario entre los diferentes controladores de brazos robot (como el IRC5 de ABB o el KR C4 de KUKA) y la BarrettHand. De esta forma, el control de la BarrettHand se podría realizar además de con un ordenador, desde el propio robot, con su lenguaje de programación.

- Aumentar las funcionalidades del paquete de control de la BarrettHand ampliando la lista de servicios disponibles utilizando los sensores de fuerza disponibles. Permitiría agarrar objetos con distinto grado de deformabilidad de forma adaptativa.

- Portar el paquete de planificación de agarres GraspIt! disponible en ROS (para distribuciones Fuerte y Electric) a las nuevas versiones o bien generar uno nuevo.

CAPÍTULO 10

DIAGRAMAS

10. DIAGRAMAS

En este capítulo se resumirán los diagramas generales que definen los elementos y el funcionamiento de las aplicaciones de captura de una pelota (véase capítulo 4) y seguimiento (véase capítulo 5).

10.1 Diagrama de la arquitectura general del sistema

En la figura 10.1 se resumen aquellos elementos que han sido utilizados en este TFG.

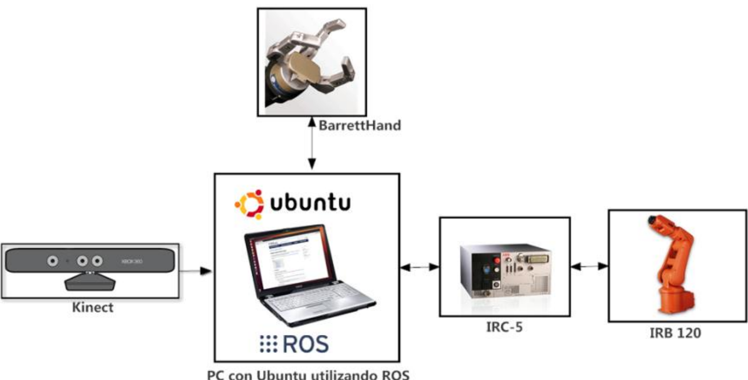


Figura 10.1: Arquitectura general del sistema

10.2 Aplicación de captura de una pelota en movimiento

En la figura 10.2 se muestra el esquema general de los componentes software y hardware más importantes que se han utilizado en esta aplicación.

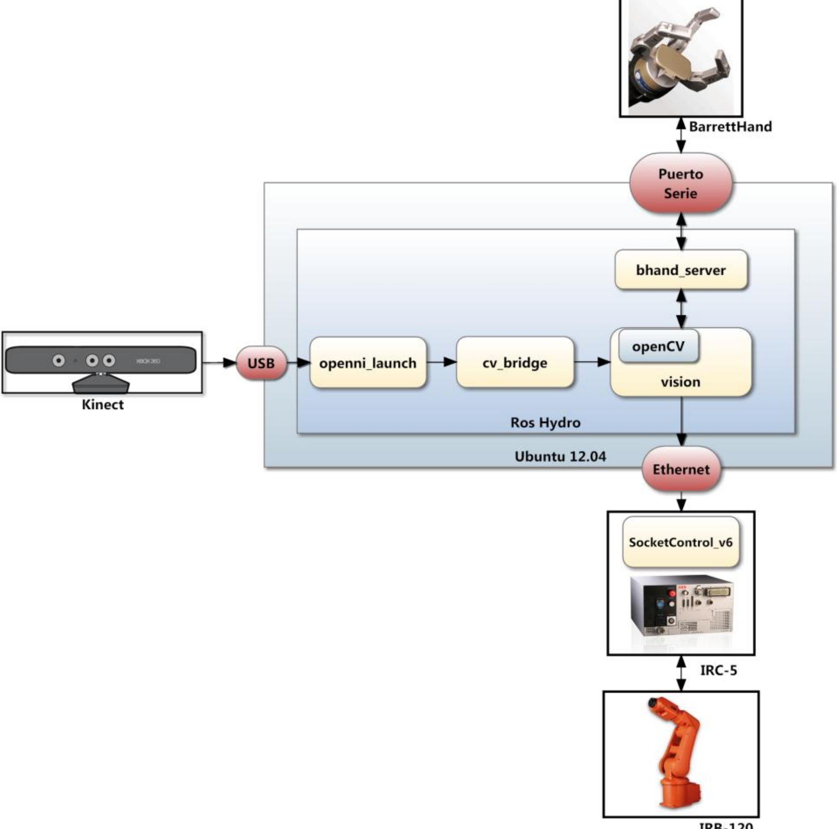


Figura 10.2: Arquitectura Aplicación de captura

Por otro lado, en la figura 10.3 se resume la lógica de funcionamiento del programa encargado de la aplicación.

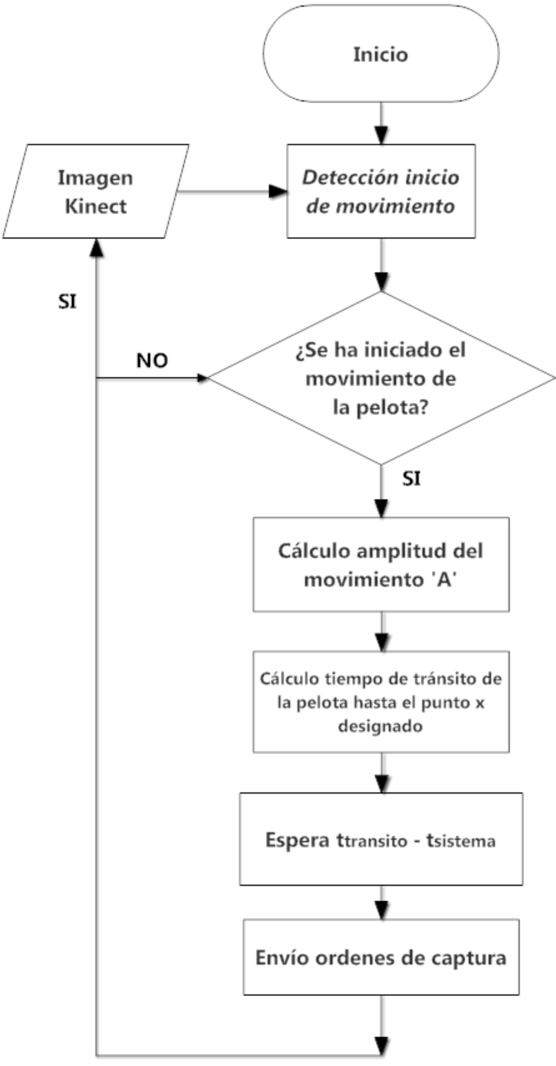


Figura 10.3: Aplicación de captura. Diagrama de flujo

10.3 Aplicación de seguimiento

En la figura 10.4 se muestra el esquema general de los componentes software y hardware más importantes que se han utilizado en esta aplicación.

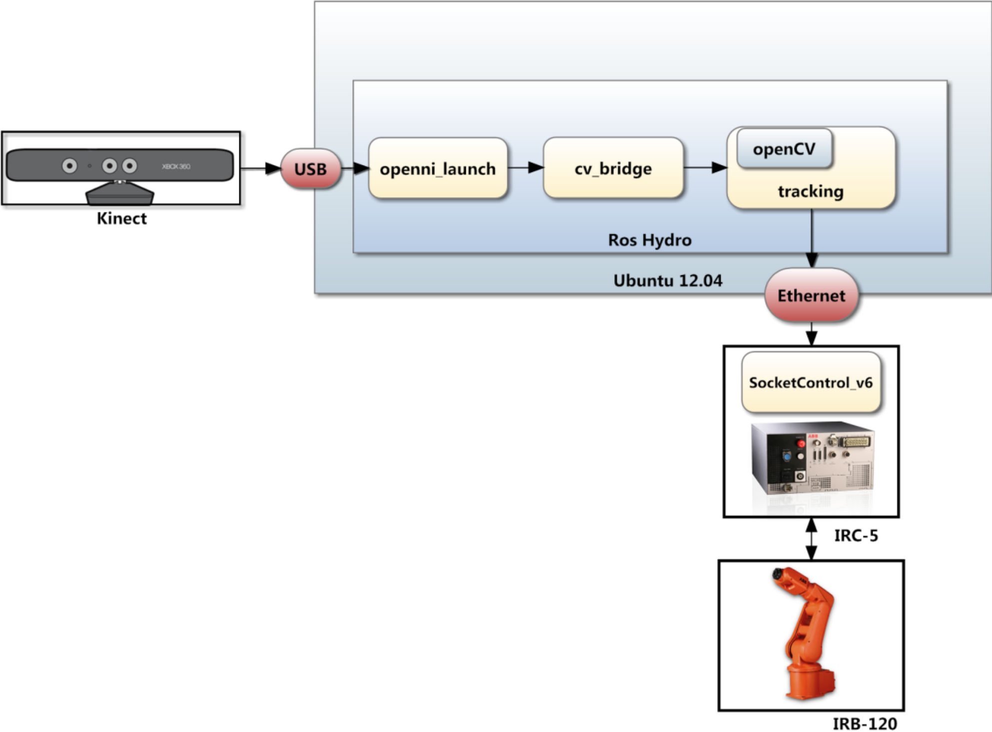


Figura 10.4: Arquitectura Aplicación de seguimiento

En la figura 10.5 se resume la lógica de funcionamiento del programa encargado de la aplicación.

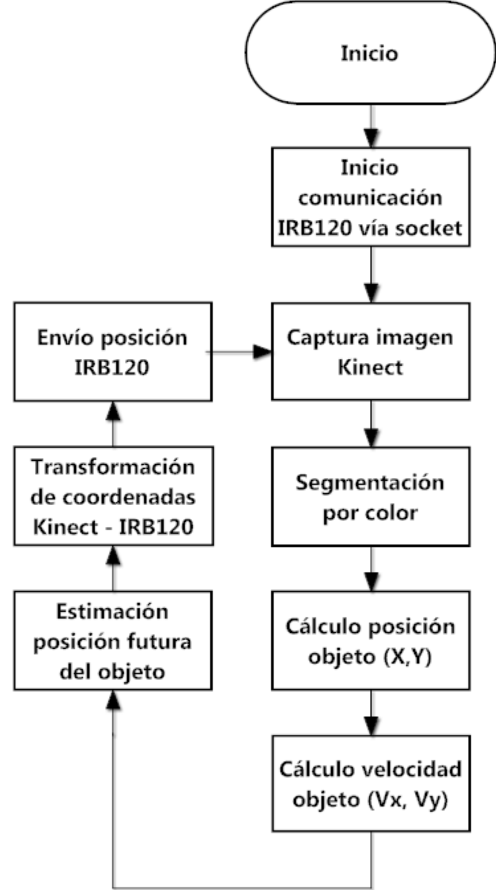


Figura 10.5: Aplicación de seguimiento. Diagrama de flujo

CAPÍTULO 11

PLIEGO DE CONDICIONES

11. PLIEGO DE CONDICIONES

11.1 Requisitos Hardware

- Brazo robot industrial ABB IRB 120 con controlador IRC5
- Mano robótica BarrettHand BH8-262
- Cámara Kinect 1.0
- PC con procesador Intel Core 2 Duo a 2GHz (32 bits) con 2GB de RAM y tarjeta de red
- Conversor SERIE-USB

11.2 Requisitos Software

- Sistema operativo Ubuntu 12.04 LTS 32 bits
- Framework ROS Hydro Medusa
- Aplicación ROS openni_launch
- Aplicación RAPID SocketControl
- Aplicación GraspIt!
- Sistema operativo Microsoft Windows 7
- RobotStudio
- Matlab/Simulink R2010a

CAPÍTULO 12

PRESUPUESTO

12. PRESUPUESTO

12.1 Presupuesto de ejecución material

El presupuesto de ejecución material resume el coste de los elementos hardware y software utilizado así como el coste de la mano de obra.

- **Coste equipos**

Tabla 12.1: Costes equipos (21% IVA incluido)

HARDWARE	PRECIO [€/ud]	UNIDADES	TOTAL [€]
ABB IRB 120 con IRC5	12000	1	12000
BarrettHand BH8-262	40000	1	40000
Kinect	150	1	150
Ordenador portátil Toshiba P200	1000	1	1000
Conversor SERIE-USB	12	1	12
TOTAL HARDWARE			**53162**
SOFTWARE	PRECIO [€/ud]	UNIDADES	TOTAL [€]
Ubuntu 12.04	0	1	0
ROS	0	1	0
SocketControl	0	1	0
Microsoft Windows 7	50	1	50
RobotStudio	0	1	0
Matlab R2010a	500	1	500
Microsoft Office 2010	80	1	80
SmartDraw	200	1	200
TOTAL SOFTWARE			**830**
TOTAL MATERIAL			**53992**

- **Coste mano de obra**

Tabla 12.2: Costes mano de obra

CONCEPTO	PRECIO[€/mes]	TIEMPO [mes]	TOTAL [€]
Documentación	1200	1	1200
Implementación	1500	4	6000
Redacción memoria	1000	1	1000
TOTAL MANO DE OBRA			**8200**

12.2 Importe total

Tabla 12.3: Coste total

CONCEPTO	COSTE [€]
Material	53992
Mano de obra	8200
Impresión memoria	150
TOTAL	**62342**

CAPÍTULO 13

MANUAL DE USUARIO

13. MANUAL DE USUARIO

13.1 Introducción

En las siguientes páginas se resumirán los pasos que se deben seguir para instalar y ejecutar las diferentes aplicaciones utilizadas en este proyecto.

13.2 Configuración de la BarrettHand

Antes de poder utilizar la BarrettHand es necesario cargar el Firmware. Las instrucciones del proceso se encuentran en el manual de usuario de la BarrettHand [18].

13.3 Instalación de las aplicaciones

13.3.1 Instalación ROS Hydro

Para realizar la instalación de ROS Hydro sobre Ubuntu 12.04 se debe disponer de conexión a internet y ejecutar los siguientes comandos en un terminal:

```
$ sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu precise main" > /etc/apt/sources.list.d/ros-latest.list'
$ wget http://packages.ros.org/ros.key -O - | sudo apt-key add –
$ sudo apt-get update
$ sudo apt-get install ros-hydro-desktop-full
$ sudo rosdep init
$ rosdep update
$ echo "source /opt/ros/hydro/setup.bash" >> ~/.bashrc
$ source ~/.bashrc
$ source /opt/ros/hydro/setup.bash
```

Con estos comandos ROS Hydro habrá sido instalado en el sistema. A continuación se debe configurar el espacio de trabajo que será utilizado para construir los diferentes paquetes de ROS.

```
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/src
$ catkin_init_workspace
$ cd ~/catkin_ws/
$ catkin_make
$ source devel/setup.bash
```

Una vez han sido ejecutados, aparecerá en el directorio *home* una carpeta llamada *catkin_ws*. En el interior de este directorio encontramos tres nuevas carpetas:

- build
- devel
- src

La carpeta src se utilizará para usar los paquetes construidos en este TFG.

13.3.2 Instalación del paquete ROS openni

Los pasos para instalar el software de captura de imágenes de la Kinect openni son:

```
$ sudo apt-get install ros-hydro-openni-camera
$ sudo apt-get install ros-hydro-openni-launch
```

13.3.3 Instalación del paquete ROS bhand

Para instalar el paquete *bhand* se distinguen dos casos:

- Sistema de 32 bit
- Sistema de 64 bit

Todos los archivos necesarios se encuentran en la carpeta InstalacionBhand localizada en el dispositivo de almacenamiento adjunto en este documento.

Figura 13.1: Carpetas instalación paquete bhand

A) Sistema de 32 bit

Para instalar el paquete en un sistema de 32 bit se utilizarán los archivos contenidos en:

/InstalacionBhand/32bit

Figura 13.2: Carpetas instalación bhand versión 32bit

Se deben seguir los siguientes pasos:

1. Instalación librerías de Barrett.

En primer lugar se deberá instalar las librerías de Barrett que son utilizadas en el paquete bhand para comunicarse con la BarrettHand.
Para ello se dispone del instalador ***bhand12_30_10.freeglut3.deb***.
Al ejecutar dicho archivo se abrirá el Centro de Software de Ubuntu y la instalación se realizará de forma automática.

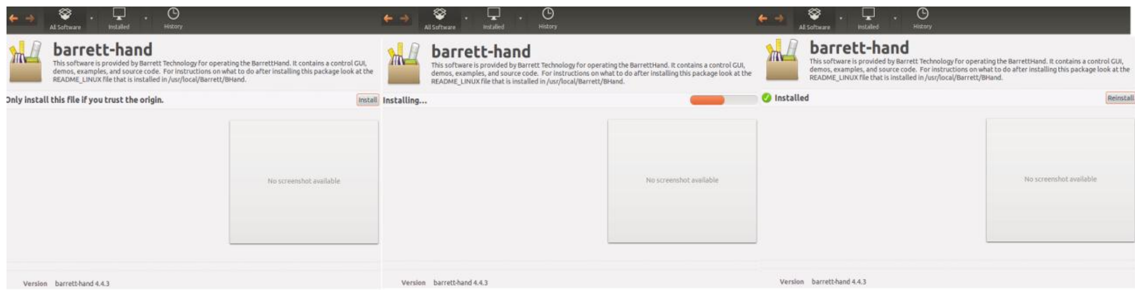


Figura 13.3: Instalación librerías BarrettHand 32bit

2. Instalación paquete bhand

Se deberá copiar la carpeta ***bhand*** en el directorio de trabajo de ROS:

/home/nombre_usuario/catkin_ws/src

3. Construcción del paquete bhand

Para la construcción de paquetes siempre habrá que posicionarse en el directorio del espacio de trabajo de ROS, *catkin* (*/home/nombre_usuario/catkin_ws*). Para construir el paquete *bhand* a partir del código fuente se debe ejecutar en un terminal:

```
$ cd ~/catkin_ws/
$ catkin_make
```

B) Sistema de 64 bit

Para instalar el paquete en un sistema de 64 bit se utilizarán los archivos contenidos en:

/InstalacionBhand/64bit

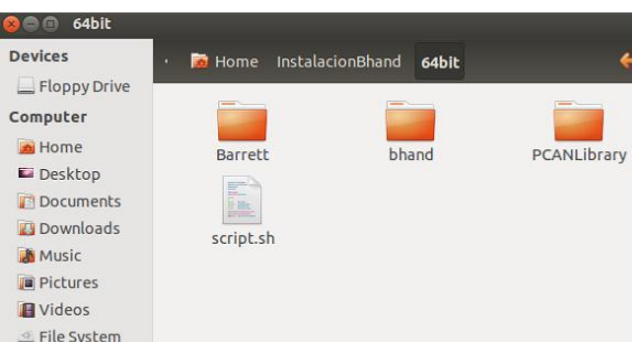


Figura 13.4: Carpetas instalación bhand versión 64bit

En este caso no se dispone de un instalador automático (.deb) por lo que se deberán instalar las diferentes librerías de forma manual.
Para facilitar la instalación se ha generado el archivo ***script.sh*** que permite ejecutar los diferentes comandos de terminal de forma automática.
Los pasos a seguir son:

1. Situar carpeta InstalacionBhand

En primer lugar se copiará la carpeta InstalacionBhand en el directorio:

/home/nombre_usuario

2. Ejecutar el script de instalación

```
$ sudo sh script.sh
```

El script pedirá por la pantalla del terminal el nombre de usuario de Ubuntu.

3. Construcción del paquete bhand

```
$ cd ~/catkin_ws/
$ catkin_make
```

Si se quiere realizar manualmente la instalación de las librerías en lugar de utilizar el script se deben seguir las siguientes instrucciones:

1. Situar carpeta InstalacionBhand

En primer lugar se copiará la carpeta InstalacionBhand en el directorio:

/home/nombre_usuario

2. Instalación librería Peak-System

Se trata de una librería empleada en Linux para comunicación por bus CAN y es utilizada por la librería de Barrett.

```
$ cd /home/nombre_usuario/InstalacionBhand/64bit/PCANLibrary/peak-linux-driver-7.11
$ make
$ sudo su -c "make install"
```

3. Instalación librerías Barrett

En este paso se instalarán las librerías de Barrett que se deben copiar en el directorio /usr/local. Para poder copiar esta carpeta en un directorio del sistema se debe realizar como super-usuario.

```
$ sudo cp -R /home/nombre_usuario/InstalacionBhand/64bit/Barrett /usr/local
```

Como esta carpeta ha sido copiada en modo super-usuario, los permisos para modificar dicha carpeta están restringidos a super-ususario. Para permitir que el resto de usuarios puedan modificar los archivos contenidos en la carpeta:

```
$ sudo chmod -R 777 /usr/local/Barrett
```

4. Instalación paquete bhand

Se deberá copiar la carpeta *bhand* en el directorio de trabajo de ROS:

/home/nombre_usuario/catkin_ws/src

Se puede hacer mediante terminal con las siguientes instrucciones:

```
$ cp -R /home/nombre_usuario/InstalacionBhand/64bit/bhand /home/nombre_usuario/catkin_ws/src
$ sudo chmod -R 777 /home/nombre_usuario/catkin_ws/src/bhand
```

5. Construcción del paquete bhand

```
$ cd ~/catkin_ws/
$ catkin_make
```

13.3.4 Instalación del paquete ROS vision

Se deben seguir los siguientes pasos:

1. Situar carpeta vision

Se copiará la carpeta *vision* en el directorio de trabajo de ROS:

/home/nombre_usuario/catkin_ws/src

2. Construcción del paquete vision

```
$ cd ~/catkin_ws/
$ catkin_make
```

13.3.5 Instalación del paquete ROS tracking

Se deben seguir los siguientes pasos:

1. Situar carpeta tracking

Se copiará la carpeta *tracking* en el directorio de trabajo de ROS

: /home/nombre_usuario/catkin_ws/src

2. Construcción del paquete tracking

```
$ cd ~/catkin_ws/
$ catkin_make
```

13.3.6 Instalación de GraspIt!

Para instalar GraspIt! se parte del fichero *"graspit-2.2.tar.gz"* disponible en el sitio web de GraspIt! [16]. Una vez se han descomprimido los archivos que contiene en una carpeta, se debe ejecutar en un terminal:

1. Descargar e instalar dependencias.

```
$ sudo apt-get install libqt4-dev
$ sudo apt-get install libqt4-opengl-dev
$ sudo apt-get install libqt4-sql-psql
$ sudo apt-get install libcoin60-dev
$ sudo apt-get install libsoqt4-dev
$ sudo apt-get install libblas-dev
$ sudo apt-get install liblapack-dev
$ sudo apt-get install libqhull-dev
```

2. Construir la aplicación

Ejecutar en la carpeta raiz de GraspIt!:

```
$ qmake graspit.pro
$ make
```

13.4 Uso de las aplicaciones

13.4.1 Uso del paquete bhand

Una vez el paquete bhand ha sido instalado, para ejecutar el nodo bhand_server se deben realizar los siguientes pasos.

1. **Iniciar nodo Master**

Ejecutar en un terminal:

```
$ roscore
```

2. **Inicializar el espacio de trabajo**

Ejecutar en un nuevo terminal:

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
```

3. **Dar permisos de acceso al puerto USB**

```
$ sudo chmod a+rw /dev/ttyUSB0
```

4. **Iniciar nodo bhand_server**

```
$ rosrun bhand bhand_server
```

Una vez realizados estos pasos, el nodo *bhand_server* contactará con la BarrettHand y se mantendrá ejecutándose en segundo plano.
A continuación se explica brevemente como hacer uso de los servicios que ofrece este nodo. Existen dos formas de utilizar los servicios del nodo *bhand_client*: desde una aplicación de ROS o bien desde el terminal.

A) Control mediante aplicación ROS

1. **Copiar archivos de servicio .srv**

Se debe copiar la carpeta de servicios *srv* localizada en el paquete *bhand* en el directorio raíz del paquete que se esté construyendo, quedando de la siguiente forma:

/home/nombre_usuario/catkin_ws/src/nombre_paquete/srv

2. **Declarar ficheros de cabecera de las variables compartidas de los servicios**

```
#include <nombre_paquete/Open_Close_Hand.h>
#include <nombre_paquete/Set_Fingers_Angles.h>
#include <nombre_paquete/Set_Fingers_Speed.h>
```

3. **Declarar las variables compartidas**

Después de haber inicializado el manejador del nodo:

```
ros::NodeHandle nh;
```

Se debe generar la referencia a los servicios e inicializar las variables compartidas:

```
ros::ServiceClient bhand_client_open =  nh.serviceClient<nombre_paquete::Open_Close_Hand>("/bhand_server/open");

ros::ServiceClient bhand_client_Angles =
nh.serviceClient<nombre_paquete::Set_Fingers_Angles>("/bhand_server/set_fingers");

ros::ServiceClient bhand_client_speed =
nh.serviceClient<nombre_paquete::Set_Fingers_Speed>("/bhand_server/set_fingers_speed");

nombre_paquete::Open_Close_Hand srv;
nombre_paquete::Set_Fingers_Angles srv2;
nombre_paquete::Set_Fingers_Speed srv3;
```

4. Llamada al servicio

Para llamar a un servicio se debe rellenar la variable compartida asociada a dicho servicio:

```
srv2.request.ThetaF1=45;
srv2.request.ThetaF2=45;
srv2.request.ThetaF3=45;
srv2.request.ThetaF4=0;
```

En este ejemplo, se hace uso del servicio *set_fingers* estableciendo los ángulos de los tres dedos a 45°. Una vez se han rellenado los campos de la variable compartida se hace la llamada al servicio:

```
bhand_client_Angles.call(srv2);
```

Una vez se ha ejecutado el servicio, para poder consultar la respuesta del nodo servidor:

```
float thetaF1;
thetaF1=float(srv2.response.ThetaF1);
```

En este caso se almacena en la variable thetaF1 el ángulo final del primer dedo.

Para más información se recomienda consultar el código fuente del nodo *bhand_client* del paquete *bhand*.

B) Control mediante terminal

En algunos casos puede ser más interesante realizar la comunicación con el nodo *bhand_server* por medio del terminal. La principal ventaja de este método es que no será necesario construir un paquete de ROS y realizar la programación del código fuente.
Las instrucciones que se muestran a continuación pueden ser utilizadas directamente sobre el terminal, o bien, pueden ser incluidas en un *script* ejecutable de Linux (.sh) con el fin de automatizar el proceso. Se debe utilizar la función *rosservice* de ROS.

1. Abrir dedos

```
$ rosservice call /bhand_server/open '!!str 123'
```

El valor de la cadena 123 indica que se deben abrir del dedo uno al tres. Se podrá configurar con los dedos que se quieran accionar.

2. Cerrar dedos

```
$ rosservice call /bhand_server/close '!!str 123'
```

3. Configurar articulaciones

```
rosservice call /bhand_server/set_fingers "ThetaF1: 45.0
ThetaF2: 45.0
ThetaF3: 45.0
ThetaF4: 0.0"
```

Establece los ángulos de los tres dedos a 45°.

4. Configurar velocidad del movimiento de las articulaciones

```
rosservice call /bhand_server/set_fingers_speed "SpeedF1: 100
SpeedF2: 100
SpeedF3: 100
SpeedF4: 100"
```

13.4.2 Uso del socket en IRC5

Para el uso de la aplicación RAPID que maneja el socket de comunicación se debe estar familiarizado con el uso del IRC 5 y el FlexPendant [19].

¡Advertencia! Las aplicaciones diseñadas utilizan velocidades de movimiento del IRB 120 elevadas, pudiendo llegar a ser muy peligroso. Se recomienda consultar el manual de seguridad [20] antes de ejecutar ninguna aplicación. También es muy recomendable utilizar la primera vez la configuración *manual* del IRB 120 [19] que mantiene la velocidad limitada para asegurar que el funcionamiento es correcto.

Los pasos que se deben seguir para cargar y ejecutar el programa son los siguientes:

1. Cargar programa Socket_Control_v6

Editor de programas → Tareas y programas → Archivo → Cargar programa → Localizar el archivo Socket_Control_v6 en el explorador de archivos.

2. Cambiar el modo de funcionamiento del robot a Automático

Consultar manual de operador del IRC 5 [19].

3. Ejecutar el programa RAPID

13.4.3 Ejecución Aplicación de captura de una pelota en movimiento

1. Iniciar nodo Master

```
$ roscore
```

2. Iniciar openni

```
$ roslaunch openni_launch openni.launch
```

3. Iniciar nodo bhand_server

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ sudo chmod a+rw /dev/ttyUSB0
$ rosrun bhand bhand_server
```

4. Iniciar SocketControl_v6 en IRC5

Iniciar la ejecución de programa RAPID SocketControl_v6. Consultar [19].

5. Iniciar nodo vision

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ rosrun vision vision
```

13.4.4 Ejecución Aplicación de seguimiento

1. Iniciar nodo Master

```
$ roscore
```

2. Iniciar openni

```
$ roslaunch openni_launch openni.launch
```

3. Iniciar SocketControl_v6 en IRC5

Iniciar la ejecución de programa RAPID SocketControl_v6. Consultar [19].

4. Iniciar nodo tracking

```
$ cd ~/catkin_ws/
$ source devel/setup.bash
$ rosrun tracking tracking
```

13.4.5 Ejecución Aplicación de clasificación de objetos

1. Iniciar openni

```
$ roslaunch openni_launch openni.launch depth_registration:=true
```

2. Iniciar planificador de movimiento y RViz

```
$ roslaunch hand_moveit_config move_group_fast.launch
```

3. Iniciar nodo table

Nodo encargado de representar la mesa.

```
$ rosrun table table
```

4. Iniciar nodo object_attach

Nodo desarrollado en el TFG de Javier Moriano Martín [15] encargado representar los objetos.

```
$ rosrun object_attach object_attach
```

5. Iniciar nodo threeDOR

Nodo desarrollado en el TFG de Alberto Lázaro Enguita [14] encargado del reconocimiento del objeto y obtención de su posición, dimensiones y orientación.

```
$ rosrun threeDOR threeDOR
```

6. Iniciar nodo bhand_server

Para iniciar el nodo de control de la BarrettHand se debe consultar el apartado 13.4.1.

7. Iniciar nodo attach

Nodo desarrollado en el TFG de Javier Moriano Martín [15] encargado del control del IRB 120 y de la comunicación con el nodo *bhand_server*.

```
$ rosrun attach attach
```

BIBLIOGRAFÍA

[1] *“Sitio web SCHUNK”,* http://www.schunk.com/ [Último acceso, 08/Septiembre/2014]

[2] *“Sitio web Robotiq”,* http://robotiq.com/en/ [Último acceso, 08/Septiembre/2014]

[3] *“Sitio web Shadow Robot Company”*, http://www.shadowrobot.com/ [Último acceso, 08/Septiembre/2014]

[4] *“Sitio web Elumotio*n”, http://www.elumotion.com/ [Último acceso, 08/Septiembre/2014]

[5] *“Sitio web ROS”,* http://www.ros.org/ [Último acceso, 08/Septiembre/2014]

[6] *“Sitio web Barrett Technology”*, http://www.barrett.com/ [Último acceso, 08/Septiembre/2014]

[7] *“BH8 C-Function Library Manual”,* 4.3x ed, Barrett Technology Inc*,* 2002.

[8] “*BH8 Series User Manual Firmware”,* 4.3x ed, Barrett Technology Inc, 2005.

[9] *“Sitio web ABB robotics”,* http://new.abb.com/products/robotics [Último acceso, 08/Septiembre/2014]

[10] *“Sitio web Kinect”*, http://msdn.microsoft.com/en-us/library/hh438998.aspx [Último acceso, 08/Septiembre/2014]

[11] Raymond A. Serway and John W. Jewett, *“Physics for Scientists and Engineers with modern physics”,* Brooks/Cole, 7th ed.

[12] *“Manual de referencia técnica. Instrucciones, funciones y tipos de datos de RAPID”,* Rev.J*,* ABB Robotics.

[13] Marek Jerzy Frydrysiak, “*Socket base communication in RobotStudio for controlling ABB-IRB120 robot. Design and development of a palletizing station”,* TFG, UAH, 2014.

[14] Alberto Lázaro Enguita, “*3D Object recognition and pose estimation using VFH descriptors*”, TFG, UAH, 2014.

[15] Javier Moriano Martín, “*Trajectory planning for the IRB120 robotic arm using ROS*”, TFG, UAH, 2014.

[16] *“Sitio web GraspIt!”,* http://www.cs.columbia.edu/~cmatei/graspit/ [Último acceso, 08/Septiembre/2014]

[17] Matei Th. Ciocarlie, “*Low-Dimensional Robotic Grasping: Eigengrasp Subspaces and Optimized Underactuation”,* Columbia University, 2010.

[18] “*BHControl Interface Manual”,* Barrett Technology Inc, 2002.

[19] “*ABB Operating manual, IRC5 with FlexPendant”,* ABB Robotics.

[20] “*Operating manual, Emergency safety information”*, H ed, ABB Robotics.

[21] Jason M. O’kane, *“A Gentle Introduction to ROS”*, University of South Carolina, 2014.

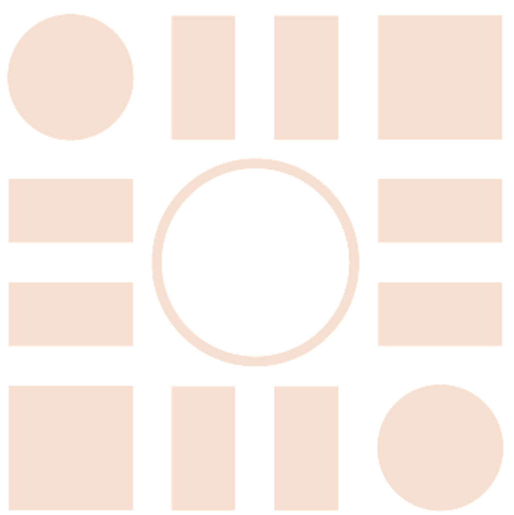
[22] Gary Bradski and Adrian Kaehler, “*Learning OpenCV”,* O’Reilly, 2008.

[23] Hugo A. Tosone y Federico Cavalieri, “*Momentos de 1º y 2º orden”,* Universidad Tecnológica Nacional, 2008.

[24] Andrew T. Miller and Peter K. Allenn, “*Examples of 3D Grasp Quality Computations”*, Columbia University, 1999.

[25] Andrew T. Miller and Peter K. Allen, "*GraspIt! A Versatile Simulator for Robotic Grasping",* IEEE Robotics & Automation Magazine, Dec. 2004.

Universidad de Alcalá
Escuela Politécnica Superior



ESCUELA POLITECNICA
SUPERIOR