Informe Final
Estrategias cooperativas para un equipo de fútbol de robots humanoides

Santiago Duarte, Paul Green
Tutor: Facundo Benavides

Facultad de Ingeniería
Universidad de la República
Uruguay

2016

Resumen

El presente documento aborda las diferentes aristas del trabajo desarrollado en el proyecto *FutRob*, cuyo objetivo principal fue la implementación de comportamiento colaborativo en sistemas multi-robot en la disciplina de fútbol de robots.

En el contexto mencionado, se entiende como comportamiento colaborativo el abanico de acciones que se vean potenciadas por la condición de funcionar bajo un esquema de más de un robot y obteniendo un mejor resultado. En particular, se trabajó sobre los casos de estudio de *"pase y recepción"* entre 2 robots y *"posicionamiento del robot arquero"* en relación a su ubicación en el centro del arco.

Como trabajo previo para alcanzar el objetivo último, se trabajó sobre el desarrollo de la plataforma del robot, adaptando una placa *Raspberry Pi* al kit robótico *Bioloid Premium Kit*, desarrollando una serie de requerimientos de integración de hardware, que abarcan desde conectores hasta la construcción de una placa de interfaz de hardware entre los motores y la *Raspberry Pi*.

La principal dificultad para el armado de la plataforma de hardware fue la elaboración propia del circuito eléctrico que nos permitió conectar los motores AX-12 a la placa *Raspberry Pi*, ya que nos implicó profundizar en conocimientos electrónicos que conocíamos apenas superficialmente. El desafío fue construir un componente a medida para convertir un bus *half-duplex* (semi dúplex) en un bus *full-duplex* (dúplex completo) y viceversa. En segundo lugar, debimos afrontar la dificultad para conseguir componentes electrónicos que suelen ser de uso específico en robótica, desde las placas *Raspberry Pi*, incluyendo tarjeta de red inalámbrica USB compatible, carcasa plástica, conectores micro-USB en forma de codo para alimentación, y conectores MOLEX macho y hembra para actuadores Dynamixel.

Paralelamente trabajamos en diseñar una arquitectura de software que a la vez optimizara y aprovechara al máximo los recursos de hardware disponibles. El diseño está fuertemente basado en los 4 hilos físicos del procesador de la placa, y supone la implementación en hilos de ejecución independientes para: la lógica de razonamiento y valoración de reglas de comportamiento, la comunicación mediante protocolo WiFi, el manejo de acciones a través de los motores, y la captura y procesamiento de imágenes a través de la cámara web.

Finalmente el trabajo se completó con la implementación de la solución de software propuesta. Ésta fue concebida con la capacidad de ser ampliamente configurable, y con el objetivo de lograr un producto totalmente extensible, facilitando futuros proyectos de ampliación de la solución.

Índice

1 Introducción **1**
1.1 Motivación 1
1.2 Objetivos 2
1.3 Estructura del documento 4

2 Marco de trabajo **6**
2.1 Competencia robótica: Robocup 6
2.1.1 Categorías de RoboCup Soccer 8
2.1.2 Restricciones para RoboCup Humanoid KidSize 9
2.2 Marco teórico 14
2.3 Software reutilizado 15
2.3.1 Visión de robots I y II 15
2.3.2 Salimoo: Fútbol de robots para la liga humanoide de RoboCup 15
2.3.3 Futbot: Fútbol de robots humanoides 16
2.3.4 Forrest: Desarrollo de un equipo de fútbol de RoboCup 16

3 Plataforma de trabajo **18**
3.1 Kit robótico Bioloid Premium 18
3.2 Placa controladora Raspberry Pi 19
3.3 Cámara web Logitech C920 20
3.4 Sistema operativo base: Raspbian 21
3.5 Placa de integración zAX12pi 22

4 Descripción de la solución de hardware **24**
4.1 Placa Pi2AX12: interfaz entre motores y placa 24
4.2 Ensamblado del sistema 28
4.2.1 Regulador de voltaje 32
4.2.2 Conectores 33

5 Descripción de la arquitectura de software **35**
5.1 Módulo de toma de decisiones 38
5.1.1 Manejador de reglas 39
5.1.2 Manejador de acciones 42
5.1.3 Modelo del mundo 47
5.2 Módulo de comunicación 58
5.2.1 Monitorización de la comunicación 62
5.3 Módulo de visión 63
5.4 Módulo de manejo de motores 71

5.4.1 Librerías utilizadas 76
5.5 Módulo de registro de actividad del sistema 78

6 Casos de estudio 82
6.1 Individuales 82
6.1.1 Patear un penal 82
6.1.2 Atajar un penal 89
6.2 Colaborativos 95
6.2.1 Descubrimiento, salutación y anuncio 95
6.2.2 Posicionamiento del robot arquero 98
6.2.3 Pase y recepción 103

7 Conclusiones 109
7.1 Combinación de áreas de conocimiento 109
7.2 Complejidad de ensayos experimentales 109
7.3 Resultados condicionados por el hardware 110
7.4 Mecanismos de diagnóstico del sistema 111

8 Trabajo futuro 112
8.1 Incorporar un giroscopio 112
8.2 Mejoras de precisión en el módulo de visión 112
8.3 Algoritmos para refinamiento de posiciones y movimientos . . 113
8.4 Optimizar la discretización del espacio de movimientos 114
8.5 Evaluación de reglas utilizando lógica difusa 115

Índice de figuras

1 Ejemplo de esquema de robot humanoide . . . 10
2 Placa zAX12pi versión final, con resistencias . . . 23
3 Esquema del buffer tri-estado 74LS241 . . . 25
4 Placa Pi2AX12 versión 0.1, incorporando chip 74LS241 . . . 25
5 Placa Pi2AX12 versión 0.4, diseño y tamaño ajustados . . . 27
6 Placa Pi2AX12 versión 1.0 final . . . 27
7 Vista dorso de la placa Pi2AX12 versión 1.0 final . . . 28
8 Vista superior de la placa Pi2AX12 versión 1.0 final . . . 28
9 Bioloid básico frente . . . 29
10 Bioloid básico reverso . . . 29
11 Conexiones de extremidades . . . 30
12 Acoplada Pi2AX12 . . . 30
13 Acoplada Raspberry Pi . . . 30
14 Conectada . . . 30
15 Conectores Pi2AX12 . . . 31
16 Esquema de conexión y conexión física . . . 31
17 Conexión con Raspberry . . . 32
18 Conexión con batería . . . 32
19 Regulador de voltaje UBEC 8S-5A . . . 33
20 Cable USB en L conectado a Raspberry Pi . . . 34
21 Diagrama de módulos del sistema. . . . 35
22 Diagrama detallado de componentes . . . 37
23 Ejemplo de archivo de configuración “ini.config”. . . . 38
24 Diagrama del robot y las distancias a la pelota. . . . 50
25 Inicialización *commRecv* . . . 63
26 Inicialización *commSender* . . . 63
27 Ensayos para calibración de visión. . . . 66
28 Ensayo: medidas de distancia calculadas por el módulo de visión. 67
29 Ensayo: desvío de medida calculada respecto a medida real. . . 67
30 Conclusiones: optimización de ángulo de tilt para cálculo de distancia. . . . 68
31 Diagrama de flujo de manejo de motores . . . 72
32 Ejemplo de archivo de log. . . . 80
33 Diagrama de escenario patear 1 . . . 83
34 Diagrama de escenario patear 2 . . . 83
35 Diagrama de escenario patear 3 . . . 84
36 Imagen de escenario patear . . . 84
37 Diagrama atajar 1 . . . 91

38 Diagrama atajar 2 91
39 Diagrama atajar 3 92
40 Imagen atajar 92
41 Flujo de comunicación de descubrimiento entre 2 robots. 97
42 Diagrama de escenario de reposicionamiento de arquero. 100
43 Imagen de escenario de reposicionamiento de arquero. 101
44 Esquema de escenario de pase. 105
45 Imagen de escenario de pase. 105

1. Introducción

El proyecto pretende incorporar técnicas colaborativas en un conjunto de robots humanoides con el propósito de conformar un equipo de fútbol de robots que adopte comportamientos de cooperación entre sus componentes. Mediante estos mecanismos, se busca mejorar el rendimiento del equipo en función de lograr el principal objetivo de la disciplina de fútbol de robots: ganar un partido que enfrenta dos equipos sujetos a una serie de restricciones reglamentarias. Estas restricciones condicionan principalmente las características y dimensiones de los componentes de hardware y su manipulación, lo que motiva alcanzar la ventaja deportiva utilizando como herramienta la solución de software, que debe implementar la ejecución de acciones en función de un algoritmo de toma de decisiones en función de la información cognitiva.

El principal aliado para la implementación de técnicas colaborativas es la función cognitiva de la comunicación; en este caso a través de una red inalámbrica con puntos de acceso directamente en cada robot.

El abordaje de la solución se da en el marco de la integración de un conjunto de componentes partiendo desde un kit de robótica, liberando las dependencias de licencias propietarias y sustituyendo los componentes afectados por alternativas de uso libre. Esto supone un desafío de integración no solamente a nivel de hardware, considerando interfaces de conexión, aspectos eléctricos y mecánicos; así como a nivel de software, considerando la sustitución de librerías que conlleva el cambio de sistema operativo base, e incorporación de nuevas librerías para el manejo de los componentes de hardware agregados.

1.1. Motivación

En el transcurso de los últimos años hemos sido testigos del incremento de la automatización de tareas mecánicas por artefactos de robótica. En primera instancia, en la industria: fábricas y grandes superficies. En estos escenarios se utilizan robots de gran porte; el contexto posibilita grandes inversiones para el desarrollo y construcción de robots que agreguen valor y reduzcan tiempos de retorno a la cadena de producción.

Luego, en una segunda etapa, con la disminución en costos de fabricación, surgieron una variedad de robots de aplicación para tareas domésticas, didácticas, recreativas y hasta de uso personal. De los ejemplos más avanzados recientemente podemos considerar los artefactos de cuidado doméstico como aspiradoras robóticas o incluso cortadoras de césped robóticas.

El derribamiento de barreras tecnológicas en aspectos de hardware posibilita la expansión de la robótica a potencialmente cualquier campo. Simul-

táneamente, esto abre una serie de nuevas corrientes de investigación para el desarrollo conjunto de soluciones de software que maximicen el resultado de los recursos de hardware disponibles en los diferentes segmentos. Así, se avanza constantemente sobre el análisis, diseño e implementación de mecanismos de inteligencia artificial que doten de sentido y tornen de utilidad todo tipo de sistemas de robots.

Este proyecto se concentró esencialmente en retomar parte de las actividades de implementación de inteligencia artificial para la toma de decisiones, y potenciarla complementando con el intercambio de información en un sistema de robots que trabajan de forma colaborativa. Por ende, se buscó avanzar hacia una solución multi-robot con coordinación entre sus miembros. Para esto, se trabajó sobre los mecanismos de comunicación, tanto a nivel técnico para el intercambio de datos, como a alto nivel, para establecer una sintaxis clara y estructurada y una semántica que permite al menos cubrir las necesidades de comunicación identificadas.

El trabajo se presentó como un desafío interesante para potenciar las capacidades de inteligencia artificial de un sistema con interacción directa sobre el espacio físico que lo rodea. A su vez, se trató de un trabajo polifacético que implicó un grado de investigación y desarrollo en diversas capas de una solución tecnológica como: el proceso de ingeniería de software, la integración de componentes de software y una estricta sincronización entre ellos, el análisis y la incorporación de nuevas tecnologías y protocolos, una constante conciencia de las restricciones de procesamiento y memoria dinámica que impone la plataforma y que obligaron la búsqueda de soluciones óptimas a nivel de implementación, e incluso el desarrollo de componentes de hardware específicos que hicieran posible la sustitución de la placa de procesamiento original de un kit de robótica.

1.2. Objetivos

El proyecto se enmarcó en proyectos anteriores del Grupo MINA de la Facultad de Ingeniería en esta misma rama de estudio, que indujeron un nuevo conjunto de requerimientos a investigar y desarrollar. En la sección 2.3 del documento se aborda en mayor detalle cada uno de los proyectos predecesores.

Como bien sintetiza el nombre del proyecto, *"Estrategias cooperativas para un equipo de fútbol de robots humanoides"*, el principal objetivo se centró en el análisis e implementación de técnicas de cooperación en un sistema multi-robot. El abordaje abarcó dos aristas: el uso de herramientas de comunicación entre sus componentes y la implementación de una estructura de toma de decisiones en forma coordinada entre éstos. Complementariamente

se presentaron una serie de condiciones previas que formaron parte de un conjunto básico de requisitos que subyacen la implementación de la comunicación entre robots.

Por un lado, el proyecto tuvo como premisa la utilización de plataformas de uso abierto, sin restricciones de licenciamiento. Esto aplica tanto para los recursos de hardware como para las herramientas de software que se utilizaron para el diseño de la solución. A su vez, el proyecto debió retomar y ampliar el camino transitado por los proyectos predecesores en la rama de robótica, particularmente con la plataforma *Bioloid Premium Kit*, y especialmente en la aplicación de técnicas para la competencia en la RoboCup en su categoría *KidSize football*.

Esto da lugar al primer objetivo de este proyecto: completar la integración de los componentes de plataformas abiertas al sistema base *Bioloid Premium Kit*. Para esto, se debieron analizar las posibilidades de integración de la solución previa con las librerías disponibles en la placa Raspberry Pi, y luego completar el diseño del sistema en forma íntegra, incluyendo el cambio de cableado, conectores, disposición de componentes, y el consecuente desplazo del centro de masa original.

El segundo objetivo del proyecto fue la implementación de una lógica de inteligencia artificial basada especialmente en una combinación del aporte de los resultados de proyectos previos. Si bien el desarrollo de la solución se concibió sujeto a un conjunto de parámetros que permiten regular el comportamiento del sistema considerando futura escalabilidad, se trabajó tomando como premisa que el sistema consta de tres robots ocupando un espacio físico determinado por características distintivas, y enfrentándose a otro grupo de tres robots. Esto hace que el traslado del trabajo realizado en el proyecto *Forrest* [1], de gran utilidad en virtud de este objetivo, no sea directo, puesto que se realizó sobre una plataforma de emulación de robots y contexto. Fue preciso trabajar sobre un modelo que encapsule a más alto nivel los roles de los jugadores, y se debió circunscribir la estructura de razonamiento de los robots a las estrictas restricciones determinadas por la plataforma de hardware.

Complementariamente, la implementación de la estructura de toma de decisiones debió hacer hincapié en la coordinación de los agentes del sistema. Para esto, si bien pueden tenerse especiales consideraciones en la lógica del razonamiento contemplando una *coordinación estática*, uno de los objetivos centrales del proyecto fue incorporar a la solución capacidades de comunicación entre sus componentes. Considerando que la meta de ganar el partido es única y global a todo el sistema, la coordinación entre sus componentes para lograr una solución óptima se convirtió en esencial. Incorporar capacidades de comunicación implicó realizar un análisis de las diferentes alternativas de

componentes de hardware, protocolos de comunicación y la definición de una mensajería acorde a las necesidades del proyecto.

Finalmente, para cubrir los objetivos mencionados anteriormente, surgió un requisito previo ineludible que conformó otro de los objetivos satélite del proyecto: realizar un estudio del estado del arte para el diseño e implementación de estrategias de cooperación y colaboración en sistemas de robótica multi-agente. Esto permitió comprender en forma precisa desde qué sitio partimos en la línea de desarrollo e investigación, cuáles son las alternativas posibles, y hacia dónde encausar los esfuerzos en virtud de lograr resultados de mejor valor.

1.3. Estructura del documento

El documento está estructurado siguiendo un orden similar al orden cronológico en que fueron abordados los diferentes aspectos de la solución. Desde la etapa de investigación hasta la etapa de pruebas del sistema, pasando por el diseño y construcción de los componentes en cada capa.

El capítulo 2 "Marco de trabajo" aborda la etapa de investigación y marco teórico, describiendo el marco regulador de la competencia en la que se circunscribe el proyecto, RoboCup, y los componentes de software utilizados como insumo para el desarrollo de la solución.

En el capítulo 3 "Plataforma de trabajo" se describen detalladamente los componentes de hardware que se toman como punto de partida para la construcción de la plataforma de trabajo. Se describe el kit de robótica utilizado y componentes anexos como placa controladora sustituta, cámara web, placa de red, cableado y placa de interfaz de hardware.

La descripción de la solución comienza en el capítulo 4 "Descripción de la solución de hardware" donde se detalla el armado de la plataforma completa y el trabajo realizado en relación al hardware, destacándose el desarrollo de la placa de interfaz de hardware Pi2AX12 que permite la conexión de sistemas dúplex-completo con sistemas semi-dúplex.

En el capítulo 5 "Descripción de la arquitectura de software" se aborda una descripción integral de los componentes de lógica desarrollados para la solución, incluyendo una descripción detallada de los principales módulos.

El objetivo del proyecto de implementar la cooperación entre robots a través de la comunicación se ve reflejado en los casos descritos en el capítulo 6 "Casos de estudio". Se describen dos grupos de comportamientos implementados: individuales y colaborativos.

El capítulo 7 "Conclusiones" se plantea un resultado general de los objetivos planteados al comienzo del proyecto; y se sugieren una serie de caminos

a investigar como posibles investigaciones a futuro en el capítulo 8 “Trabajo futuro”.

Adicionalmente, se incluye como anexo al presente documento la documentación técnica del sistema, llamado *Ensamblado y utilización del sistema FutRob* [2], con un abordaje práctico abarcando aspectos de ensamblado y utilización del sistema.

2. Marco de trabajo

La implementación de la solución está circunscrita a la consideración de principalmente tres dimensiones: las restricciones reglamentarias que impone la competencia y disciplina en la que enmarcan las bases de la propuesta de trabajo y en la que se enfocan los objetivos de postulación; los avances en las ramas de investigación que confluyen en el desarrollo de la solución; y especialmente, los proyectos de investigación realizados anteriormente por la propia institución de la Universidad de la República en esta área.

2.1. Competencia robótica: Robocup

En octubre de 1992, un grupo de investigadores japoneses organizó un taller de trabajo sobre los *"Grandes Desafíos de la Inteligencia Artificial"* [3] para debatir sobre esta temática. Como resultado de la actividad, surgió la idea de utilizar el fútbol como promotor para trabajos de investigación y desarrollo en ciencia y tecnología. Se hicieron profundas investigaciones cubriendo aspectos como la viabilidad a nivel tecnológico, determinación del impacto social, y viabilidad económico-financiera. Particularmente, se delinearon un conjunto de reglas, y se desarrollaron una serie de prototipos de plataformas robóticas capaces de simular algunas actividades relacionadas al juego de fútbol. Como resultado de este estudio, se concluyó que el proyecto era tanto viable como deseable. Paso siguiente, un grupo de investigadores japoneses entre los que se incluyen Minoru Asada, Yasuo Kuniyoshi, y Hiroaki Kitano, lanzaron en Japón una primera edición de competición de fútbol de robots, a la que llamaron *"Robot J-League"* (debiendo su nombre a la *J-League*, nombre de la primera división del fútbol profesional nipón). Rápidamente se generaron repercusiones en la comunidad alrededor del mundo, solicitando extender este evento hacia la comunidad internacional como un proyecto conjunto. Atendiendo este pedido, se renombró el proyecto como la *"Iniciativa de la Copa Mundial de Robots"* (*"Robot World Cup Initiative"*), y se adoptó la denominación *"RoboCup"* como apócope.

Paralelamente a estos hechos, un conjunto de investigadores habían estado utilizando el fútbol como marco para sus trabajos de investigación. Tal es el caso de Itsuki Noda en el *ElectroTechnical Laboratory (ETL)*, un centro gubernamental de investigación en Japón, investigando sobre sistemas multi-robot en fútbol y comenzando el desarrollo de un simulador dedicado para partidos de fútbol. Este simulador se convertiría luego en el servidor oficial para la disciplina de fútbol simulado en la RoboCup. Independientemente, en el *Laboratorio del Profesor Minoru Asada* de la Universidad de Osaka (Japón) [4] por un lado, y la Profesora Manuela Veloso y su estudiante Peter Stone en

la *Universidad Carneige Mellon* en Pittsburgh, Pensilvania (Estados Unidos) [5], se encontraban trabajando sobre el desarrollo de robots con capacidades para jugar al fútbol. Según reconoce la propia organización *RoboCup*, "sin la participación de estos pioneros en el campo de estudio, RoboCup nunca hubiera comenzado".

En setiembre de 1993 se hizo oficial el primer anuncio de la iniciativa, incluyendo un boceto del marco normativo que lo regularía. Luego, los asuntos de organización y aspectos técnicos fueron discutidos en diversas conferencias y talleres de trabajo, incluyendo el *"AAAI-94, JSAI Symposium"* y en varios congresos de la comunidad de robótica.

En ese momento, el equipo de Itsuki Noda en el *ElectroTechnical Laboratory (ETL)* anunciaba el lanzamiento de la primer versión del motor simulador de fútbol: *Soccer Server versión 0 (versión LISP)*. Se trató del primer simulador de código abierto para el área de fútbol y con la posibilidad de trabajar con ambientes multi-agente. Inmediatamente lanzaron la versión 1.0 del motor: *Soccer Server (C++ version)*, que se distribuyó vía web. La primer demostración pública de este simulador se llevó a cabo en la IJCAI-95 (*The 1995 International Joint Conference on Artificial Intelligence*). Fue en esta conferencia que se hizo oficialmente el anuncio de organizar la Primer Copa Mundial de Fútbol de Robots en conjunto con la IJCAI-97 a desarrollarse en Nagoya, Japón. Además, se definió organizar la Pre-RoboCup-96 para adelantar la identificación de potenciales problemas asociados con la organización del evento RoboCup a gran escala. Se optó por otorgar 2 años de tiempo de preparación para que los equipos de investigación pudieran establecer sus equipos de desarrollo tanto de robots como de simulación, sin afectar sus agendas.

La Pre-RoboCup-96 se desarrolló durante el IROS-96 (*International Conference on Intelligence Robotics and Systems*) en Osaka, Japón, entre el 4 y 8 de noviembre de 1996, con la participación de 8 equipos compitiendo en la liga de simulación y una demostración de robots reales para la categoría *Middle Size*.

La primer RoboCup oficial, juego y conferencia, se desarrolló en 1997 con gran éxito. Más de 40 equipos participaron (considerando robótica real y simulada), y más de 5.000 espectadores asistieron.

Ya consolidada, la competencia RoboCup se erige como una fuerza motivadora para potenciar la investigación en el área de inteligencia robótica, facilitando un problema estándar para toda la comunidad, donde la meta final del proyecto conjunto es la de construir un equipo de once robots humanoides capaces de enfrentar y vencer al campeón mundial de fútbol (humano) en 2050. Desde la primer RoboCup en 1997, ha tenido un crecimiento continuo como proyecto conjunto de investigación global, donde participan alrededor

de 4000 investigadores de 40 países y regiones de todo el mundo, y es uno de los proyectos más ambiciosos del siglo XXI. Actualmente se divide en 3 disciplinas, por sus nombres en inglés: *RoboCup Soccer*, apuntando directamente hacia el objetivo final descrito anteriormente, *RoboCup Rescue*, una aplicación con fines sociales en actividades de rescate ante cualquier tipo de desastres, y *RoboCup Junior*, una iniciativa educativa internacional para introducir a los jóvenes estudiantes a la robótica. Además, recientemente se sumó la disciplina *RoboCup @Home* con el objetivo de promover el desarrollo de la robótica aplicada a las situaciones más cotidianas como artefactos de uso doméstico.

2.1.1. Categorías de RoboCup Soccer

La competición RoboCup está organizada en diferentes ligas según área de interés, y divisiones en áreas más específicos dependiendo de la rama. Para el presente caso de estudio, los robots utilizados corresponden, dentro de la competencia *RoboCup Soccer*, a la liga *Humanoid*, que concentra robots con aspecto humanoide. Dentro de esta liga, existen 3 divisiones en categorías que segmentan este tipo de robots según una serie de restricciones, siendo la principal el tamaño de los robots: *Kid Size* con robots de altura entre 40 y 90 cm, *Teen Size* con una altura de entre 80 y 140 cm, y *Adult Size* con una altura de entre 130 y 180 cm.

Estas diferentes categorías dentro de la competencia de fútbol de robots humanoide fueron concebidas con el fin de atacar diferentes aspectos del juego de fútbol de cara a fortalecer el aporte para impulsar la visión para 2050 de RoboCup: vencer al mejor equipo a nivel mundial en fútbol tradicional, con futbolistas humanos. En los comienzos, existían las categorías *Kid Size* y *Teen Size*, y la competencia consistía únicamente en tandas de tiros desde el punto penal. Aquí el foco estaba puesto en el posicionamiento del jugador para patear y la coordinación de movimientos para ejecutar la acción de patear, y luego en el reconocimiento de la trayectoria de la pelota y la ejecución de la acción de obstrucción por parte del arquero. A partir de 2005, en la categoría *Kid Size* comenzó la disputa en forma de partidos de fútbol completos, con reglas semejantes al fútbol tradicional, con equipos de 4 jugadores: un arquero y 3 jugadores de campo. En 2010 la categoría *Teen Size* le siguió el rumbo, adoptando idéntica forma de disputa, aunque en escenarios de mayor tamaño, acorde a la diferencia de tamaño en los propios jugadores, y con equipos de tan solo 2 jugadores. Esto motivó la creación de la categoría *Adult Size* en el año 2010, donde se introdujeron robots de tamaño humano (entre 130 y 180 cm), y con una forma de disputa de *shot-goal* como se utilizó durante un período en la MLS (*Major League Soccer*, primera división del

fútbol profesional de Estados Unidos) del fútbol real. Esto consiste en un enfrentamiento uno a uno entre el atacante y el guardametas, con el atacante partiendo desde el centro de la cancha en posesión del balón y disponiendo de un determinado tiempo acotado para culminar la jugada, que puede terminar eventualmente en gol, o simplemente con el tiempo agotado, la pelota fuera del terreno de juego o en posesión del arquero.

Los enfrentamientos entre equipos se disputan en canchas con dimensiones según la categoría. En el año 2014 se actualizaron por última vez las dimensiones del terreno de juego, donde se utiliza el mismo campo para las categorías *Teen Size* y *Adult Size*, y las dimensiones para la categoría *Kid Size* cada vez se asemejan más a las de las categorías de mayor tamaño. Además, se utilizan elementos distintivos para identificar los límites de la cancha con unas *marcas* cilíndricas en un código de colores predeterminado en conjunto con las líneas que delimitan el terreno. Los jugadores a su vez, se diferencian según equipos por el color de vestimenta que los cubre. De esta forma, un robot puede identificar a otro en su campo visual como *compañero de equipo* o como *rival*.

Cuadro 1: Dimensiones del campo de juego (en cm)

	Objeto	**KidSize**	**TeenSize & AdultSize**
A	Campo: largo	900	900
B	Campo: ancho	600	600
C	Arco: profundidad	50	60
D	Arco: ancho	180	260
E	Área: largo	60	100
F	Área: ancho	345	500
G	Punto penal: distancia	180	210
H	Círculo central: diámetro	150	
I	Líneas: ancho mínimo	70	

Durante este proyecto, la plataforma de trabajo utilizada se circunscribió a las restricciones de la competencia *RoboCup Humanoid Soccer* en su categoría *Kid Size*. En líneas generales, estas restricciones determinan la cantidad de jugadores y las reglas del juego, e imponen sobre el robot condiciones pseudo humanas tanto para la ejecución de los movimientos como para la disposición, cantidad y variantes de sensores.

2.1.2. Restricciones para RoboCup Humanoid KidSize

El presente proyecto se desarrolló en el marco de las regulaciones impuestas para la categoría *Kid Size* según se describe brevemente a continuación.

El robot debe tener una altura de entre 40 y 90 cm, y no existen restricciones en cuanto al peso del mismo.

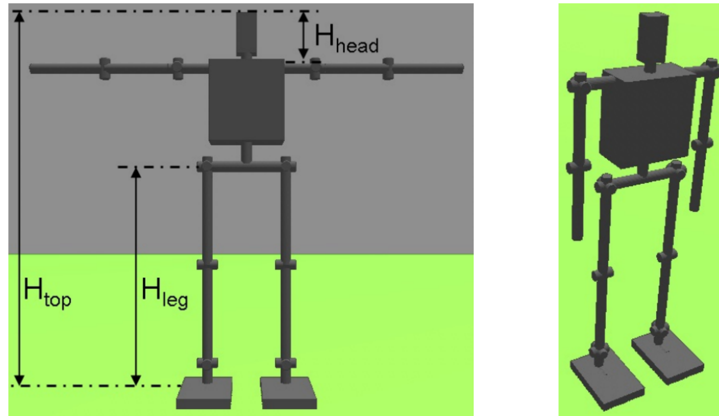

Figura 1: Ejemplo de esquema de robot humanoide

1. **Altura**: El robot debe medir entre 40 y 90 cm de altura.

2. **Peso**: Actualmente no existen restricciones para este ítem.

3. **Tamaño**:

   a) La superficie de cada pie debe ser menor o igual a $(2{,}2 \cdot H_{com})^2 /32$.

   b) El cociente entre largo y ancho del rectángulo que circunscribe la superficie del pie debe ser menor o igual a 2.5.

   c) El robot completo debe *"caber"* en un cilindro de diámetro $0{,}55 \cdot H_{top}$

   d) La suma de los largos de ambos brazos y el ancho del torso debe ser menor a $1{,}22 \cdot H_{top}$. Ambos brazos deben tener el mismo largo.

   e) No existe una configuración de los actuadores tales que el robot se extienda más de $1{,}5 \cdot H_{top}$.

   f) El largo de las piernas (H_{leg}) incluyendo los pies cumple que: $0{,}35 \cdot H_{top} \leq H_{leg} \leq 0{,}70 \cdot H_{top}$.

   g) El largo de la cabeza (H_{head}) incluyendo el cuello cumple que: $0{,}05 \cdot H_{top} \leq H_{head} \leq 0{,}25 \cdot H_{top}$.

   h) El largo de las piernas se debe medir con el robot parado y se mide desde la primera articulación de la pierna contra el torso y hasta la punta del pie.

4. **Sensores**: los equipos son alentados a incorporar a su plataforma sensores que tengan una funcionalidad equivalente a las capacidades sensoriales del ser humano. Asimismo, se deberán colocar en una posición semejante a los sensores biológicos. En particular:

   a) El único sensor externo activo permitido es el sonido, con características humanas en relación a volumen y rango de frecuencias emitidas, y con un único dispositivo emisor de sonido (parlante) que puede ubicarse en la cabeza, cuello o tronco. No está permitido ningún otro tipo de sensor externo activo como leds, ni emitiendo luces o cualquier tipo de onda electromagnética.

   b) Sensores externos como cámaras y micrófonos no pueden ubicarse en las extremidades o torso del robot. Deberán colocarse en la cabeza o cuello.

   c) El campo de visión del robot está limitado a 180°, considerando el conjunto de cámaras que posea el robot como un único campo de visión. A su vez, el movimiento de *pan-tilt* de la cabeza también está restringido para ser semejante al del ser humano. El *pan* (horizontal) está limitado a 270°, lo que implica un movimiento de ±135° en relación al centro; mientras que el *tilt* (vertical) está limitado a ±90° en relación a la línea horizontal. Esto implica que, considerando al robot posicionado sobre la marca central de la cancha, no podrá ver los 2 arcos simultáneamente en ninguna posición corporal ni con ninguna combinación de ángulos de *pan* y *tilt*.

   d) La cantidad de cámaras está limitada a una configuración de visión estéreo (máximo 2 cámaras).

   e) Sensores de tacto, fuerza y temperatura pueden ser colocados en cualquier posición del robot.

   f) Sensores internos al robot pueden medir cualquier parámetro de interés, incluyendo (y no limitado a) voltajes, corrientes, fuerzas, movimientos, aceleración, campo magnético y velocidad rotacional. Pueden estar ubicados en cualquier parte dentro del robot.

5. **Comunicación y control**:

   a) Mientras se disputa el partido, los robots deben participar de forma autónoma. Sin alimentación externa, control remoto, o cualquier agente de inteligencia externo.

b) Los robots se podrán comunicar únicamente a través de la WLAN que brinda la organización del torneo, y el ancho de banda de todo el equipo no deberá superar 1 Mbit/s. Los robots no deberán depender ni confiar en la disponibilidad y confiabilidad de la red. Deberán poder desarrollar el juego tanto si la red no se encuentra disponible o tiene un bajo nivel de servicio. Solamente los robots están autorizados a comunicarse a través de esta WLAN, y ningún otro componente externo podrá acceder. Además, no deberá existir ninguna otra red WLAN habilitada, y las comunicaciones internas de los equipos podrán hacerse vía LAN.

c) Los robots podrán comunicarse entre ellos en cualquier momento durante el juego. Cualquier comunicación de un agente externo está prohibida. La única posibilidad que existe para supervisar externamente el comportamiento y la comunicación de los robots, es recibiendo desde los robots paquetes UDP, conectándose con una computadora al router inalámbrico oficial de la competición vía una *tethered LAN*. El envío de cualquier tipo de información hacia el robot, debe ser hecho durante los tiempos de interrupción de la competencia, o usando un cable directo entre el robot y la computadora en el tiempo de mantenimiento del robot.

d) Durante el juego se utilizará un controlador oficial (*referee box*). Éste utiliza UDP para enviar como *broadcast* la información a los robots, incluyendo tiempo transcurrido del juego, marcador, estado actual del juego (que pueden ser: *ready, set, playing, finished*), e información específica de robots penalizados. El código es abierto, y para incentivar el uso de este recurso, a los equipos que la utilicen, se les otorgará 15 segundos de ventaja luego de cada interrupción del juego.

e) No se autoriza la presencia de personas en el campo de juego mientras el juego se encuentra en marcha. Para manipular un robot, se debe pedir autorización al árbitro del juego previo a entrar al campo. Cada equipo debe designar únicamente una persona como manipulador de los robots. El manipulador no podrá tocar un robot del equipo contrario, para evitar cualquier daño al robot en cuestión.

6. **Colores y marcas**:

a) Los robots participantes deberán tener color negro o gris oscuro y opaco (es decir, que no reflejen la luz). Podrán también ser

plateados (en color tipo aluminio), grises o blancos, en cuyos casos los pies deben ser negros. Cualquier color utilizado en el campo, pelota, otros robots o marcas, debe ser evitado.

b) Los robots deberán estar distinguidos con marcas del equipo. Estas marcas son de color magenta para un equipo y cian para el otro. Piernas y brazos del robot deberán estar cubiertos por el color que corresponda. De cada lado del robot, debe ser observable al menos una marca en el brazo y una en la pierna. Las marcas deberán tener al menos 5 cm de alto, y un ancho acorde a las dimensiones del brazo o pierna.

c) Los robots de cada equipo deberán estar identificados de forma que puedan ser fácilmente distinguidos uno a uno: deben estar marcados con nombres o números. El robot que cumpla el rol de arquero debe tener una marca única que lo distinga del resto de los robots de su equipo.

7. **Seguridad**: Los robots que participen de la competencia *Humanoid League* no deben representar ningún peligro para seres humanos, otros robots, o el campo de juego. Esto incluye específicamente configuraciones de armado del robot que son objetivamente capaces de causar estos daños, por ejemplo con puntas sobresalientes del cuerpo del robot. Cualquiera está autorizado a tomar las medidas necesarias para prevenir cualquier daño urgente.

8. **Robustez**: Los robots participantes en la *Humanoid League* deben estar construidos de forma robusta. Deben mantener la integridad estructural durante el contacto con el campo, la pelota, u otros jugadores. Sus sistemas sensoriales deben ser capaces de tolerar niveles significativos de ruido y perturbaciones causadas por otros jugadores, los árbitros, los manipuladores de robots, e incluso la audiencia.

9. **Manipulación**: Se recomienda que los robots cuenten con un mango incorporado a la zona del cuello que permita levantarlo verticalmente. Deberían tolerar esto sin causar daños al robot ni a la persona que lo levanta.

El detalle completo de la regulación utilizada en este proyecto, vigente para la competición RoboCup Fútbol Humanoide 2014 [6] está disponible en el sitio oficial de la competencia [7].

2.2. Marco teórico

La primer etapa del proyecto tuvo fuerte énfasis en la investigación de las temáticas relacionadas con la comunicación e implementación de coordinación en sistemas multi-robot. Se elaboró un documento de estado del arte [8] en relación a estas áreas de conocimiento, que aborda en forma detallada las principales líneas de abordaje para este tipo de problemas.

La investigación se condujo en etapas de construcción del conocimiento, comenzando por el estudio de un componente de robótica, denominado agente, y luego sistemas multi-agente, las definiciones de cooperación y coordinación, y el rol de la comunicación en estos aspectos. Se estudiaron una serie de modelos teóricos de cooperación, incluyendo modelos basados en lógica difusa y la implementación de modelos de robótica de enjambre.

Adquirir un nivel de conocimiento básico en el área posibilitó abordar el estudio de proyectos previos en el área que pretendieron resolver la misma problemática. Se estudiaron algunos de los proyectos que marcaron el liderazgo en las competencias internacionales en los últimos años, y finalmente se estudió el proyecto *Forrest* [1] llevado a cabo en la Facultad de Ingeniería de la República. De aquí se tomaron las principales ideas para la construcción de la arquitectura del sistema propuesto por este proyecto, particularmente el concepto abstracto de "modelo del mundo". Este concepto implica encapsular una lógica que mantiene y actualiza constantemente un modelo de la realidad que rodea al robot. Por un lado toma como insumo la capacidad sensorial del robot, incluyendo visión y comunicación, y por otra parte se alimenta de la propia inteligencia artificial del software que enriquece el modelo. A su vez, realiza una serie de simplificaciones que permiten abstraer la realidad hacia un modelo implementable con los recursos disponibles. La utilidad de este componente debe ser el de brindar información precisa de la realidad a partir de los datos que conforman el modelo. Datos de la posición relativa de otros objetos como pelota, jugadores rivales y compañeros, arcos, señalizaciones del campo, entre otros.

El proyecto "Forrest" no se implementó sobre robots reales, sino que está basado en el *framework UvA Trilearn* [9], que implementa un equipo de robots simulado. Esto implica que los agentes que participan de este modelo son capaces de tener completo conocimiento de la realidad que los rodea, aspecto que no aplica para el caso de estudio de este proyecto donde se trabajó con robots reales.

2.3. Software reutilizado

El desarrollo de la solución se apoyó en una serie de proyectos previos en el área, desarrollados por la Facultad de Ingeniería de la Universidad de la República, y buscó retomar estas líneas de trabajo.

2.3.1. Visión de robots I y II

Primeramente, el proyecto "Visión de robots (VisRob)" [10] abordó la problemática de la captura y procesamiento de imágenes para el reconocimiento de objetos, su ubicación y distancia. Este primer proyecto, en 2003, tomó como plataforma base de trabajo el esquema de la categoría *Middle League MiroSot de FIRA* donde, en cancha, los 5 robots por equipo son monitorizados continuamente por una única *webcam* en la parte superior de la cancha, y conectada directamente a una computadora personal.

Luego, y retomando el trabajo realizado por *VisRob* [10] especialmente en la aritmética de la solución, el proyecto "Visión robótica y reconstrucción espacial con aplicaciones prácticas (VisRob 2)" [11] amplió la librería para la captura y procesamiento de imágenes portando una versión para la plataforma *HaViMo*; módulo de hardware de visión original del sistema *Bioloid Premium Kit*. El reconocimiento de objetos está dado por una configuración paramétrica originalmente establecida para el fútbol de robots en la categoría *KidSize* de *RoboCup*. Esta librería incluyó además características primitivas de comunicación entre robots a través de Zigbee, protocolo de comunicación alternativo al popularmente conocido WiFi, utilizado nativamente por la plataforma Bioloid.

2.3.2. Salimoo: Fútbol de robots para la liga humanoide de RoboCup

Simultáneamente, en el proyecto "Salimoo: Fútbol de robots para la liga humanoide de RoboCup" [12] se implementó la lógica de comportamiento para el sistema *Bioloid Premium Kit* en su conjunto y sobre la plataforma original: la placa *CM-510* y el módulo de visión *HaViMo*. Este trabajo representa una base sólida para la línea de investigación que se abordó en este proyecto, ya que contempla todos los desafíos esenciales para el juego de fútbol de robots. Conformó una solución integral para el sistema Bioloid en su plataforma original, incluyendo la implementación de la movilidad, la habilidad sensorial para captura y procesamiento de información utilizando la librería *VisRob 2* [11], con la estrategia de comportamiento del sistema en su conjunto modelado por un software con inteligencia artificial y desarrollado

íntegramente por el grupo de investigación de la Facultad de Ingeniería de la UdelaR.

2.3.3. Futbot: Fútbol de robots humanoides

Seguidamente, surgió la iniciativa de trabajar sobre la liberación de las restricciones impuestas por la plataforma y el giro hacia un paradigma de componentes de uso abierto, alineado con el paradigma propuesto por la Universidad de la República y el gobierno mismo a través de su agencia promotora de asuntos referentes a la innovación, AGESIC (*"Agencia para el Desarrollo del Gobierno de Gestión Electrónica y la Sociedad de la Información y del Conocimiento"*). Estos objetivos se abordaron primeramente en el proyecto proyecto "FutBot: Fútbol robótico humanoide" [13], para luego ser retomadas por este proyecto. En el proyecto *FutBot* [13], luego de evaluar diferentes alternativas, se propuso la siguiente sustitución de componentes: la placa CM-510 por una Raspberry Pi [14], y el módulo de visión HaViMo por una cámara web Logitech C920. Esto dio lugar a la necesidad de portar la solución desarrollada hacia el nuevo hardware, sustituyendo la totalidad de las librerías utilizadas y modificando gran parte de las interfaces entre las diferentes capas del software. Asimismo, la sustitución del hardware de visión por uno de prestaciones ampliamente superiores, permitió desarrollar en forma evolutiva la solución presente al momento de comenzar el proyecto actual.

Como continuación de la línea de trabajo establecida por estos proyectos fundacionales en la rama, se debió retomar la adaptación de los nuevos componentes de uso abierto introducidos por *FutBot* [13]. Esto involucra en forma transversal el conjunto de la solución construida por los diferentes proyectos que repasamos.

2.3.4. Forrest: Desarrollo de un equipo de fútbol de RoboCup

Retomando la visión histórica enmarcada en los proyectos *VisRob* [10], *VisRob 2* [11], y *Salimoo* [12]; en una línea paralela el proyecto *Forrest* [1] se centró en los aspectos de inteligencia artificial y coordinación en un equipo con múltiples robots en cancha. Por contrapartida con el resto de los proyectos, se abstuvo de las restricciones de hardware, implementando la solución sobre un *framework* que permite emular sistemas de robots. Esto le dio a *Forrest* [1] la posibilidad de contar con un número más amplio de robots -la solución propuesta contempla 11 robots por equipo-, lo que dio pie para una implementación de roles de robots que varía en forma dinámica con el transcurso del juego, y la implementación de una estructura de razonamien-

to en 2 niveles. Básicamente este esquema se resume en evaluar las acciones a tomar en función de un valor de ponderación y complementar la evaluación agregando a cada opción la evaluación del conjunto de futuras acciones disponibles en el caso hipotético de ejecutar en primer lugar la acción previa.

3. Plataforma de trabajo

La premisa de trabajo propuesta implica la liberación de restricciones de componentes propietarios en toda su extensión, tomando como plataforma base el kit de robótica *Bioloid Premium Kit*. Primeramente se plantea incorporar un sistema operativo de código abierto que permita escalar el desarrollo de la solución, lo que trae aparejado la sustitución de la placa base del sistema original por una de arquitectura abierta. Luego de analizar alternativas, la opción elegida fue la placa Raspberry Pi, lo que abrió la posibilidad de, mediante la conectividad USB que ofrece, incorporar una nueva cámara con mayores prestaciones técnicas para la captura y procesamiento de imágenes como lo es la Logitech C920 y también un componente de hardware dedicado para la comunicación mediante WiFi en una tarjeta de red USB.

3.1. Kit robótico Bioloid Premium

El kit de robótica *Bioloid Premium* incluye todos los componentes necesarios para la construcción de un robot humanoide programable y completamente funcional. La placa controladora del kit, CM-510 [15], incorpora las interfaces con los actuadores y con componentes adicionales como sensores externos, incluyendo giroscopio, cámara o un módulo de comunicación Zigbee. La alimentación está resuelta con una batería de litio que trabaja a 11.1v, que potencia los actuadores Dynamixel AX-12.

A su vez, al ser un kit completamente modular, cuenta con la versatilidad de ser ensamblado en múltiples configuraciones con formatos, incluyendo algunos que tienen su correspondiente en el mundo real, además del formato humanoide, como ser: perro, dinosaurio, araña, escorpión; además de cualquier otra configuración que el usuario desee. Entre las características más destacadas, se encuentran: sensores varios incluyendo sensor de distancia, receptor infrarrojo, puertos de entrada-salida para agregar otros sensores; control de mando a distancia (mediante protocolo *"wireless Zigbee"*); comportamiento programable en lenguaje similar a C; aprendizaje de movimientos mediante software específico *RoboPlus*.

La controladora CM-510 [15] es una placa programable a través del software propietario de la plataforma, *RoboPlus*, que permite el desarrollo con un estilo de programación similar al lenguaje C de módulos independientes implementando acciones, llamados tareas, que son luego combinables entre sí. Como alternativa a la programación en *RoboPlus*, existe la posibilidad de desarrollar un firmware para controlar la placa y los motores Dynamixel utilizando C embebido. El microcontrolador incluido en la placa es el Atmel ATMega2561 de 8-bit y 16 MHz [16]. La placa cuenta con puertos de co-

nexión con los actuadores Dynamixel AX-12 con 3 vías: DATA, Vcc, GND. Además, cuenta con conexiones para integración con una computadora personal y alimentación a desde fuente externa.

Los actuadores que componen el kit son motores Dynamixel AX-12 [17], con un ángulo de giro de 300° en pasos de 0,29° ($2^{10} = 1024$ posiciones) y una velocidad máxima de giro de 59 rpm. Estos motores se conectan en forma serial encadenada (*daisy chain*) directamente a la placa controladora del kit, desde la que reciben la alimentación e intercambian datos a través de un único bus que sirve tanto para el envío como para el recibo de datos en modo semidúplex (*half-duplex Rx/Tx*). A través de este bus, la placa controladora coloca los paquetes de datos a ser enviados hacia los motores, indicando qué motor es el destinatario del mensaje, pudiendo ser uno específicamente o todos los motores. El mensaje es colocado en forma de broadcast en los buses de datos y todos los motores lo reciben. Luego, la controladora de cada motor resuelve si descartar o adoptar el mensaje según la identificación del destinatario y el identificador del propio motor. De igual modo, cuando un motor envía una respuesta hacia la controladora, estos datos son colocados en el mismo bus de datos de por el que recibe la información desde la controladora. Es decir que debe existir una coordinación completa entre los intervalos de envío y recepción de datos desde la controladora principal del sistema, la placa CM-510. El incumplimiento de esta restricción podría llevar a colocar datos de entrada y de salida en el bus simultáneamente y provocar daños eléctricos en ambos componentes: actuadores y placa controladora.

3.2. Placa controladora Raspberry Pi

La Raspberry Pi [14] es una pequeña computadora del tamaño de una tarjeta de crédito formada por una placa única. Puede conectarse a un monitor de computadora o a una televisión y es compatible con un teclado y ratón estándar.

La placa es desarrollada por la Fundación Raspberry Pi, registrada en el Reino Unido como una organización benéfica con fines educativos. La meta de la Fundación es el avance en la educación de niños y adultos particularmente en el campo de las computadoras, ciencia informática y afines.

El primer modelo, llamado *Raspberry Pi 1 Modelo B*, fue lanzado en abril de 2012 con un precio de mercado de U$S 35, y fue seguido por dos versiones más económicas, *Raspberry Pi 1 Modelo A* y *Raspberry Pi 1 Modelo A+*, lanzadas en 2013 y 2014 y a precios de U$S 25 y U$S 20 respectivamente.

Las 3 versiones del primer modelo de Raspberry comparten placa base, procesador y memoria RAM: Broadcom BCM2835, ARM11 76JZF-S single core 700MHz, 256 MB RAM. Luego del lanzamiento original, el modelo B

fue dotado con 512 MB RAM.

Luego, en febrero de 2015, la Raspberry Foundation lanzó un nuevo modelo, la *Raspberry Pi 2, Modelo B* con una serie de importantes mejoras tanto en rendimiento como en el manejo de la carga eléctrica y la demanda de consumo en picos de actividad, sobretodo a través de los puertos de entrada/salida como son los puertos USB y el puerto RJ-45. Los principales componentes de hardware introducidos en este modelo son el aumento al doble de la memoria RAM disponible, quedando en 1 GB RAM, y el procesador ARMv7 Cortex-A7 900 MHz quad-core. Las características completas incluyen: 4 puertos USB 2.0, GPIO de 40 pines, puerto full HDMI, puerto ethernet, salida de audio con conector de 3.5 mm, salida de video compuesto, interfaz para cámara (CSI), interfaz para display digital (DSI), slot para tarjetas de memoria Micro SD, GPU de gráficos 3D VideoCore IV.

La implementación del proyecto, en el marco de la liberación de los componentes de uso propietario, y tomando como base la plataforma Bioloid Premium Kit, supuso la sustitución de la placa controladora original por una Raspberry Pi. El modelo adoptado finalmente fue la *Raspberry Pi 2, Modelo B*. La característica fundamental que supuso la elección de esta versión es la capacidad de manejar múltiples hilos de ejecución a nivel de hardware gracias a su procesador de 4 núcleos.

Existen diversas adaptaciones (*"portings"*) de sistemas operativos que pueden ejecutarse sobre la placa *Raspberry Pi*, entre los que se destacan: Ubuntu Mate (versión desktop), Snappy Ubuntu Core (para desarrolladores), Windows 10 IoT Core (*Internet Of Things*), y Raspbian [18]. Este último es el sistema operativo oficial para la Raspberry Pi Foundation, soportado por la Raspberry Pi. Está basado en Debian e incluye una serie de software didáctico preinstalado.

Para el desarrollo del proyecto, se utilizó como sistema operativo base el *Raspbian Wheezy*, basado en *Debian Wheezy*.

3.3. Cámara web Logitech C920

El proyecto *FutBot* [13], en su instancia de relevamiento y selección de componentes para un nuevo diseño del sistema [19], partiendo de un módulo de visión HaViMo 1.5 determinó la sustitución de éste por una cámara web Logitech C920.

Las principales características que motivaron esta elección, considerando la disponibilidad de los recursos en Uruguay, fueron las capacidades de grabación de video de la cámara Logitech C920 con una resolución de 1920x1080

píxeles y a una velocidad de 30 FPS[1] en comparación con los escasos 160x120 píxeles a 8 FPS del módulo HaViMo 1.5. Por otra parte, la compatibilidad de la cámara Logitech C920 con sistemas operativos basados en Linux, la librería OpenCV, y el lente con ajuste de foco mecánico automático fueron otras características de la cámara Logitech C920 que motivaron la elección.

Dado que se buscó optimizar los recursos de hardware disponibles para el sistema, un aspecto importante de la cámara Logitech C920 es la capacidad de realizar internamente el procesamiento del video utilizando el algoritmo de compresión (códec) H.264 [20] mediante hardware dedicado. En contraste con esto, las cámaras usualmente capturan video en formato descomprimido y envían directamente el flujo de datos hacia el componente conectado, en este caso la Raspberry Pi 2, y es el CPU externo el encargado de realizar el procesamiento y compresión de las imágenes.

Como último detalle, el lente Carl Zeiss con foco mecánico automático de 20 posiciones ofrece un gran rendimiento en diferentes condiciones de luz y exposición [21], lo que asegura una identificación más certera de los objetos en el rango de visión y disminuye el rango de dispersión de colores.

3.4. Sistema operativo base: Raspbian

El objetivo con el que fue concebido Raspbian [18] es el de convertirse en la principal alternativa de sistema operativo base para la Raspberry Pi.

Raspbian procura mantenerse lo más cercano y similar a Debian dentro de las posibilidades. Debian es un sistema operativo con vasta documentación utilizado diariamente por millones de usuarios en el mundo, lo que alimenta una amplia base de conocimientos en constante crecimiento a lo largo de la web. Por ende, casi cualquier información que aplica a Debian, aplicará a la misma versión de Raspbian. Información para versiones previas de Debian podría aplicar también, pero contemplando eventualmente algún ajuste tal como si se pretendiera aplicar a versiones más nuevas del propio Debian.

Más específicamente, el sistema operativo Raspbian es una adaptación no oficial de "Debian Wheezy armhf", con ajustes en la compilación para producir código optimizado para arquitecturas *hard float* que ejecutará sobre la Raspberry Pi. Esto ofrece un rendimiento significativamente mejor para aplicaciones que hacen uso de operaciones de aritmética de punto flotante. Otro tipo de aplicaciones también se ven beneficiadas con un mejor rendimiento gracias a la utilización de instrucciones avanzadas del CPU ARM de la Raspberry Pi.

Raspbian surgió originalmente gracias al trabajo de dos desarrolladores:

[1]FPS: cuadros por segundo, por su sigla en inglés *Frames Per Second*.

Mike Thompson (mpthompson) y Peter Green (plugwash)[2]. Actualmente se sustenta en una comunidad creciente de miembros de la comunidad Raspberry Pi que buscan obtener el máximo rendimiento de sus dispositivos.

3.5. Placa de integración zAX12pi

La sustitución de la placa controladora del kit Bioloid Premium, el microcontrolador CM-510, por una placa Raspberry Pi 2 constituyó un cambio de alto impacto en la arquitectura de hardware original; se sustituyó el componente encargado de la comunicación con los actuadores del sistema, los motores Dynamixel AX-12, y con los sensores externos como cámara web y dispositivos de red.

Dado que tanto la cámara web como la placa de red inalámbrica utilizadas tienen una interfaz USB, y que la Raspberry Pi 2 cuenta con 4 conectores USB 2.0, la integración de estos componentes a nivel de hardware se puede considerar de menor dificultad. Por otra parte, considerando los motores y la forma que éstos se conectan a la placa CM-510, surgieron dos puntos de incompatibilidad con la Raspberry Pi 2 que debieron ser resueltos.

Por un lado, la placa CM-510 cuenta con 5 conectores AX-12 de 3 pines que representan, en cada cable, (Vcc), (GND) y (DATA), y que permiten conectar conjuntos disjuntos de motores interconectados en disposición serial. El bus de transferencia de datos (DATA) es a través de un puerto serial en modo semi-dúplex. Esto quiere decir que la placa CM-510 envía y recibe datos hacia y desde los motores AX-12 por un mismo bus. En contraposición, la placa Raspberry Pi 2 ofrece como interfaz serial dos puertos Rx y Tx para la recepción y envío de pulsos eléctricos en el bus de datos, es decir, un bus serial dúplex completo (*full-duplex*). Por lo tanto, para lograr la conexión de los motores con la placa, surge la necesidad de desarrollar una interfaz de hardware capaz de soportar la comunicación desde la Raspberry Pi 2 hacia los AX-12 y viceversa, mediante la conexión entre el bus semi-dúplex de los motores AX-12 y el bus dúplex completo de la Raspberry Pi 2. Esto representó un gran desafío a nivel tecnológico, tanto eléctrico como de software, ya que se debió implementar la sincronización del canal de transferencia de datos en forma rigurosa. Este fue el punto de mayor sensibilidad en la integración tecnológica de nuevos componentes al kit ya que cualquier falta de sincronización podría provocar comportamientos inesperados por fallas en el envío o recepción de información de los motores, y hasta incluso causar un daño permanente de los componentes de hardware involucrados: actuadores AX-12 y placa Raspberry Pi.

[2] Raspbian Team: `https://www.raspbian.org/RaspbianTeam`

La primer solución propuesta fue la placa denominada *"zAX12pi"* que surgió en el marco del proyecto *FutBot*[13]. Constaba de una sencilla placa que resuelve los aspectos eléctricos y de conectores mediante la construcción de un circuito compuesto únicamente por 3 resistencias y 7 conectores MOLEX hembra, y 4 pines para las conexiones de entrada: Vcc, GND, Rx, Tx, según se observa en el diagrama de la figura 2, que representa la versión 4.0 de esta placa.

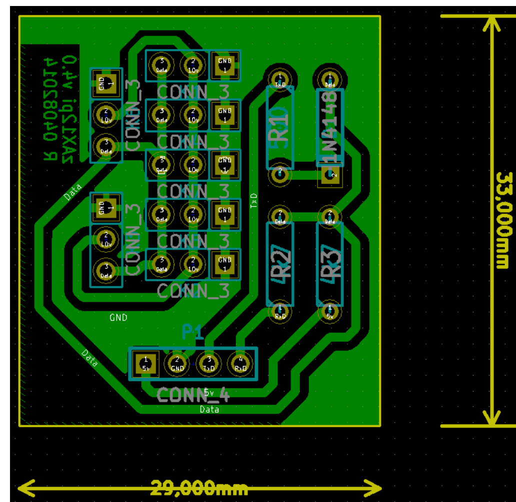

Figura 2: Placa zAX12pi versión final, con resistencias

Sobre esta placa se realizaron pruebas de concepto programando breves rutinas en lenguaje Python sobre la Raspberry Pi para el encendido y apagado de LEDs indicadores de los actuadores. Se realizaron pruebas conectando entre 1 y 5 motores AX-12 combinando entre conexiones en serie y conexiones independientes, desde todos los motores en serie y hasta uno por conector MOLEX. Como resultado, se observó que la placa es capaz de controlar entre 1 y 4 motores dependiendo del escenario, pero a partir de conectar el quinto motor, en ningún escenario el mensaje contenido en el pulso eléctrico llega a destino.

Producto de los ensayos realizados, se profundizó el análisis en función de los componentes a conectar y los problemas encontrados hasta el momento. Se encontraron proyectos similares que evidencian buenos resultados utilizando un buffer tri-estado (*tri-state buffer*), particularmente circuitos que utilizan el chip 74LS241 [22]. En la sección 4.1 se aborda el desarrollo del circuito que implementa esta interfaz de hardware entre los motores AX-12 y la placa Raspberry Pi 2.

4. Descripción de la solución de hardware

El desarrollo de la solución se divide en dos grandes componentes: hardware y software. Esta sección profundiza sobre los aspectos de hardware que se desarrollaron para hacer posible la integración de los diferentes componentes de la plataforma: el kit Bioloid Premium, la placa Raspberry Pi 2 y sus periféricos de video y red, y la fuente de alimentación.

4.1. Placa Pi2AX12: interfaz entre motores y placa

El principal desafío a nivel de hardware supuso la integración de la nueva placa controladora del sistema, la Raspberry Pi 2, en sustitución de la placa CM-510 original del kit Bioloid Premium.

Si bien el problema comenzó a ser abordado en un trabajo previo como *FutBot*[13], los resultados obtenidos en esa instancia no fueron concluyentes: se logró controlar una pequeña cantidad de motores, en escenarios de baja exigencia, y con una alta tasa de fallas de comunicación.

Como resultado de un análisis profundo sobre los aspectos de bajo nivel a resolver, se identificó el problema en forma generalizada: permitir la interacción de dos componentes de hardware, motores y Raspberry Pi 2, que utilizan bus de datos semi dúplex y dúplex completo respectivamente. Para lograr esto, el circuito eléctrico a desarrollar debía contar con un componente que resuelva esto: un buffer tri-estado.

Se trabajó entonces sobre el diseño de una nueva placa, bajo el nombre de "Pi2AX12", con conectores MOLEX para los actuadores Dynamixel, y que implementara una interfaz de comunicación entre un bus serial dúplex completo (interfaz de la placa Raspberry Pi) y un bus serial semi dúplex (interfaz de los motores AX-12). Para esto, como se observa en la primer versión del circuito en la figura 4, se conectaron los puertos de entrada/salida de la Raspberry Pi a los conectores de la derecha del chip 74LS241. Luego, se diseñó el circuito para colocar los conectores MOLEX hembra conectados al sector izquierdo de la placa 74LS241. En el cuadro 2 se muestra la correspondencia de conexiones tomando como referencia el los puertos del chip 74LS241 que se observa en la figura 3.

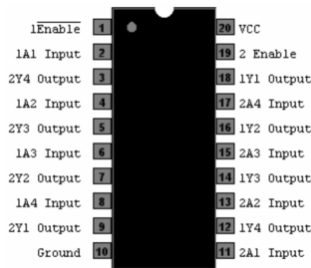

Figura 3: Esquema del buffer tri-estado 74LS241

Cuadro 2: Conexiones de la placa Pi2AX12

N	**74LS241**	**Molex**	**RPi**
1	1Enable		GPIO18
2	1A1 Input	3	
3	2Y4 Output	3	
10	GND	1	GND
17	2A4 Input		Tx
18	1Y1 Output		Rx
19	2Enable		GPIO18
20	Vcc	2	Vcc

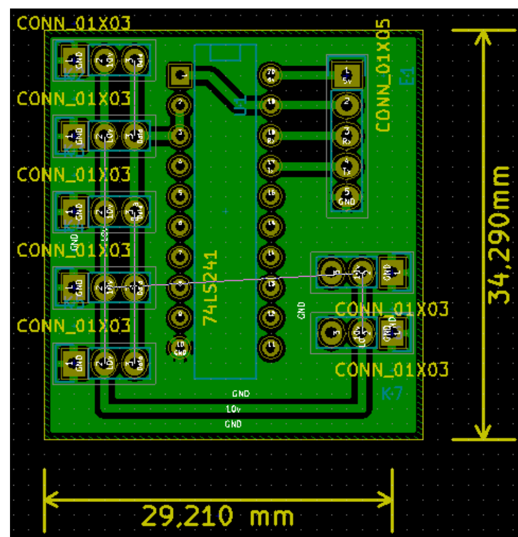

Figura 4: Placa Pi2AX12 versión 0.1, incorporando chip 74LS241

Se completaron una serie de iteraciones sobre el prototipo inicial que

permitieron verificar la prueba de concepto del circuito realizado. Se verificó la solución tanto a nivel eléctrico como su integración a nivel del software.

Luego de esto, se abordó una etapa final orientada al diseño de la placa y la disposición de los componentes, apuntando hacia aspectos más alineados a la conexión del cableado necesario. El diseño propuesto en la versión original (figura 4) dispone 7 conectores MOLEX con la siguiente distribución: 5 a un lado de la placa y 2 sobre el lado opuesto. Al ensamblar el sistema utilizando esta placa, se determinó que la disposición de los conectores no facilita la conexión de los motores y cables de alimentación y datos. Esto motivó un nuevo diseño como el observado en la versión 0.4 en la figura 5 incorporando las siguientes mejoras: placa de tamaño similar a la Raspberry.

- **Tamaño**: placa de tamaño similar a la Raspberry Pi, para permitir el ensamblaje de la Raspberry Pi superpuesta a la placa Pi2AX12.
- **Perforaciones para ensamblado**: se incluyeron perforaciones en la placa para permitir el ensamblado debajo de la Raspberry Pi utilizando tornillos de fijación a la espalda del robot.
- **Disposición de conectores MOLEX**. Se ubicaron los conectores MOLEX en forma similar a la placa CM-510:
  - 2 en el lado superior para conectar los motores de la cabeza y cables de potencia hacia la Raspberry Pi,
  - 2 en cada lado derecho e izquierdo para motores de pierna y brazo de cada lado respectivamente,
  - 1 conector sobre el lado inferior para conectar la fuente de alimentación.

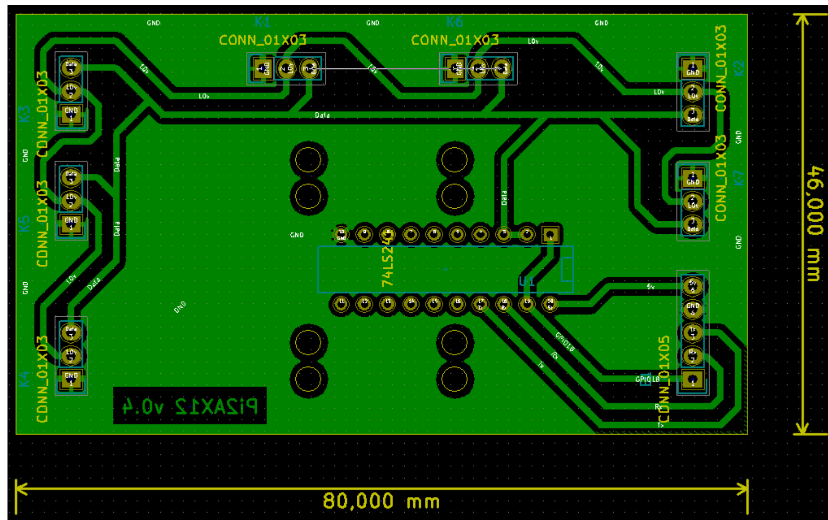

Figura 5: Placa Pi2AX12 versión 0.4, diseño y tamaño ajustados

Finalmente, luego de validar el diseño del circuito, se generó la versión final ajustando levemente las perforaciones y grabando la leyenda con la referencia del proyecto directamente sobre la placa, como se observa en la figura 6.

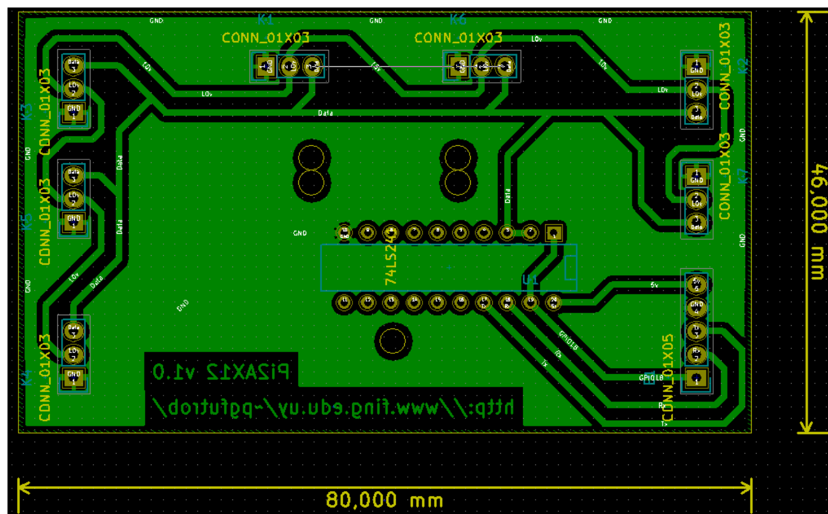

Figura 6: Placa Pi2AX12 versión 1.0 final

La placa "Pi2AX12" fue desarrollada íntegramente en el marco del proyecto, para lo cual se evaluaron y ensayaron diferentes técnicas y procedimientos para la construcción de una placa. Producto de reiterados ensayos de laboratorio, se logró obtener resultados óptimos en la elaboración del circuito y

se desarrolló una documentación que resume estos pasos y reúne una serie de buenas prácticas aplicables.

En las figuras 7 y 8 se observan tomas fotográficas de la placa construida en vistas inferior y superior respectivamente.

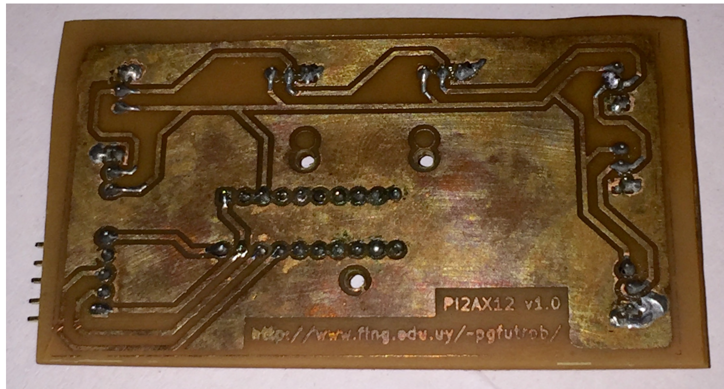

Figura 7: Vista dorso de la placa Pi2AX12 versión 1.0 final

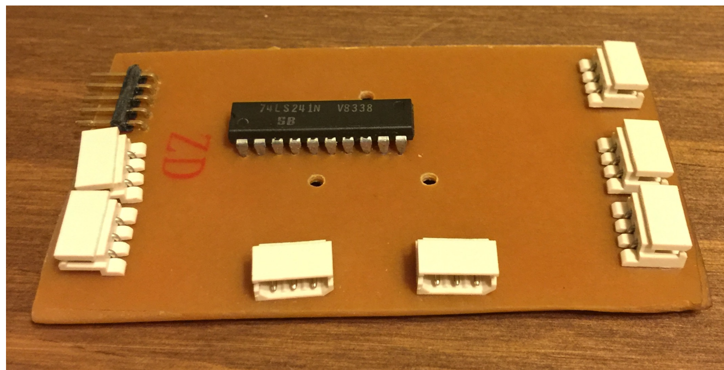

Figura 8: Vista superior de la placa Pi2AX12 versión 1.0 final

4.2. Ensamblado del sistema

Luego de presentados los diferentes componentes que conforman la nueva plataforma introducida por este proyecto, describimos a continuación en el proceso de transformación realizado para ensamblar el kit FutRob. En el documento llamado *Ensamblado y utilización del sistema FutRob* [2] que se incluye como anexo se realiza una descripción pormenorizada del procedimiento de ensamblado y utilización del sistema.

Partiendo del robot original del kit Bioloid Premium, armado en su forma humanoide siguiendo parcialmente las instrucciones del propio manual de uso,

y quitamos la placa controladora y la batería. En las figuras 9 y 10 observamos una vista del frente y reverso del robot respectivamente. En la figura 10 se observan los 5 conectores correspondientes a las diferentes secciones del cuerpo del robot: cabeza y 4 extremidades, 2 inferiores y 2 superiores. Estos cables son los que se utilizaron para conectar a la placa Pi2AX12.

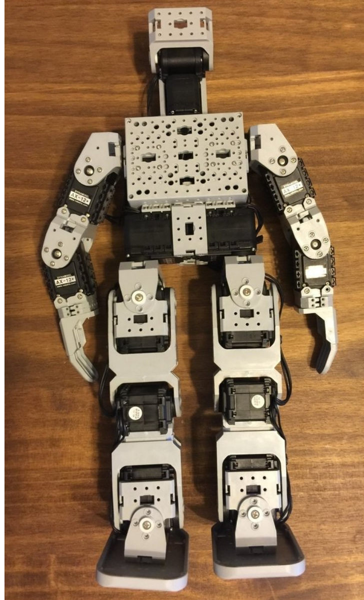

Figura 9: Bioloid básico frente

Figura 10: Bioloid básico reverso

Sobre la espalda se fijaron tornillos que permitieron sujetar la placa Pi2AX12 y la carcasa plástica que contiene la Raspberry Pi a la estructura del robot.

Los 5 cables de actuadores, 4 de extremidades inferiores y superiores y 1 de la cabeza, se conectaron a la placa Pi2AX12 según se observa en la figura 11. Los 2 conectores libres en la placa Pi2AX12 se utilizaron para conectar por debajo la fuente de alimentación externa (batería de litio), y en la parte superior la alimentación para la placa Raspberry Pi. Luego de montar la placa Pi2AX12 (figura 12), se colocó la carcasa de acrílico incluyendo la placa Raspberry Pi dentro.

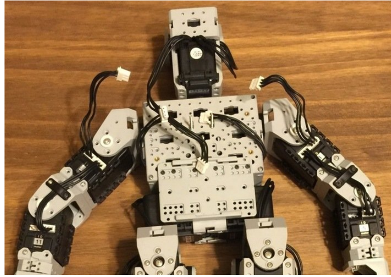

Figura 11: Conexiones de extremidades

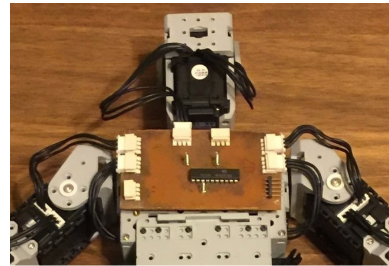

Figura 12: Acoplada Pi2AX12

En las figuras 13 y 14 se observa el robot armado con la nueva placa y la Raspberry Pi de forma tal que se minimiza el impacto sobre la distribución de masa de los componentes, reduciendo en la mayor medida posible el efecto sobre la ubicación del centro de estabilidad de la plataforma en su conjunto.

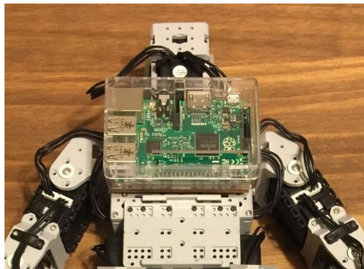

Figura 13: Acoplada Raspberry Pi

Figura 14: Conectada

En la figura 15 se observa, en una vista lateral de la espalda del robot, cómo se combinan los nuevos componentes de la solución: la placa Pi2AX12 y la placa Raspberry Pi dentro de su carcasa plástica. La placa Pi2AX12 logró el objetivo de facilitar las conexiones con los actuadores producto de la disposición de sus conectores MOLEX, sin afectar la ubicación de la placa Raspberry Pi, que se coloca completamente superpuesta.

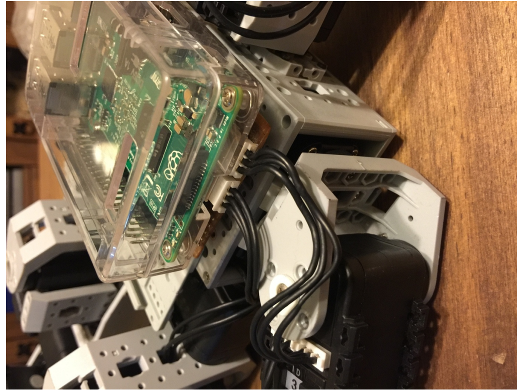

Figura 15: Conectores Pi2AX12

Luego, se conectaron los puertos de entrada/salida de la Raspberry Pi a la placa Pi2AX12. Este bus será el encargado de la comunicación que tendrá la Raspberry Pi desde y hacia los actuadores, utilizando como interfaz la placa Pi2AX12. Se conectaron 5 cables: Rx, Tx, GPIO18 (control de habilitación/deshabilitación del buffer tri estado), Vcc y GND según la el esquema de la figura 4.2; a la derecha se observa una imagen de la conexión física en el robot.

Conexión entre placa Pi2AX12 y Raspberry Pi 2

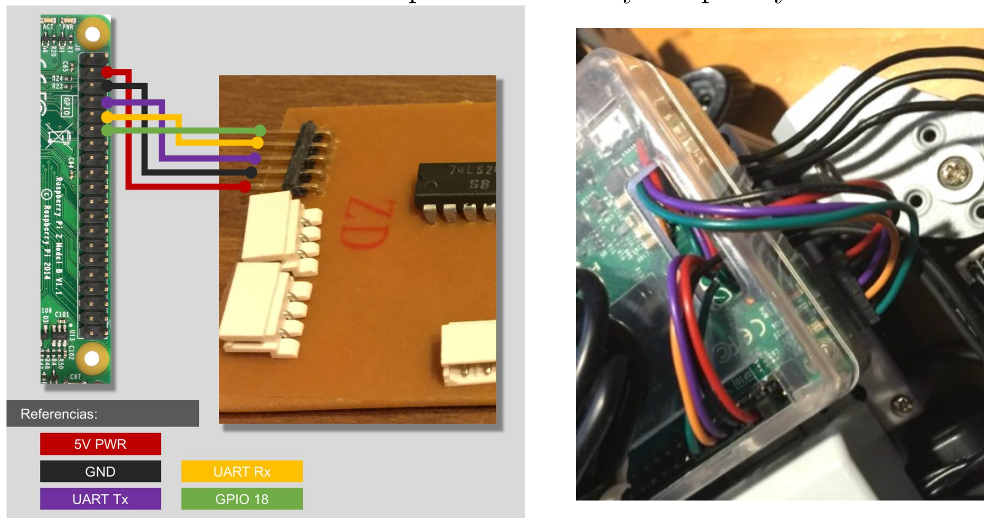

Figura 16: Esquema de conexión y conexión física

Finalmente se deben conectaron los cables de alimentación. Por un lado, el que conecta el puerto micro USB de la Raspberry Pi a la placa Pi2AX12 para proveer a la Raspberry de energía; por otro, el que conecta la placa

Pi2AX12 a la batería. El cable utilizado para conectar la placa Pi2AX12 con la Raspberry contiene un regulador de voltaje ya que la Raspberry Pi utiliza un voltaje distinto al que tiene la batería. En la siguiente sección se explican los detalles de dicho componente. Ambas conexiones pueden verse en las figuras 17 y 18.

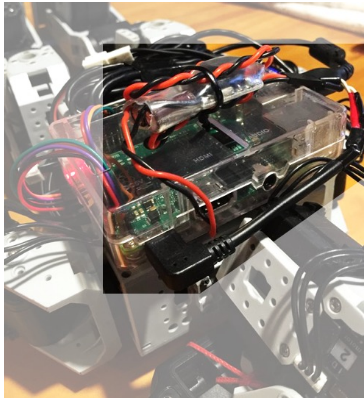

Figura 17: Conexión con Raspberry

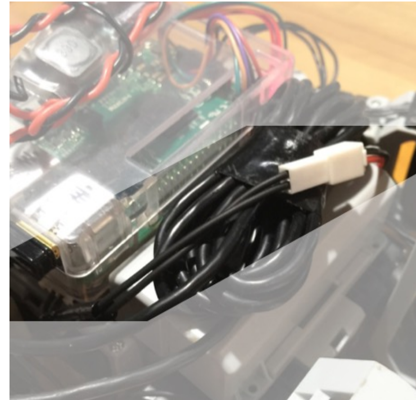

Figura 18: Conexión con batería

4.2.1. Regulador de voltaje

Debido a que las baterías Bioloid tienen un voltaje de 11.1v y la Raspberry precisa ser alimentada con 5v, es imprescindible el uso de un componente que "convierta" el voltaje de la batería al necesario por la placa. Este componente se llama regulador de voltaje o circuito eliminador de baterías (BEC en inglés) y tiene como objetivo eliminar la necesidad de usar distintas baterías para la placa y los motores. El modelo usado es el UBEC 8S-5A que se observa en la figura 19. Puede recibir voltajes de entrada desde 5V a 30V y tiene una salida de 5V y 5A continuos. Es un componente comúnmente usado en la construcción de helicópteros radio-controlados ya que su peso no supera los 18 gramos, por lo que su uso no introduce grandes cambios en la distribución del peso del robot.

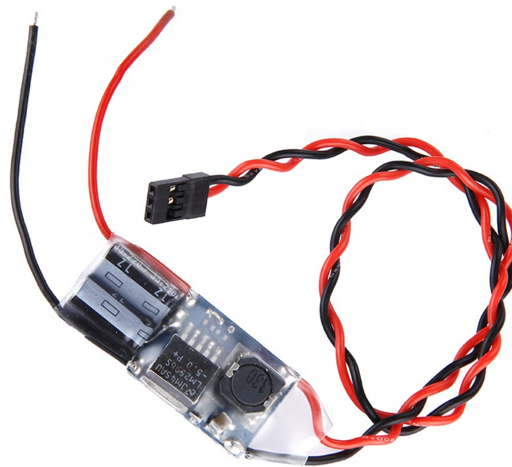

Figura 19: Regulador de voltaje UBEC 8S-5A

Este componente está conectado por un lado a uno de los conectores MOLEX de la placa Pi2AX12, del cual solo utiliza el pin de corriente y el pin de tierra, y por el otro lado está conectado a la entrada micro USB de la Raspberry Pi. Para dicha conexión se usaron los cables detallados en la sección 4.2.2.

4.2.2. Conectores

Para conectar el componente regulador de voltaje a la Raspberry Pi se usaron cables micro USB con terminación en forma de L para lograr un diseño más compacto de todo el conjunto. Debido a la forma en la que está puesta la Raspberry Pi en los robots, el conector micro USB queda posicionado hacia arriba. Es por este motivo que se optó por no usar un cable normal debido a que este sobresaldría demasiado y sería propenso a desconectarse, doblarse o romperse en caso de que el robot sufra una caída. Con el uso de los cables en forma de L se logra una mayor robustez, ya que el cable no sufre de movimientos debido a que queda junto a la carcasa de la Raspberry como se ve en la figura 20.

Figura 20: Cable USB en L conectado a Raspberry Pi

5. Descripción de la arquitectura de software

En esta sección se detallan los componentes de software involucrados en la solución del sistema implementado. A alto nivel, el comportamiento del sistema puede describirse como un ciclo que repite los siguientes pasos:

1. El robot percibe el entorno a través de la cámara web y los sensores WiFi.

2. La información obtenida es procesada y utilizada para evaluar un modelo de reglas de comportamiento.

3. Como resultado, se ejecuta una acción. Por ejemplo: patear, caminar, atajar, girar.

Para modelar dicho comportamiento se definió una arquitectura de software dividida en cuatro módulos: el módulo de toma de decisiones, el módulo de comunicación, el módulo de visión y el módulo de manejo de motores.

El diagrama de la figura 21 ilustra los módulos del sistema y sus dependencias funcionales. El módulo de toma de decisiones depende del módulo de comunicación y del módulo de visión para tomar la decisión de la acción a ejecutar. Además, también depende del módulo de manejo de motores para que el robot ejecute la acción elegida.

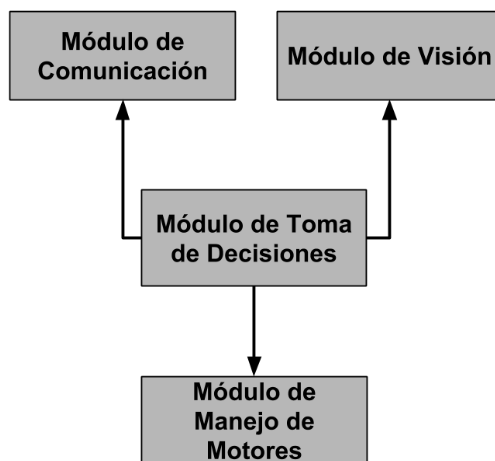

Figura 21: Diagrama de módulos del sistema.

Cada módulo cumple con una función específica:

- El módulo de visión es el encargado de obtener las imágenes de la cámara y realizar el procesamiento necesario para determinar si los distintos elementos del juego de fútbol se encuentran dentro del campo de visión del robot. Los elementos reconocidos por éste módulo son: la pelota, los arcos, los otros robots, tanto rivales como compañeros y las marcas que delimitan el medio de la cancha. Esta información es usada por el módulo de toma de decisiones. Parte de los componentes de este módulo fueron implementados por los proyectos *VisRob* [10] y *VisRob 2* [11] y luego modificados por el proyecto *FutBot* [13].

- El módulo de comunicación tiene la tarea de recibir los mensajes que son enviados por otros robots del equipo y guardarlos hasta que sean consumidos por el módulo de toma de decisiones. Asimismo, este módulo implementa las funcionalidades para el envío de mensajes hacia otros robots.

- El módulo de toma de decisiones se encarga de decidir qué acción debe ejecutar el robot en cada momento. Al comienzo del programa se carga un archivo que define ciertas reglas de comportamiento. Luego, en cada ciclo de ejecución estas reglas se evalúan para definir la próxima acción a ejecutar. El resultado de la evaluación de cada regla está fuertemente ligado al estado del entorno en ese momento, información que es proporcionada por los módulos de comunicación y visión.

- El módulo de manejo de motores obtiene una acción a ejecutar y es el responsable de comunicarse con cada uno de los motores de forma que el robot realice los movimientos que definen dicha acción. La base del código de este módulo es tomada del proyecto *Salimoo* [12].

En la figura 22 se muestra un diagrama detallado con todos los elementos que componen la arquitectura de software, separando los componentes en capas e indicando los proyectos de grado involucrados en su implementación. La lógica del módulo de toma de decisiones se descompuso en elementos más pequeños para detallar mejor su comportamiento. También se muestran las dependencias funcionales entre los distintos elementos.

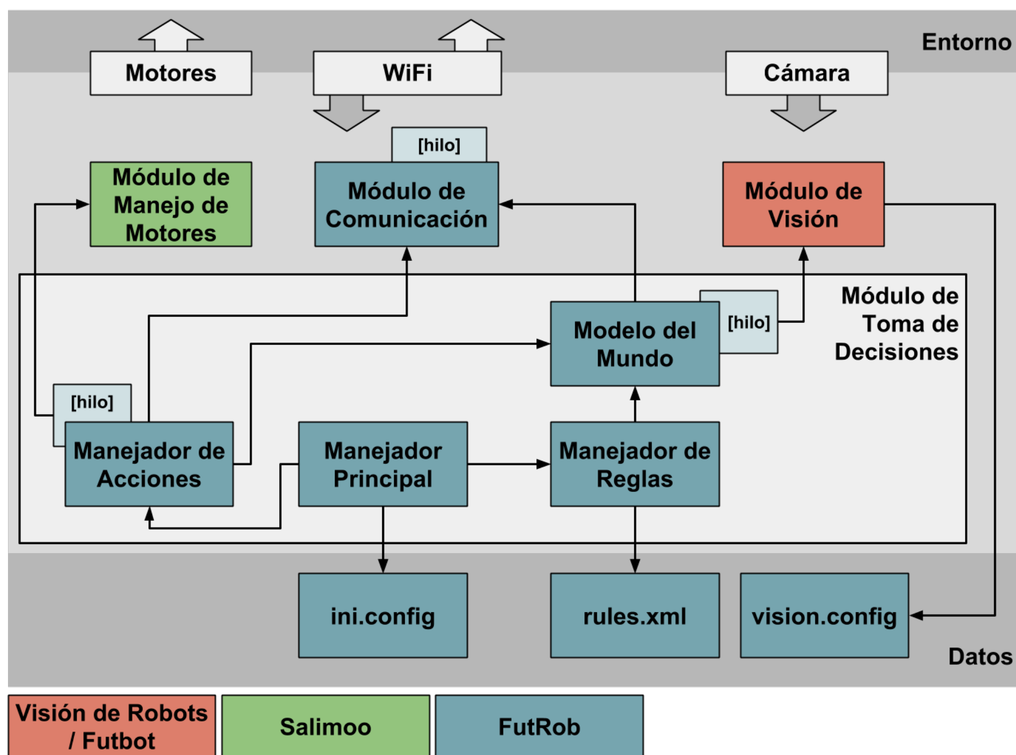

Figura 22: Diagrama detallado de componentes

Los componentes presentes en la capa de datos son archivos de configuración que se utilizan para parametrizar varios aspectos de la ejecución del sistema. Cada archivo se carga al momento de iniciar el sistema. Los archivos utilizados son:

- `ini.config`: contiene los parámetros de configuración generales. Es de tipo clave-valor.
- `rules.xml`: contiene la definición de las reglas. Se aborda detalladamente en la sección 5.1.1.
- `vision.config`: contiene la definición de los entornos de colores para cada elemento reconocido por el módulo de visión, además de otras configuraciones relacionadas con la visión y las imágenes. Se aborda detalladamente en la sección 5.3.

Las configuraciones incluidas en el archivo `ini.config` se mencionan a lo largo de diferentes secciones del documento. Abarcan desde la definición

de la identidad del robot, hasta el comportamiento de componentes como la visión y el registro de actividad (log). En la figura 23 se incluye un archivo de ejemplo. Las líneas que comienzan con el carácter “#” no son consideradas por el intérprete.

```
1  #----------------------------------
2  # Robot initial configuration file |
3  #----------------------------------
4  [robot_profile]
5  id=1
6  #roleEnum: [1=goalkeeper, 2=field_player]
7  role=1
8  #Enable/disable logging
9  logEnabled=1
10 logfile=futrob.log
11 logAction=1
12 logMotors=0
13 logVision=1
14 logComm=1
15 logRules=0
16 #Vision stub enable/disable
17 visionStubEnabled=0
18 visionStubSrc=objSampleData.input
19 #Rules source file. Mandatory.
20 rules=rules.xml
21 #Automatic lookAtBall action
22 autoLookAtBall=0
```

Figura 23: Ejemplo de archivo de configuración “ini.config”.

Varios de los componentes interactúan con el entorno del robot. El módulo de manejo de motores, por medio del movimiento de los motores, modifica el entorno cada vez que el robot se mueve para realizar una acción. El módulo de comunicación interactúa con el entrono de forma bidireccional ya que puede enviar mensajes al entorno que son escuchados por otros robots y recibir del mismo otros mensajes de los demás robots. Dichos mensajes se envían a través de la red WiFi. El módulo de visión recibe del entorno la información que obtiene de la cámara.

5.1. Módulo de toma de decisiones

La solución se diseñó de forma que el comportamiento de los robots sea fácilmente ajustable por el usuario del programa sin tener que modificar el código. Si bien los posibles movimientos de los robots están predefinidos y codificados, se pueden establecer diferentes estrategias de juego utilizando los predicados disponibles para crear reglas. Estas reglas definirán las acciones a ejecutar en función del estado en que se encuentre el entorno.

Los tres elementos principales involucrados en la toma de decisiones son: las reglas, los predicados y las acciones. Las acciones y los predicados están predefinidos e implementados en el programa, pero las reglas pueden ser modificadas y son un elemento externo al programa.

Los predicados modelan el estado del entorno en un momento dado y son implementados en el modelo del mundo.

Cada acción modela un determinado movimiento del robot y su implementación se encuentra dentro del manejador de acciones.

Las reglas se construyen utilizando los predicados y operadores lógicos para indicar la acción a ejecutar en cada momento. Están definidas en un archivo XML y son cargadas en tiempo de ejecución por el módulo manejador de reglas que se encarga de evaluarlas.

El comportamiento de los robots queda definido por las reglas, los predicados utilizados, y las acciones asociadas a las mismas.

5.1.1. Manejador de reglas

Las reglas están definidas en un archivo de texto en formato XML que es cargado por el manejador de reglas al momento de iniciar la ejecución del programa. Por defecto el programa intenta cargar `rules.xml`. Este nombre de archivo está definido en el archivo de configuración inicial `ini.config` bajo la clave `rules`. Esto facilita el intercambio entre diferentes conjuntos de reglas, cambiando únicamente el valor de la clave. Tener el conjunto de reglas definidas en un archivo encapsula en forma independiente una parte importante del comportamiento del robot, permitiendo variarlo sin necesidad de recompilar el código. Esto ofrece al usuario una gran flexibilidad a la hora de definir distintas estrategias de juego.

La prioridad de las reglas está dada por el orden en el que están definidas. El programa evalúa las reglas en cascada; en caso que una regla sea falsa, se evalúa la siguiente hasta encontrar la primera verdadera. Cuando se evalúa positivamente una regla, se ejecuta la acción asociada para luego volver a comenzar el proceso de evaluación de reglas comenzando desde la primera regla. Una ventaja de este mecanismo es que en la gran mayoría de los casos no será necesario evaluar todas las reglas para definir la acción a ejecutar, logrando de esta forma disminuir la utilización de recursos.

Las reglas que definen el comportamiento del robot se basan en los predicados que ofrece el sistema. Éstos corresponden a diferentes estados en que puede encontrarse el entorno del robot en un momento dado; por ejemplo, si el robot ve la pelota, o si el robot se encuentra frente al arco rival. Cada predicado toma un valor *booleano*. Luego, las reglas son operaciones lógicas sobre estos predicados. Cada regla puede contener una cantidad arbitraria

de operaciones.

Las operaciones implementadas son:

- `not`: se utiliza solamente con un argumento; evalúa verdadero cuando el argumento evalúa falso.

- `and`: se utiliza con dos o más argumentos; evalúa verdadero cuando todos sus argumentos son verdaderos.

- `or`: se utiliza con dos o más argumentos; evalúa verdadero cuando alguno de sus argumentos es verdadero.

Los argumentos de las operaciones pueden ser tanto predicados como otras operaciones lógicas. A su vez, también es válida una regla que contenga únicamente un predicado, sin ninguna operación. Esto permite generar desde reglas sencillas con pocas operaciones hasta reglas muy complejas con una gran cantidad de operaciones, ofreciendo la posibilidad de adoptar distintos enfoques a la hora de definir el comportamiento por parte de los usuarios de la plataforma.

Un posible enfoque es la definición de muchas reglas sencillas con pocas operaciones que en su conjunto logren definir una estrategia de juego más compleja. Este enfoque tiene como ventaja que las reglas resultan más fáciles de entender debido a su simplicidad pero tiene la desventaja de que el numero de reglas puede ser demasiado grande haciendo que el trabajo de ordenarlas por prioridad resulte demasiado complejo.

Otro enfoque es definir pocas reglas con un alto grado de complejidad, donde sean usadas varias operaciones por cada regla. Esto permite generar estrategias complejas que tomen en cuenta varias variables del entorno para cada decisión, pero tiene como desventaja de que las reglas resultan más difíciles de entender y por lo tanto de probar y modificar para ajustar a un comportamiento deseado.

También existe la posibilidad combinar estos enfoques y generar un conjunto de reglas que tenga tanto reglas sencillas como algunas más complejas para cuando la estrategia lo amerita. La cantidad de reglas no está limitada, por lo que el usuario puede definir tantas reglas como quiera.

El archivo XML que define las reglas tiene que seguir la estructura que se detalla en el fragmento 1.

Fragmento 1: Ejemplo de archivo de definición de reglas

```
1 <?xml version="1.0" encoding="US-ASCII" ?>
2 <RuleSet>
3   <Rule name="regla1" desc="descripcion regla1">
```

```
4      <def>
5        <not>
6          <value term="predicado1"/>
7        </not>
8      </def>
9      <action name="accion1" var1="A" var2="B"/>
10   </Rule>
11
12
13   <Rule name="reglaN" desc="descripcion reglaN">
14     <def>
15       <and>
16         <value term="predicado1"/>
17         <value term="predicado2"/>
18         <not>
19           <value term="predicado3">
20         </not>
21       </and>
22     </def>
23     <action name="accionN" var1="valorN"/>
24   </Rule>
25 </RuleSet>
```

Los *tags* utilizados en el archivo de definición de reglas son:

- `<RuleSet>`: Contiene todas las reglas definidas en el archivo. Es el tag raíz del documento.

- `<Rule>`: Define una regla. Los tags <Rule>son los únicos hijos válidos del tag <RuleSet>. Tienen dos atributos opcionales: *name* y *desc*, que corresponden al nombre y a la descripción de la regla respectivamente. Estos datos no son usados en la ejecución del programa y se incluyen únicamente para facilitar la compresión de las reglas al escribirlas o leerlas y para depurar el programa ya que el nombre de las acciones ejecutadas queda guardado en un log.

- `<def>`: Contiene la definición de la regla. Solamente puede contener un hijo directo, ya sea una operación lógica (<not>, <and>, <or>) o un predicado (<value>)

- `<action>`: Define la acción a ejecutar en caso que la regla se cumpla. Junto con <def>son los únicos hijos válidos del tag <Rule>. Tiene

un atributo *name* que indica el nombre de la acción. Los nombres de las acciones están predefinidos y tienen que ser escritos igual a como se definen en la sección 5.1.2 para que sean reconocidos correctamente por el programa. En caso que la acción reciba parámetros se colocará un atributo adicional por cada parámetro, donde el nombre del atributo debe ser el nombre del parámetro definido por la acción y el valor del atributo es el valor que se quiere que tome ese parámetro.

- `<value>`: Representa un predicado. El nombre del predicado se indica en el atributo *term* y tiene que ser escrito igual a como está definido. La lista de predicados se encuentra en la sección 5.1.3.
- `<not>`: Representa el operador lógico *not*. Solo puede contener un hijo directo.
- `<and>` : Representa el operador lógico *and*. Puede tener varios hijos.
- `<or>` : Representa el operador lógico *or*. Puede tener varios hijos.

5.1.2. Manejador de acciones

Las acciones están relacionadas al movimiento del robot. Cada acción equivale a una serie de movimientos de los motores que en su conjunto modelan cierto comportamiento; por ejemplo: caminar, patear, atajar. Esto quiere decir que la acción es el nivel de granularidad más fino que se tiene para controlar el movimiento del robot por parte del usuario de la plataforma.

La implementación de cada acción está definida y codificada en el programa en capas de más bajo nivel, como se detalla en la sección 5.4. Cada acción tiene un nombre que la identifica y es ejecutada cuando el manejador de reglas evalúa positivamente una regla a la que está asociada. Más adelante en esta misma sección se listan las acciones disponibles en el sistema.

Para ofrecer mayor versatilidad, algunas acciones definen parámetros que varían el comportamiento de la misma. La acción de caminar puede recibir como parámetro la distancia en centímetros que se pretende que el robot camine. La acción de galopas laterales recibe un parámetro análogo, y las acciones de giro reciben como parámetro el valor del ángulo a girar.

Se distinguen dos tipos de acciones: las acciones básicas y las acciones compuestas. Las acciones básicas son aquellas que no se pueden descomponer en acciones más simples. Cada una de ellas, dependiendo de sus parámetros, queda definida como una serie de movimientos de los motores del robot. Si bien estas acciones están disponibles para su utilización directamente en la construcción de las reglas de comportamiento del sistema, este uso podría

resultar complejo y dar lugar a conjuntos demasiado específicos y extensos de reglas para definir el comportamiento.

Esto motiva la existencia de las acciones compuestas, que ofrecen un nivel de abstracción más elevado en comparación con las acciones básicas. Un ejemplo de acción compuesta es *"findBall"*, que define los movimientos del robot necesarios para lograr contacto visual con la pelota. Para completar esta acción, se utilizan combinaciones de acciones básicas como el movimiento de la cabeza del robot e incluso girar en el lugar para lograr ver en todas las direcciones. Definir el comportamiento de buscar la pelota a nivel de reglas que usan predicados y acciones básicas puede resultar difícil teniendo en cuenta que no se puede acceder a información específica como la posición exacta de la pelota desde el modelado de las reglas.

Las acciones compuestas modelan una tarea del robot asociada al juego de un partido de fútbol. Utilizan internamente las acciones básicas para ser implementadas y además acceden a las variables del entorno. La acción de posicionarse frente a la pelota para patear al arco puede requerir que el robot gire en el lugar y que se desplace lateralmente para quedar enfrentado a la pelota con la pierna correcta, todo esto mientras se ubica de frente al arco rival. Este tipo de comportamiento es el que se encapsula en las acciones compuestas.

A continuación se listan las acciones básicas con sus respectivos parámetros. En el nivel principal se lista el nombre de la acción, mientras que como subíndice se listan los parámetros en caso que corresponda. Tanto el nombre de la acción como el nombre de los parámetros se deben escribir en el archivo de reglas tal como están aquí definidos:

- `none`: El robot no realiza ningún movimiento. Es la acción de mantenerse en la posición actual. Útil especialmente para el robot arquero, en caso de detectar que la pelota viene al arco y exactamente en su dirección, o cuando está en una espera pasiva porque la pelota está lejos del arco. A nivel interno de la aplicación tiene sentido modelar esta acción, aunque rara vez sería útil para elaborar una regla de comportamiento.

- `initialPosition`: El robot se coloca en su posición inicial de reposo. Es la posición de equilibrio estático.

- `walk`: El robot camina hacia adelante. El tamaño mínimo del paso es una constante de configuración del sistema `MINI_STEP_DISTANCE` , con valor inicial 2 cm.

    - *distance*: Distancia en centímetros. El valor por defecto es la distancia equivalente a un paso. (Opcional).

- `walkRight`: El robot se desplaza lateralmente hacia la derecha.
  - *distance*: Distancia en centímetros. El valor por defecto es la distancia equivalente a un paso lateral. (Opcional).
- `walkLeft`: El robot se desplaza lateralmente hacia la izquierda.
  - *distance*: Distancia en centímetros. El valor por defecto es la distancia equivalente a un paso lateral. (Opcional).
- `turn`: El robot gira en el lugar.
  - *angle*: Indica el ángulo de giro del robot en grados. Para valores positivos el robot se mueve hacia la derecha y para valores negativos el robot se mueve a la izquierda.
- `turnHeadVertically`: El robot mueve la cabeza verticalmente.
  - *angle*: Indica el ángulo de **destino** de la cabeza en grados. El valor 0 indica que la cabeza apunta hacia el frente, paralelo al suelo. Para valores negativos la cabeza se mueve hacia abajo. El valor mínimo es -60 y el valor máximo es 0, ya que no tiene sentido el movimiento de la cabeza hacia arriba.
- `turnHeadHorizontally`: El robot mueve la cabeza horizontalmente.
  - *angle*: Indica el ángulo de **destino** de la cabeza en grados. El valor 0 indica que la cabeza apunta hacia el frente. Para valores positivos la cabeza se mueve hacia la derecha y para valores negativos la cabeza se mueve hacia la izquierda. El valor mínimo es -60 y el valor máximo es 60.
- `kickRight`: El robot patea con la pierna derecha.
- `kickLeft`: El robot patea con la pierna izquierda.
- `frontGetUp`: El robot se pone de pie si está caído hacia adelante.
- `backGetUp`: El robot se pone de pie si está caído hacia atrás.
- `saveRight`: El robot hace el movimiento de atajar hacia la derecha.
- `saveLeft`: El robot hace el movimiento de atajar hacia la izquierda.

Las acciones compuestas, además de estar implementadas utilizando las acciones básicas, acceden a variables del modelo del mundo en su implementación de forma de corregir o modificar su comportamiento en caso de detectar cambios en el entorno.

Considerando la forma humanoide del robot, los movimientos que éste implementa guardan gran similitud con los movimientos de los humanos. Haciendo un paralelismo, los humanos nos movemos con lo que denominamos *equilibrio dinámico*; implica que estamos rompiendo la posición de equilibrio y recuperándola constantemente. Por ejemplo, para caminar, despegamos un pie del suelo e inclinamos el tronco hacia adelante, provocando una caída hacia adelante que luego compensamos apoyando el pie opuesto sobre el suelo.

Este mismo comportamiento es replicado en los movimientos del robot, que se encuentra continuamente rompiendo la posición de equilibrio para lograr un movimiento, y luego recuperando el equilibrio nuevamente. Esto explica por qué los movimientos, y consecuentemente las acciones básicas y complejas, no pueden ser detenidas en forma abrupta, ya que se debe garantizar el nuevo estado de equilibrio del robot para evitar una caída. Para lograr esto, el manejador de acciones implementa un mecanismo que manipula en forma atómica cada serie de movimientos entre 2 posiciones de equilibrio. Cuando se desea interrumpir una acción, el sistema envía una señal hacia el módulo manejador de motores (que se describe en la sección 5.4) indicando que, una vez alcanzada la siguiente posición de equilibrio, se detenga.

Esta capacidad de interrumpir una acción previo a completarse totalmente, dota al sistema de la capacidad de modelar cambios en la toma de decisiones en medio de la ejecución de cualquier acción.

Por ejemplo, si se decide ejecutar la acción básica de caminar 50 centímetros hacia la posición de la pelota, pero en la mitad del recorrido la pelota cambia de posición, probablemente no sea deseable continuar con la caminata. Las reglas deberían reflejar esta situación y definir la ejecución de una nueva acción. En este caso la acción de caminar puede detenerse cuando el robot llega a un punto de equilibrio, que sería al finalizar un paso. Lo mismo ocurre con la acción de girar en el lugar y caminar lateralmente.

Para el caso de la caminata, si bien el tamaño mínimo de paso es $2cm$ (`MINI_STEP_DISTANCE = 2`), el sistema cuenta con la implementación de una caminata de paso largo, cuya distancia de paso está configurada en el sistema en la constante `STEP_DISTANCE`, con valor inicial de $12cm$. Por lo tanto, los trayectos de caminata se traducen en una combinación lineal de una determinada cantidad de pasos largos seguida de una cantidad de pasos pequeños. Por ejemplo, para caminar $30cm$, el robot caminará 2 pasos grandes, y lue-

go 3 pasos pequeños. La caminata se implementó de forma que la distancia que recorre el robot es siempre menor o igual a la distancia indicada por el parámetro de distancia. Por ejemplo, si se indica al robot que camine $7cm$, caminará $6cm$, puesto que el parámetro se interpreta como cota superior de la distancia a recorrer.

Las acciones compuestas implementadas son las siguientes:

- Acciones de búsqueda: El sistema ofrece la posibilidad de buscar cualquiera de los objetos significativos para el juego. El robot busca el objeto; si no está en su campo de visión, mueve la cabeza hacia ambos lados y hacia abajo para intentar localizarla. Las acciones de búsqueda para cada objeto son las siguientes:
  - `findball`: busca la pelota.
  - `findpartner`: busca un robot compañero de equipo.
  - `findopponent`: busca un robot oponente.
  - `findrightlm`: busca la marca de campo derecha.
  - `findleftlm`: busca la marca de campo izquierda.
  - `findmygoal`: busca el arco propio.
  - `findoppgoal`: busca el arco rival.
  - `findmylp`: busca poste del arco propio.
  - `findopplp`: busca poste del arco rival.
- `lookAtBall`: Si el robot ve la pelota, mueve la cabeza para dejar la pelota en el centro de su campo de visión de forma de no perderla de vista en caso que ésta cambie de posición. Esta acción no resulta de mucha utilidad si es utilizada por una regla, ya que las condiciones que tienen que cumplirse para que sea útil realizarla no están disponibles como predicados para ser usados por las reglas. Sin embargo, en la configuración general del robot puede especificarse si se quiere que el mismo ejecute la acción automáticamente cuando sea más conveniente de forma de mantener la pelota dentro del campo de visión de la cámara.
- `followBall`: Si el robot ve la pelota, camina hacia la misma corrigiendo su dirección en caso de desviarse.
- `passBall`: Si el robot ve la pelota a una distancia cercana, busca un compañero, se posiciona de frente al mismo y la patea.

- `kickBall`: Si el robot ve la pelota a una distancia cercana, busca el arco rival, se posiciona de frente al mismo y la patea.
- `clearBall`: Si el robot ve la pelota a una distancia cercana, la patea para despejarla.
- `save`: Si el robot ve la pelota acercándose, espera a que este a una distancia cercana y realiza la maniobra de atajar hacia el lado por el que viene la pelota.
- `relocate`: El robot arquero se posiciona en el centro del arco. En caso de estar desviado del centro del arco y tener conciencia de ello, el robot arquero se mueve con pasos laterales hasta recuperar la posición central.

Para que la ejecución de las acciones no provoque interrupciones en el sistema mientras el robot termina de ejecutarlas, éstas se ejecutan en un hilo independiente, dedicado exclusivamente a este fin. De esta forma el sistema es capaz de seguir evaluando las reglas y percibir el entorno mientras el robot se mueve.

Cuando el manejador de acciones recibe una acción a ejecutar, primero comprueba que no sea la acción que se encuentra ejecutando en ese momento. En caso de ser la misma acción, la descarta. Si se trata de una nueva acción, el manejador de acciones le indica al hilo que hay una nueva acción para ejecutar y se la envía. Cuando el hilo recibe la notificación de que hay una nueva acción, detiene o finaliza la que se encuentra ejecutando en ese momento para luego comenzar a ejecutar la nueva.

El hilo utiliza las operaciones disponibles en el módulo de manejo de motores para controlar los motores del robot.

5.1.3. Modelo del mundo

El modelo del mundo es el encargado de implementar los predicados que el sistema ofrece para la construcción de las reglas de comportamiento. Cada uno modela un determinado aspecto del entorno de los robots y su valor varía en cada momento. Pueden hacer referencia a uno o más datos obtenidos por los sensores en un momento dado. Estos datos se actualizan constantemente a medida que los sensores realizan lecturas sobre el entorno. Los posibles valores que pueden tomar son *verdadero* o *falso*, por lo que pueden ser utilizados directamente en las operaciones lógicas que componen las reglas.

El sistema cuenta con dos formas de obtener información del entorno: la visión y la comunicación con otros robots. La información de la visión es obtenida mediante el módulo de visión desde la cámara web colocada en el

Fragmento 2: WorldModel.h - Variables de estado

```
1 private:
2     int myID;
3     int myRole;
```

robot a la altura de la cabeza. La información de la comunicación se recibe mediante el módulo de comunicación, que la toma desde de la red WiFi.

Además, el modelo del mundo es el responsable de mantener una serie de valores que determinan y controlan el comportamiento de la lógica del robot. Llamamos a estos valores las “variables de estado” del robot, y se incluyen en el fragmento 2. Esta implementación permite que la lógica ejecutada sea idéntica para cualquiera de los robots miembros del equipo, facilitando no solo la mantenibilidad del sistema sino que haciendo posible un comportamiento dinámico en relación a los posibles roles. Es decir, es la propia lógica la encargada de determinar el comportamiento y el conjunto de acciones a ejecutar dependiendo de la configuración de las variables de estado del robot. Claros ejemplos son las acciones de atajadas, que son ejecutadas únicamente si el robot tiene como rol arquero. Un ejemplo de comportamiento que aplica únicamente para los jugadores de campo es el envío de mensajes al arquero indicando su posición relativa respecto del centro del arco.

Para tener siempre disponibles los predicados, que pueden ser consultados de forma inmediata para la evaluación de las reglas, el modelo del mundo ejecuta un hilo que está constantemente actualizando su información sobre los objetos detectados por la visión. La forma de interactuar con el hilo es mediante memoria compartida. El hilo actualiza la información de este espacio compartido en cada iteración donde realiza un llamado al módulo de visión preguntando por la posición de los objetos del entorno. Cuando se realiza una llamada a un predicado del modelo del mundo, se devuelve el valor almacenado en memoria. De esta forma no es necesario volver a evaluar los predicados en cada consulta que realiza el manejador de reglas ni recabar información desde los sensores sino que se accede al dato disponible en la memoria del sistema, ahorrándose tiempo de procesamiento.

El hilo también es responsable de realizar un procesamiento adicional sobre la información obtenida del módulo de visión con el objetivo de mejorar su confiabilidad. En ocasiones puede ocurrir que, cuando un objeto se encuentra dentro del campo de visión del robot, el módulo de visión devuelva aisladamente respuestas puntuales que no constaten esta presencia del objeto. Cuando los datos tienen estas características, es alta la probabilidad de que haya ocurrido un error puntual en el módulo de visión y que el objeto

haya estado siempre visible para el robot. Por lo tanto, sería deseable que el sistema contemple esta casuística y responda consecuentemente, mitigando este tipo de situaciones.

Para suavizar esta volatilidad en la información, el procesamiento que realiza el hilo con los datos obtenidos obliga a que, para cambiar el estado de visible a no visible (o viceversa) de cualquier objeto, sea necesaria una cantidad mínima de iteraciones donde el módulo de visión responda la misma información.

En lo que hace al juego de fútbol, se considera la pelota como el objeto más importante del juego ya que para una amplia mayoría de las acciones es necesario interactuar con ella. Atendiendo esta condición, el módulo de visión contempla en forma particular el objeto pelota. La información obtenida desde la cámara web recibe un tratamiento especial para asegurar que los datos que la representan sean lo más precisos posible.

Cada vez que se consulta al módulo de visión sobre los objetos captados por la cámara, la posición de la pelota se guarda en una estructura que contiene los datos históricos de sus últimas posiciones y una marca que indica el momento en que fueron registradas. Esta información es utilizada para realizar una estimación más precisa de su posición que solamente el último dato de provisto por la visión. Como sabemos que la pelota al rodar sobre el suelo se mueve en forma rectilínea mientras no sufra un cambio en su dirección producto del choque con otro objeto, las diferentes posiciones devueltas por el módulo de visión a lo largo del tiempo deberían ajustarse a conjuntos de líneas rectas.

Al momento de consultar la posición de la pelota, si se tiene una determinada cantidad de muestras en el registro histórico de posiciones, se puede contar con una mejor aproximación de su posición. Para lograrlo, se utilizan las muestras históricas para determinar la mejor aproximación de recta que pasa por dichas posiciones usando la técnica de análisis de mínimos cuadrados. Luego se devuelve una posición sobre la recta calculada. Cada vez que se obtiene una nueva posición de la pelota desde el módulo de visión, este dato se utiliza para realizar el cálculo de la velocidad y la aceleración valiéndose de las posiciones anteriores. De este modo se puede tener, para cada posición, una velocidad y aceleración instantánea en ese punto. Por lo tanto, el módulo de visión aporta los datos en forma discreta, a una frecuencia de 5 veces por segundo aproximadamente, para alimentar un modelo continuo de la trayectoria de la pelota. Esto acorta la brecha entre la capacidad de procesamiento de la visión y la alta frecuencia de utilización de los datos de modelo del mundo por parte de la lógica de inteligencia artificial del robot. La posición de la pelota es conocida en todo momento, calculada como una estimación de su posición actual ajustada por el modelo de trayectoria y considerando

los datos de velocidad, aceleración, y últimas posiciones conocidas.

La información de las posiciones que provee el módulo de visión corresponde al ángulo en radianes (a) y a la distancia en centímetros (d) del robot a cada objeto. Como realizar los cálculos necesarios para las estimaciones de posición resulta muy complejo usando los datos de esa forma, se transforman a un formato más sencillo de manipular. Para realizar los cálculos se posiciona al robot en la intersección entre dos ejes: X e Y, y se calculan las distancias de los objetos a ambos ejes (pX y pY). En la figura 24 se muestra un diagrama del robot, la información obtenida del módulo de visión (a y d) y las distancias calculadas (pX y pY).

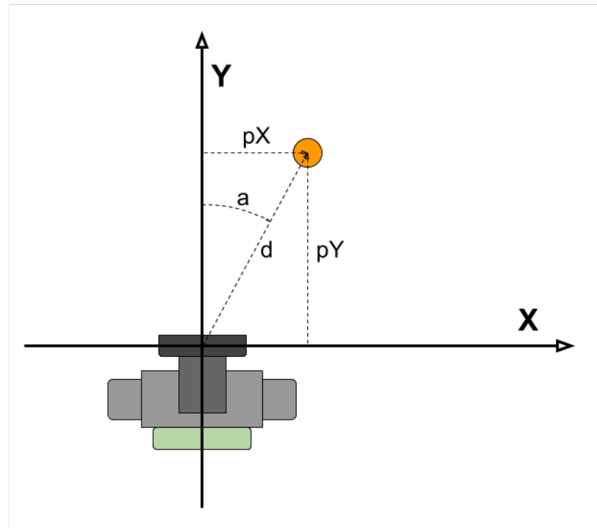

Figura 24: Diagrama del robot y las distancias a la pelota.

Para calcular los datos de pX y pY se usan las siguientes fórmulas:

$$pX = d * sin(a * PI/180) \tag{1}$$

$$pY = d * cos(a * PI/180) \tag{2}$$

Cada vez que se obtiene un nuevo dato sobre la poción de la pelota se guarda en una lista junto con una marca de tiempo para luego poder hacer los cálculos de velocidad y aceleración.

A partir que se tienen dos muestras, el sistema comienza a realizar los cálculos correspondientes. Primero se calcula la mejor recta que pasa por todos los puntos usando la técnica de mínimos cuadrados.

Queremos encontrar los coeficientes m y n de la recta:

$$y = mx + n \tag{3}$$

que mejor se aproxima a la trayectoria que realiza la pelota al moverse. Contamos con los puntos (pX_1, pY_1),...,(pX_N, pY_N) correspondientes a la posición de la pelota desde el momento en que se comenzaron a obtener los datos (pX_1, pY_1), hasta el dato de la última posición obtenida (pX_N, pY_N).

Para calcular los coeficientes m y n definimos la función X^2 como:

$$X^2(m,n) = \sum_{i=1}^{N}(pY_i - m * pX_i - n)^2 \tag{4}$$

que define una medida de la desviación total de los datos obtenidos por la visión respecto a los datos calculados usando el modelo lineal $y = mx + n$.

Los m y n que minimizan X^2 cumplen:

$$\frac{\delta X^2}{\delta m} = 0 \qquad \frac{\delta X^2}{\delta n} = 0 \tag{5}$$

de lo que se deduce que:

$$m = \frac{N \sum pX_i pY_i - \sum pX_i \sum pY_i}{N \sum pX_i^2 - (\sum pX_i)^2} \tag{6}$$

y

$$n = \frac{N \sum pX_i^2 \sum pY_i - \sum pX_i \sum pX_i pY_i}{N \sum pX_i^2 - (\sum pX_i)^2} \tag{7}$$

Usando estas ecuaciones se calculan los coeficientes m y n.

Una vez que se tiene la recta, al obtener una nueva muestra, se calcula la distancia del nuevo punto a la recta. Si el nuevo punto se encuentra a una distancia mayor que un cierto umbral ϵ dado, entonces el punto se marca y no se agrega al cálculo de mínimos cuadrados.

Si se tienen dos puntos consecutivos marcados que se encuentran ambos en el mismo semiplano, entonces se asume que la pelota cambió la trayectoria. Cuando hay un cambio de trayectoria se borran todas las muestras que se tienen hasta el momento y se comienza a realizar los cálculos con las nuevas muestras, actualizando el modelo y obteniendo la nueva recta.

Mientras se asume que la pelota se está moviendo en línea recta, se calculan la velocidad y aceleración en cada punto. La velocidad se calcula a partir de que se tienen dos muestras y la aceleración a partir de las tres

muestras. Para el cálculo estimado de la velocidad de la muestra i se calculan sus componentes vX_i y vY_i usando los puntos (pX_i, pY_i) y (pX_{i-1}, pY_{i-1}) correspondientes a los tiempos t_i y t_{i-1} respectivamente.

Se tienen entonces, para la muestra i, las ecuaciones:

$$vX_i = \frac{pX_i - pX_{i-1}}{t_i - t_{i-1}} \tag{8}$$

$$vY_i = \frac{pY_i - pY_{i-1}}{t_i - t_{i-1}} \tag{9}$$

Para el cálculo de la aceleración de la muestra i, se utiliza la información de la velocidad (vX_i, vY_i) y (vX_{i-1}, vY_{i-1}) en los tiempos t_i y t_{i-1} respectivamente:

$$aX_i = \frac{vX_i - vX_{i-1}}{t_i - t_{i-1}} \tag{10}$$

$$aY_i = \frac{vY_i - vY_{i-1}}{t_i - t_{i-1}} \tag{11}$$

Una vez que se calcula tanto la velocidad como la aceleración en cada punto, se calcula el módulo de cada una. Como la aceleración debería ser constante durante todo el trayecto de la pelota, aceleración inducida principalmente por la fuerza de rozamiento entre la pelota y la superficie, se realiza un promedio de todos los datos de aceleración calculados con las muestras anteriores y se utiliza ese promedio para los cálculos posteriores. Si se usa el dato de la aceleración teniendo en cuenta únicamente el último cálculo realizado y el dato tiene un error significativo, entonces los cálculos que se realicen a partir del mismo no van a resultar tan precisos en comparación a los realizados con el promedio. Cuantas más muestras se tengan, más preciso será el dato de la aceleración promedio.

Con estos datos es posible estimar la posición de la pelota en un determinado momento en el futuro, así como también estimar el momento en que la pelota pasaría por un determinado punto en el plano de la cancha. Esto es útil, por ejemplo, para el arquero. Cuando el robot arquero detecta que la pelota sigue una trayectoria que se dirige al arco, puede saber en cuánto tiempo la pelota alcanzaría la línea del arco y en qué posición. De esta forma puede adelantarse a ese momento y posicionarse de forma tal que al realizar la maniobra de atajar, pueda detener la pelota y evitar el gol.

Además de la pelota, también es importante contar con información sobre los demás objetos involucrados en el juego. El modelo del mundo provee las datos sobre la posición y las distancias a: los arcos, las marcas que indican la

mitad de la cancha, y los otros robots tanto compañeros como rivales. Dicha información solamente está disponible en caso que los objetos se encuentren en el rango de visión del robot y sean detectados por la cámara.

A continuación se listan los predicados disponibles en el modelo del mundo:

- `ballInSight`: Indica si el robot está viendo la pelota.
- `ballIsMine`: Indica si el robot tiene la posesión de la pelota, que equivale a calcular si está a una distancia que lo habilite a patearla.
- `ballIsOurs`: Indica si el equipo tiene la posesión de la pelota, que equivale a calcular si el predicado *ballIsMine* es verdadero para algún miembro del equipo.
- `ballIsGettingCloser`: Indica si la pelota se está acercando hacia la posición del robot.
- Los predicados a continuación indican si el robot está viendo el objeto en cuestión.
    - `partnerInSight`: Jugador compañero.
    - `opponentInSight`: Jugador rival.
    - `rightMarkInSight`: Marca derecha de campo.
    - `rightMarkInSight`: Marca izquierda de campo.
    - `myGoalInSight`: Arco propio.
    - `myGoalInSight`: Arco rival.
    - `myPostInSight`: Poste propio.
    - `oppPostInSight`: Poste rival.
- Los predicados a continuación indican si el robot considera válida la ultima posición en la que vio el objeto en cuestión. Cuando el robot ve un objeto el predicado toma valor positivo hasta que el robot realice una acción que implique un movimiento de traslación o rotación.
    - `ballLastPosValid`: Pelota.
    - `partnerLastPosValid`: Jugador compañero.
    - `opponentLastPosValid`: Jugador rival.
    - `rightMarkLastPosValid`: Marca derecha de campo.

    - `rightMarkLastPosValid`: Marca izquierda de campo.
    - `myGoalLastPosValid`: Arco propio.
    - `myGoalLastPosValid`: Arco rival.
    - `myPostLastPosValid`: Poste propio.
    - `oppPostLastPosValid`: Poste rival.

- `partnerInRange`: Indica si un jugador compañero está al alcance para recibir un pase, que equivale a calcular si existe algún jugador del mismo equipo a una distancia menor a la distancia de pateo de pelota del robot.

- `opponentInRange`: Indica si un jugador rival está al alcance de tiro, que equivale a calcular si existe algún jugador del equipo rival a una distancia menor a la distancia de pateo de pelota del robot.

- `goalInRange`: Si se está viendo un arco, indica si el mismo está a una distancia menor a la distancia de pateo de pelota del robot.

- `partnerPass`: Indica si un miembro del mismo equipo realizó un pase recientemente.

- `gkNeedsRelocate`: Indica si el jugador arquero debe posicionarse nuevamente en el centro del arco (es decir, si se encuentra en una posición corrida del centro del arco).

- `gkNeedsStartSaving`: Indica si, dada la trayectoria, velocidad y aceleración de la pelota, el robot debe comenzar a moverse para lograr interceptarla con una atajada. Considera que la pelota esté moviéndose en dirección al arco, y considera la posición donde cruzará la línea de meta. Luego compara con el tiempo que tardaría el robot arquero en alcanzar esa posición. La atajada puede implicar además del movimiento de inclinación hacia los lados para atajar, movimiento de pasos laterales.

- Los predicados a continuación indican si se intentó buscar el objeto en cuestión en la posición actual del jugador. Luego que el jugador se mueve, se reinicializan estas variables de estado y se asume todo objeto como no buscado.

    - `ballSearched`: Pelota.
    - `partnerSearched`: Jugador compañero.
    - `opponentSearched`: Jugador rival.

- `rightMarkSearched`: Marca derecha de campo.
- `leftMarkSearched`: Marca izquierda de campo.
- `myGoalSearched`: Arco propio.
- `oppGoalSearched`: Arco rival.
- `myPostSearched`: Poste propio.
- `oppPostSearched`: Poste rival.

En la implementación del predicado `gkNeedsRelocate`, se consideran umbrales de distancia mínima y máxima en ambos ejes de coordenadas. Si un robot jugador observa que el arquero está desplazado una distancia muy pequeña del centro en cualquiera de los ejes, entonces directamente no lo comunica para evitar tráfico innecesario. Esto evita que el robot arquero reciba y procese mensajes para movimientos muy pequeños. Análogamente para el caso en que el robot jugador observa al robot arquero a una distancia muy amplia del centro del arco; típicamente mayor al ancho del arco en el eje de abscisas. En este caso, tampoco se comunica el mensaje. Dado que el robot arquero debería estar siempre sobre la línea de meta, es decir, entre los 2 palos, esto brinda una protección al sistema ante interpretaciones erróneas de la visión en casos que la distancia detectada es extremadamente amplia. Estos umbrales se definen en el archivo `Constants.h` bajo las siguientes constantes:

- `GK_X_DIFF_THRESHOLD_MIN`: umbral mínimo en eje de abcisas.
- `GK_X_DIFF_THRESHOLD_MAX`: umbral máximo en eje de abcisas.
- `GK_Y_DIFF_THRESHOLD_MIN`: umbral mínimo en eje de ordenadas.
- `GK_Y_DIFF_THRESHOLD_MAX`: umbral máximo en eje de ordenadas.

Para implementar el predicado `gkNeedsStartSaving` como se describe, el sistema se vale de la siguiente información preestablecida en el sistema, que se incluye en el archivo `Constants.h`, con valores experimentales de laboratorio para realizar los cálculos correspondientes:

- `GK_GOAL_WIDTH`: Medida en centímetros del ancho del arco, entre vertical y vertical.
- `GK_STANDING_STILL_COVERAGE`: Medida en centímetros del espacio que cubre el arquero atajando en posición de equilibrio; parado quieto.
- `SAVE_DISTANCE`: Medida en centímetros del máximo espacio cubierto por el arquero para atajar al realizar el movimiento de atajada lateral hacia la derecha o izquierda.

- `RIGHT_STEP_DISTANCE`: Medida en centímetros que se desplaza el robot al realizar un paso hacia la derecha.
- `LEFT_STEP_DISTANCE`: Análogo a anterior pero hacia la izquierda.
- `LATERAL_STEP_TIMING_MS`: Tiempo en milisegundos que tarda el robot en ejecutar el movimiento de paso lateral.
- `SAVE_TIMING_MS`: Tiempo en milisegundos que tarda el robot en alcanzar la posición de máxima amplitud al ejecutar el movimiento de atajada lateral (es decir, tiempo que tarda en llegar a estar agachado hacia un lado).
- `GK_SAVE_TIME_THRESHOLD_MS`: Tiempo en milisegundos utilizado como umbral de tolerancia en el cálculo. Si la diferencia de tiempo entre la llegada de la pelota y el movimiento del arquero está dentro del umbral, entonces se devuelve verdadero para intentar completar el movimiento.

La mayoría de los predicados dependen exclusivamente de la información proveniente de la visión del robot. Otro número importante de predicados se apoya en los mensajes intercambiados entre los robots del equipo. Estos predicados se implementaron modelando una estructura en memoria que guarda la información obtenida a partir de los datos de la comunicación. El módulo de comunicación, detallado en la sección 5.2, alimenta una cola de mensajes que son constantemente interpretados por la lógica del modelo del mundo y transformados en información útil para la toma de decisiones y cálculo de predicados. El tipo de datos a nivel de código lleva el nombre de *TeamMemberData* y tiene la estructura que se observa en el fragmento 3.

Fragmento 3: WorldModel.h - Estructura de datos de otros jugadores

```
1  struct TeamMemberData
2  {
3   int robotID;               // id del robot
4   int robotRole;             // rol del robot
5
6   bool ballInSight;          // indica si ve la pelota
7   bool ballIsMine;           // indica si tiene la pelota
8   int ballDistance;          // distancia a pelota en cm
9
10  bool partnerInSight;       // indica si ve un companiero
11  int partnerDistance;       // distancia a companiero
12
13  bool opponentInSight;      // indica si ve un robot rival
```

```
14    int opponentDistance;    // distancia a robot rival
15
16    bool myGInSight;         // indica si ve el arco propio
17    int myGDistance;         // distancia a arco propio
18
19    bool oppGInSight;        // indica si ve el arco rival
20    int oppGDistance;        // distancia al arco rival
21
22    int actionTaken;         //ultima accion ejecutada
23    int zone;                //zona del campo que esta
24
25    // indica si se realizo un pase
26    bool teamPass;
27    //timestamp desde el ultimo pase
28    struct timespec lastTeamPass;
29
30    //timestamp desde ultima actualizacion del jugador
31    struct timespec lastUpdate;
32 };
```

El rol del robot puede ser dinámico, es decir, un mismo robot podría adoptar diferentes roles a lo largo del desarrollo del juego. Bastaría con implementar la lógica que modele el cambio de rol y establezca el valor correspondiente para la variable de estado del robot. Luego de un cambio en esta variable, el robot puede continuar ejecutando la lógica de la solución habiendo adoptado el nuevo rol. Además, el conjunto de roles es extendible, habiéndose implementado en primera instancia los roles *arquero* y *jugador*, dado que los equipos estarán conformados por 3 robots: 1 arquero y 2 jugadores de campo.

El campo *lastUpdate* guarda una marca de tiempo que indica la última vez que se recibieron datos para actualizar la estructura del jugador. Luego, para el uso de esta información y el mantenimiento de estas estructuras, se mide el tiempo en milisegundos de antigüedad de la información y se compara contra un tiempo máximo de vida (constante en milisegundos `COMM_TTL` por *Communication Time To Live*). Si la información está dentro de su vida útil, es utilizada. Si no, la información se descarta. Este mecanismo no solamente evita la utilización de información obsoleta, sino que también contempla incluso situaciones en que un robot deja de participar o cambia su rol en el juego. Por ejemplo, situaciones como *"El robot id=1 ha abandonado el juego"* o *"El robot id=4 asume el rol de Arquero"*.

Algunos predicados que dependen, entre otros elementos, de los mensajes de los demás robots del equipo son: `ballIsOurs` que determina si la pelota

está en posesión del equipo, `partnerPass` que indica si se ejecutó una acción de pase entre compañeros de equipo, y `GKNeedsRelocate` que indica si el arquero debe reubicarse en el arco o no.

5.2. Módulo de comunicación

En la solución implementada los robots cooperan entre sí apoyados en la comunicación entre ellos a través del intercambio de mensajes. Cada robot es capaz de enviar y de recibir mensajes de los demás robots. Se desarrolló un módulo de comunicación dedicado a implementar las primitivas de mensajería a través de la red. Este módulo está encargado de recibir los mensajes de los demás módulos del sistema y transmitirlos por la red. Paralelamente, cuenta con un hilo que está constantemente escuchando por mensajes del resto de los robots.

Los mensajes son enviados usando el protocolo UDP sobre la red WiFi. Las placas Raspberry Pi no están equipadas con una interfaz de red inalámbrica, por lo que se tuvieron que utilizar adaptadores externos USB para proveer al sistema de acceso a la red WiFi.

La característica más importante que tienen que cumplir los adaptadores es ocupar el menor espacio posible, ya que los puertos USB de la Raspberry están muy cerca de los brazos del robot. Si se utilizan adaptadores muy grandes pueden interferir con el movimiento de los brazos, además de modificar el centro de masa del robot de manera considerable. Luego de analizar diferentes alternativas, se decidió utilizar los adaptadores "Nano USB EDIMAX N150"[3] por tener soporte nativo de drivers para las placas Raspberry Pi y ser relativamente chicos y livianos teniendo una velocidad de transmisión de datos suficientemente rápida.

Se eligió el uso del protocolo UDP porque en las reglas de la competencia RoboCup [23] se indica que el recurso de la red puede no estar siempre disponible y los robots deberían funcionar sin problemas en caso de no contar con conexión a la red. El protocolo UDP, al ser un protocolo de mejor esfuerzo, es el que mejor se adecua a este tipo de conexiones donde no se puede garantizar que no existan interrupciones. Además, ya que el entorno de los robots cambia constantemente, no es deseable que existan retransmisiones ya que la información del mensaje puede pasar a ser obsoleta en caso que el mensaje demore en ser enviado satisfactoriamente.

A nivel de la comunicación de red, todos los mensajes son enviados a la dirección IP de *broadcast* 255.255.255.255 por lo que son recibidos por todos los robots. A su vez, los mensajes pueden ser de tipo *broadcast* o con un

[3] www.edimax.com/en/produce_detail.php?pd_id=347

destinatario específico. Está definido dentro del protocolo de comunicación implementado, utilizando un campo *idDestination* si el mensaje es para un destinatario específico o es para todos los robots. Luego, dentro del procesamiento del mensaje al momento de su recepción, se determina si el mensaje tiene sentido para el robot que lo recibió o no.

En los mensajes tipo *broadcast* un robot envía un mensaje y todos los demás robots lo reciben y lo procesan. Este tipo de mensajes se utilizan cuando un robot quiere comunicar cierta información que puede ser útil para cualquier robot del equipo, por ejemplo, que se tiene la posesión de la pelota. En el otro tipo de mensajes un robot le envía un mensaje a otro robot de forma especifica. El mensaje es recibido por todos los robots pero es descartado por los robots a los que no está dirigido. Un ejemplo del uso de este tipo de mensajes es cuando un robot le comunica al robot arquero su posición en relación al arco.

Los mensajes tienen un formato fijo, que consta de cuatro campos de tipo numérico como se muestra en el fragmento 4.

Fragmento 4: Estructura de mensaje

```
1 struct Message
2 {
3     int idSource;        //id robot que envia
4     int idDestination;   //id robot destinatario
5     int param;           //clave de contenido de mensaje
6     int value;           //valor del mensaje
7 };
```

El hecho de que todos los mensajes tengan el mismo formato y un tamaño fijo simplifica la implementación: siempre se envían y se reciben cuatro números enteros.

Cuando el robot inicia la ejecución del sistema, toma la configuración de su rol y su identificador desde el archivo de configuración inicial y envía un mensaje al resto de los robots comunicando ambos datos. De esta forma, los robots del equipo se percatan de la existencia del nuevo robot.

En el campo *idSource* cada robot envía su propio identificador. Este identificador es usado por los robots que reciben el mensaje para asociar la información del mismo con el robot que lo envió. En el campo *idDestination* se envía el identificador del destinatario del mensaje. En caso que sea un mensaje de tipo broadcast se utiliza un valor especifico que indica que el mensaje está dirigido a todos los robots. En el campo *param* se envía una clave que describe el contenido del mensaje. Dicha clave puede representar, por ejemplo, la acción elegida para ejecutar, la tenencia de la pelota, entre

otros. Finalmente en el campo *value* se envía el valor asociado a dicha clave. Por ejemplo, para el caso de la clave que representa la acción a realizar los posibles valores son las distintas acciones que existen. En el caso que la clave represente la tenencia de la pelota los posibles valores son verdadero o falso, de manera de poder indicar al resto del equipo cuando se gana posesión de la pelota y cuando se pierde la misma.

El módulo de comunicación tiene un hilo que está constantemente escuchando por nuevos mensajes provenientes de la red. Una vez que son recibidos, realiza el chequeo de destinatario para determinar si el mensaje debe guardarse o descartarse. En caso que el mensaje decida guardarse, el mismo se almacena en una estructura de lista esperando para ser procesado.

Cuando el módulo de toma decisiones necesita evaluar una regla que tiene predicados que involucran a otros jugadores del equipo, le pide al módulo de comunicación los mensajes que tiene almacenados y los procesa. Un predicado que puede determinar el procesamiento de los mensajes es, por ejemplo, `ballIsOurs`, que indica si algún miembro del equipo tiene la posesión de la pelota.

El módulo de toma de decisiones, en el modelo del mundo, tiene una lista donde almacena la información sobre los demás jugadores del equipo. En dicha lista se guardan los datos recibidos a través del procesamiento de los mensajes. Para cada mensaje recibido, el modelo del mundo actualiza la estructura de información del jugador que envió el mensaje. Si esta estructura no existe, es decir, el mensaje recibido es el primero que se recibe de este robot, entonces el modelo del mundo da de alta este nuevo jugador compañero y crea la estructura para preservar su información. Cuando este primer mensaje es recibido, si el mismo no incluye la identificación del rol del robot que lo envía, automáticamente el robot que recibe el mensaje dispara un mensaje consultando esta información al robot que hizo el envío. Típicamente sucede cuando un robot se agrega al equipo luego que otros robots ya estén participando del juego. En este caso, el nuevo robot enviará su mensaje de introducción al juego, indicando su *id* y su *rol* a los otros robots, y luego comenzará a recibir mensajes del resto de los jugadores del equipo. Dado que no conocerá a estos jugadores, para cada mensaje de un jugador nuevo, creará la estructura e inmediatamente enviará un mensaje a ese robot en particular consultando su rol y otra información básica que corresponda. El mensaje enviado tendrá como identificador `COMM_PARAM_ASK_ROLE`.

La información que se almacena en los mensajes es, por ejemplo: la acción tomada por el jugador que envía el mensaje, si está en posesión de la pelota, si está viendo la pelota, o si está viendo algún otro elemento significativo del juego. Cuando se recibe un nuevo mensaje que representa un dato previamente guardado, el mismo se actualiza. Además, se guarda un *timestamp*

indicando fecha y hora de recepción del mensaje. Esto último se utiliza para darle una noción de envejecimiento al dato de forma de poder definir un tiempo de validez del dato a la hora de usarlo en la evaluación de las reglas.

La cola de mensajes también se utiliza como vía de comunicación entre el hilo principal y el hilo que procesa los mensajes, haciendo posible terminar la ejecución del programa de manera controlada. Como el hilo que procesa los mensajes está la mayor parte del tiempo bloqueado esperando leer mensajes de la red, se debió implementar un mecanismo que lo desbloquee luego que se dejan de recibir mensajes. Para solucionar esto, en la implementación del destructor de la instancia de la clase se crea un mensaje con origen y destinatario el propio robot, y con el tipo de mensaje `COMM_PARAM_EXIT`.

El listado completo de los diferentes tipos de mensaje está definido en el archivo `Constants.h`. A continuación se mencionan los tipos de mensaje, su semántica y posibles valores:

- `COMM_PARAM_START`: mensaje interno de comienzo de ejecución del hilo. Cuando se envía este mensaje, se envía también mensaje adicional indicando rol.
- `COMM_PARAM_ROLE`: comunica el rol del robot en el campo valor. En esta versión, los posibles valores son: arquero (1) y jugador (2).
- `COMM_PARAM_BALL_IN_SIGHT`: el robot emisor indica que tiene la pelota en su rango de vista. Generalmente acompañado de un mensaje adicional indicando la distancia.
- `COMM_PARAM_BALL_DISTANCE`: mensaje que indica a qué distancia observa el robot la pelota. La distancia es enviada en el campo valor en centímetros.
- `COMM_PARAM_ACTION_TAKEN`: mensaje que indica en el campo valor cuál fue la acción ejecutada por el robot emisor.
- `COMM_PARAM_ASK_ROLE`: mensaje de tipo *pregunta* enviado por el robot para consultar el rol de los robots destinatarios.
- `COMM_PARAM_GK_POS_X`: indica en el campo valor la posición del arquero respecto del centro del arco en el eje x.
- `COMM_PARAM_GK_POS_Y`: indica en el campo valor la posición del arquero respecto del centro del arco en el eje x.
- `COMM_PARAM_EXIT`: mensaje interno que habilita al modelo del mundo a continuar la ejecución descartando el procesamiento de mensajes.

Existen otras constantes que se definen en el sistema y son utilizadas para manipular el comportamiento del módulo de comunicación:

- `COMM_BROADCAST`: valor utilizado en el campo *idDestination* para indicar que el mensaje debe ser recibido por todos los miembros del equipo.
- `COMM_TTL`: tiempo máximo de vida de un mensaje (en ms). Luego de superado, el mensaje pierde validez. Valor por defecto: $10000ms$.

5.2.1. Monitorización de la comunicación

El desarrollo del módulo de comunicación indujo la necesidad de contar con un aplicativo que permita capturar e interpretar el tráfico de red entre robots en forma sencilla.

Si bien herramientas de monitorización de red genéricas como *Wireshark*[4] pueden resultar útiles, resulta tedioso aplicar los filtros de red correspondientes. Asimismo, no resulta sencillo interpretar la mensajería adecuadamente en relación a la codificación interna que utiliza *FutRob* para nombrar acciones y tipos de mensaje. Por lo tanto, se desarrollaron dos aplicaciones `commRecv` y `commSender` que permiten recibir el tráfico de mensajería de *FutRob* así como enviar mensajes simulando ser un robot del equipo respectivamente.

En el directorio `./comm` se incluyen los archivos fuente, `CommSender.cpp` y `CommReceiver.cpp`, así como un *makefile* de compilación. Ambos fuentes están basados en el archivo que define las constantes del sistema `Constants.h` y lo incluyen desde el directorio padre con la sentencia que se observa en el fragmento 5. Esto permite que ambas aplicaciones mantengan una sintonía con la estructura del programa *FutRob*.

Fragmento 5: Dependencia de CommReceiver.cpp y CommSender.cpp

```
1 #include "./../Constants.h"
```

En la inicialización de `commRecv` la aplicación pide al usuario configurar los parámetros de *id* y *rol* del robot que emulará, como se observa en la figura 25. Es decir, recibirá los mensajes de igual forma que el robot indicado. Cualquier mensaje que circule por la red y sea para otro destinatario, el sistema mostrará una alerta indicando que se recibió un mensaje pero fue descartado. En caso que el mensaje sea aceptado, se desplegará en pantalla su contenido completo, es decir: remitente, destinatario, tipo de mensaje, valor de mensaje. El programa cuenta con la opción de aceptar todos los mensajes de la red, funcionando como un espía de toda la mensajería; para esto se debe establecer el valor $id = 255$.

[4] Sitio oficial Wireshark: `https://www.wireshark.org`

En la aplicación `commSender`, el sistema ofrece las opciones de enviar un mensaje (msg) o salir (q) del programa. Para enviar un mensaje, el programa solicita uno a uno los cuatro campos del mensaje a enviar. Es decir: origen, destinatario, tipo de mensaje, valor. En la figura 26 se observa un envío de mensaje del robot $id = 2$ al robot $id = 3$, tipo de mensaje 20 (`COMM_PARAM_ACTION_TAKEN` según `Constants.h`), y con el valor 19 (`ACTION_KICK_BALL` según `Constants.h`).

```
[MBP-de-Paul:comm paulgreen$ ./commRecv
CommReceiver!
-->> Hint: choose id=255 to receive all msgs <<--
robot id > 1
robot role > 2
Waiting for messages...

-----| [msg=1] MSG REJECTED |-----
Waiting for messages...
```

Figura 25: Inicialización *commRecv*

```
[MBP-de-Paul:comm paulgreen$ ./commSender
CommSender!
action (msg|q) > msg
action > int msg src > 2
action > int msg dest > 3
action > int msg param > 20
action > int param value > 19

action (msg|q) > q
Program is unloading....
```

Figura 26: Inicialización *commSender*

Estas aplicaciones permiten monitorizar el diálogo entre robots e incluso generar escenarios de simulación mediante el envío de mensajes personalizados a demanda. Son especialmente útiles en instancias de desarrollo y para análisis de comportamientos extraños o fallas en la aplicación *FutRob*.

5.3. Módulo de visión

El módulo de visión utilizado en el proyecto tomó el resultado del trabajo realizado en el proyecto *FutBot*[13], que se encargó de sustituir el módulo de visión *HaViMo* por la cámara web Logitech C920, aplicando los cambios necesarios sobre los desarrollos realizados en los proyectos *VisRob*[10] y *VisRob 2*[11].

El reconocimiento de objetos a partir de la captura de imágenes se realiza mediante una técnica de agrupación de píxeles contiguos de similar color. El modelo asume que píxeles contiguos de un mismo color conforman un mismo objeto. A cada grupo de estos píxeles se le denomina *"blob"*. Además, para completar la definición de este modelo, se utiliza un umbral de tamaño mínimo para conformar un *blob*, para evitar que 4 píxeles adyacentes de un mismo color ya determinen uno. Es deseable controlar que un conjunto de píxeles adyacente tenga un tamaño considerable para ser identificado como un *blob*, que potencialmente será identificado como un objeto. Esto se define en el archivo `utilsBlobs.cpp` de la librería del proyecto *FutBot*[13] con la constante `MIN_BLOB_SIZE`. El valor utilizado en el proyecto es de 150 píxeles. Cada imagen capturada puede contener una cantidad variable de *blobs*, por lo tanto, el *procesamiento de imágenes* se trata de la conversión de las imágenes

obtenidas por la cámara en conjuntos de *blobs*, cada uno con una determinada forma, tamaño y color. Luego de completado este primer proceso, se trabaja sobre cada *blob* para comprobar o descartar una serie de hipótesis que se plantean suponiendo que se trata de algún objeto reconocido por la visión. Se observa tamaño, forma, y color del objeto para concluir si se trata de un objeto conocido o no. El rango de colores de cada objeto queda establecido en el archivo `futbotVision.conf` como se observa en el el fragmento 7.

Dependiendo de las condiciones de luz del ambiente, o incluso de eventuales cambios en los códigos de colores de los objetos, se debe ajustar el archivo `futbotVision.conf` para garantizar que los objetos se identifiquen correctamente. Para esto, se cuenta con una aplicación satélite desarrollada por el proyecto *FutBot*[13] que muestra en vivo el video capturado por la cámara web y permite seleccionar múltiples colores, de a uno, utilizando el puntero del ratón en una computadora personal. A medida que se agregan colores, el programa aplica sobre la imagen la máscara que utilizaría para interpretar el *blob* en una ventana aparte. Esto permite definir el rango de colores completo para el objeto, desde el más opaco hasta el más brillante, pudiendo comprobar en vivo si la máscara a aplicar reconocería el *blob* como el objeto buscado. En el documento de "Futbot: Pruebas de visión" [24] se incluye un análisis detallado de estos aspectos.

Además del tamaño en píxeles, otro criterio que se aplica para el filtrado de objetos que se consideran identificados por la visión es la distancia a la que son detectados. En el archivo `vision.h` de la librería de visión, la constante `MAX_DISTANCE` se establece el umbral máximo de distancia a considerar. Para la implementación del proyecto se utilizó el valor $800cm$. Es decir, si un objeto se detecta a más de $800cm$, entonces se descarta. De todas formas, las pruebas del sistema con la resolución utilizada (432x240) arrojan resultados considerablemente menores: la pelota es detectada a una distancia máxima del orden de $200cm$. Esto se podría mitigar aumentando la resolución de la captura de imágenes, lo que impactaría en: una mejor calidad de imagen ofreciendo mayor precisión, y un mayor nivel de detalle a nivel de píxeles, es decir, el objeto pelota se identificaría con una mayor cantidad de píxeles.

El procesamiento de imágenes representa el ciclo de ejecución de menor frecuencia del sistema, es decir: el más costoso. Se realizaron pruebas de reconocimiento de objetos midiendo la utilización del recurso procesador según diferentes configuraciones de resolución de la cámara web. La resolución utilizada originalmente en el código del proyecto *VisRob 2* [11] era de 160x120, dado que el módulo de captura de imágenes utilizado (HaViMo) ofrece prestaciones reducidas. Luego, en el proyecto *FutBot* [13], al sustituir el hardware de visión por la cámara Logitech C920, realizaron pruebas utilizando las resoluciones 320x240 y 640x480 píxeles. De acuerdo con la documentación, los

tiempos de respuesta del sistema considerando escenas de complejidad media para estas resoluciones arrojaron tiempos promedio de 150 milisegundos y 550 milisegundos respectivamente. En el proyecto *FutRob*, se realizaron pruebas que corroboran estos tiempos de respuesta y se observó una leve mejoría utilizando una Raspberry Pi 2. Producto de los ensayos realizados, se observó que las resoluciones que ofrecen un mejor desempeño para los casos a implementar son aquellas con una relación de aspecto más estirada, como en el formato utilizado en el cine, 16:9. Aprovechando las características que ofrece la Raspberry Pi 2: una leve mejora en los tiempos de procesamiento y la posibilidad de desarrollar en hilos de ejecución independientes, se optó por partir de la resolución 320x240 (relación 4:3) y ampliar el ancho de la imagen, estableciendo una resolución de 432x240 (relación 18:10), una de las configuraciones disponibles en la cámara utilizada.

Para obtener un mejor rendimiento a nivel de tiempos, se diseñó una arquitectura que contempla la ejecución del módulo de visión en un hilo de ejecución independiente. Dado que el procesador utilizado cuenta con 4 hilos de procesamiento a nivel de hardware, destinamos un hilo dedicado exclusivamente a la visión. Esto mejora considerablemente los tiempos de ejecución ya que se minimizan los cambios de contexto y se agrupa el uso de memoria compartida sobre un único núcleo de procesador.

Este diseño permitió además que el ciclo de ejecución del sistema no esté sujeto a los tiempos de ejecución de un ciclo de procesamiento de los datos de visión. El componente central del sistema ejecuta evaluando reglas, consumiendo la información de los componentes de comunicación y visión, y tomando decisiones a una frecuencia totalmente independiente. Paralelamente, el módulo de visión ejecuta actualizando la información en memoria compartida para cada objeto modelado luego de completar cada ciclo. Bajo estas condiciones, se obtuvo empíricamente el tiempo de ejecución de un ciclo completo de visión, que ronda los $250ms$, mientras que el módulo central ejecuta a una velocidad comparable a la frecuencia del procesador: 900 MHz.

Como parte del proceso de incorporación de la implementación del módulo de visión de *FutBot*, se debieron realizar principalmente dos cambios: calibración de la distancia focal, y calibración del tamaño mínimo del *blob* considerado por la visión. Este último cambio surgió producto de los ajustes realizados en la resolución de la cámara para la toma de imágenes.

El módulo de visión retorna como resultado primario luego de identificar un objeto la información de su posición. Ésta se describe como el ángulo respecto de la perpendicular al robot y la distancia respecto a la proyección al plano horizontal del centro del robot. Durante la etapa de pruebas, se detectó que el módulo de visión del proyecto *VisRob 2* [11] adaptado por *FutBot* [13] devolvía un ángulo positivo siempre que el objeto se encontrara hacia la

izquierda de la línea de referencia, y el valor constante -90 en caso que se encontrara hacia la derecha. Se realizaron ajustes sobre este comportamiento y se adoptó la convención de utilizar valores positivos para posiciones hacia la derecha de la perpendicular al robot y valores negativos para posiciones a la izquierda.

Para el cálculo de la distancia de un objeto reconocido, el modelo utiliza como dato de entrada el valor de la distancia focal como se muestra en la ecuación 12, así como la dimensión *ancho* del objeto. El valor de la distancia focal es propio de cada cámara y de la resolución utilizada. Para obtenerlo en la práctica, se realizan pruebas de laboratorio donde las otras 3 medidas son conocidas. En la implementación del proyecto predecesor, el valor utilizado fue de 164 píxeles (px). Pero en los ensayos del proyecto *FutRob*, se obtuvo un nuevo valor de calibración: $280px$.

$$\frac{ancho(cm)}{distancia(cm)} = \frac{ancho(px)}{distancia(px)} \tag{12}$$

Luego de obtener el valor utilizado para el cálculo de la distancia real del objeto en el módulo de visión, se realizaron pruebas para contrastar contra la medida de la distancia real del objeto al robot. Para esto, se montó un escenario como se muestra en la figura 27 utilizando como objeto la pelota y se tomaron conjuntos de 10 mediciones calculadas por el módulo de visión fijando la distancia del objeto y la posición de la cámara en el ángulo vertical, conocido como *tilt*. En sucesivos ensayos, se mantuvo fija una dimensión y se modificó la otra, y se registró el promedio de las 10 medidas leídas en cada caso. Los datos obtenidos se observan en la tabla de la figura 28.

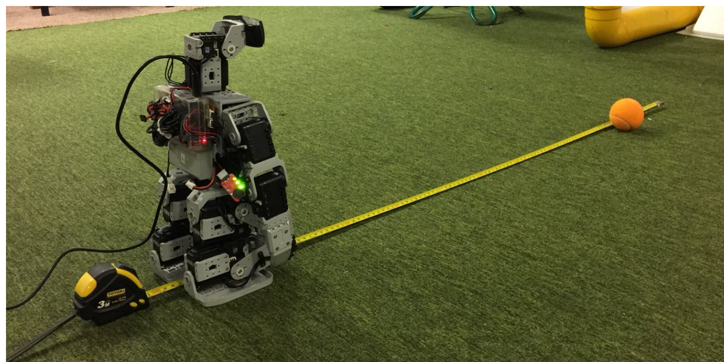

Figura 27: Ensayos para calibración de visión.

Dato visión	Tilt (grados)												
Dist. pelota (cm)	**0**	**-5**	**-10**	**-15**	**-20**	**-25**	**-30**	**-35**	**-40**	**-45**	**-50**	**-55**	**-60**
150	155.07	164.83	169.14	171.7	173.56	173.7	166.46						
130	127.6	135.4	140	140.1	141.7	146.2	139.8						
110	105.44	107.99	112	117.9	118.3	121.9	118.7	111.3					
100	94.2	96.2	102.3	104.9	106.6	109.7	106.8	101					
90	85.4	88.2	91.2	93.3	96.9	97.3	97.1	92.1					
80	75.09	76.7	79.3	81.5	84.5	84.2	85	81.8	77.7				
60		60.4	57.14	59.3	61.2	63.7	64.6	63.2	61	57.2			
40				40.8	39.6	40.5	42.3	42.9	42.6	41.2	39.2	36	
20								21.7	22.4	22.7	22.9	22.3	21
5												10.77	9.9

Figura 28: Ensayo: medidas de distancia calculadas por el módulo de visión.

Luego de registrados los cálculos de distancia realizados por el módulo de visión, se continuó el análisis de la información tomando la medida del desvío absoluto entre la distancia real de la pelota y la distancia calculada por el módulo de visión. Estos datos se colocaron en el mapa de calor de la figura 29, tomando como desvío 0 el centro y considerando en tonalidades de verde los desvíos positivos (la medida calculada es mayor a la medida real), y en tonalidades de rojo los desvíos negativos.

Error absoluto	Tilt (grados)												
Dist. pelota (cm)	**0**	**-5**	**-10**	**-15**	**-20**	**-25**	**-30**	**-35**	**-40**	**-45**	**-50**	**-55**	**-60**
150	5.07	14.83	19.14	21.7	23.56	23.7	16.46						
130	-2.4	5.4	10	10.1	11.7	16.2	9.8						
110	-4.56	-2.01	2	7.9	8.3	11.9	8.7	1.3					
100	-5.8	-3.8	2.3	4.9	6.6	9.7	6.8	1					
90	-4.6	-1.8	1.2	3.3	6.9	7.3	7.1	2.1					
80	-4.91	-3.3	-0.7	1.5	4.5	4.2	5	1.8	-2.3				
60		0.4	-2.86	-0.7	1.2	3.7	4.6	3.2	1	-2.8			
40				0.8	-0.4	0.5	2.3	2.9	2.6	1.2	-0.8	-4	
20								1.7	2.4	2.7	2.9	2.3	1
5												5.77	4.9

Figura 29: Ensayo: desvío de medida calculada respecto a medida real.

Observando el mapa de calor, se evidencia una tendencia a minimizar el error a lo largo de la diagonal, que corresponde a cuando la cámara tiene el objeto en el centro de su campo de visión. Luego, hacia los lados de la diagonal, se comete un error opuesto hacia cada lado. En la subdiagonal el error es negativo, y en la supradiagonal el error es positivo.

Por lo tanto, para reducir el error en el cálculo de la distancia del objeto, resulta interesante obtener cuál sería el mejor ángulo para posicionar la cámara tal que minimice el error de cálculo en la distancia del objeto. Para esto, teniendo la medida calculada y conociendo el ángulo de inclinación

de la cámara, aplicamos el teorema de Pitágoras para calcular en cada caso cuál sería el mejor ángulo de tilt. En la grilla de la figura 30 se incluyen los resultados y se observa para cada caso una desviación estándar del orden de la unidad para cada posición del objeto.

Como resultado, se implementó una acción de alto nivel que, dados los datos calculados por la visión en relación a la distancia del objeto, y dada la posición de la cámara (ángulo de tilt), devuelve cuál es el mejor valor de tilt para ubicar la cámara. La función implementada tiene la siguiente firma:
`double BestVerticalAngle(double distance)`

Ángulo de optimización	**Tilt (grados)**													**Ángulo óptimo y desviación de datos**	
Dist. pelota (cm)	**0**	**-5**	**-10**	**-15**	**-20**	**-25**	**-30**	**-35**	**-40**	**-45**	**-50**	**-55**	**-60**	**avg**	**stdev**
150	-13.1	-12.3	-12	-11.8	-11.7	-11.7	-12.2							**-12.11**	0.4947
130	-15.8	-14.9	-14.4	-14.4	-14.3	-13.8	-14.4							**-14.57**	0.6291
110	-18.9	-18.4	-17.8	-17	-16.9	-16.5	-16.9	-17.9						**-17.54**	0.8434
100	-20.9	-20.5	-19.4	-18.9	-18.7	-18.2	-18.6	-19.6						**-19.35**	0.9487
90	-22.9	-22.2	-21.5	-21.1	-20.4	-20.3	-20.3	-21.3						**-21.25**	0.9442
80	-25.6	-25.1	-24.4	-23.8	-23.1	-23.1	-23	-23.8	-24.9					**-24.09**	0.9597
60		-30.8	-32.2	-31.3	-30.5	-29.5	-29.1	-29.7	-30.5	-32.2				**-30.64**	1.1137
40				-41.4	-42.3	-41.6	-40.4	-40	-40.2	-41.1	-42.6	-45		**-41.62**	1.553
20								-58.9	-58.1	-57.8	-57.5	-58.2	-59.7	**-58.37**	0.8042
5												-73.3	-74.6	**-73.95**	0.9192

Figura 30: Conclusiones: optimización de ángulo de tilt para cálculo de distancia.

Durante las instancias de desarrollo y pruebas, la información que recibe el sistema a través del módulo de visión resulta esencial. Producto de esto, en reiteradas ocasiones se debió montar un escenario tal que la información percibida por el sistema de visión se corresponda con la entrada que se quiere proveer para reproducir un caso particular. Por ejemplo, para realizar pruebas de evaluación de una regla del tipo *si veo el arco rival y tengo la pelota, patear* o *si veo la pelota, informo su posición respecto de mi*, es necesario que el sistema obtenga una información determinada y precisa respecto del arco rival o la pelota. Para probar estos casos, fue necesario ubicar al robot sobre una superficie color verde, y en presencia de los objetos *arco rival* o *pelota* a una distancia y posición específicas. Este tipo de pruebas demostró demandar no solamente un tiempo de preparación considerable del escenario, sino que la disponibilidad continua de los elementos necesarios y la detallada precisión para ubicarlos en la escena. Además, ante cada cambio de condiciones de luz, se debía calibrar nuevamente los rangos de colores en el archivo `futbotVision.conf`.

Estos puntos motivaron el desarrollo de un sistema de visión *stub*. Un *stub* es un componente de código utilizado como sustituto de una funcionalidad. Habitualmente pretenden simular el comportamiento de una lógica

existente, como pueden ser respuestas de un sistema externo, o sustituir un código aún no desarrollado entregando respuestas fijas (constantes) pero con la estructura de formato esperado.

Para el caso del componente de visión *stub*, se desarrolló un módulo paralelo y con idéntica estructura al original. A diferencia del módulo de visión original, el *stub* devuelve información de presencia y posición de objetos tomados desde un archivo de texto con un formato estructurado como se observa en el fragmento 6. El archivo consta de una serie de líneas con 4 valores separados por coma, que son, en orden: número de escena, objeto, ángulo en grados y distancia en centímetros.

Fragmento 6: Archivo de entrada para stub de visión

```
1 1,ball,12,34
2 1,opp_goal,54,65
3 1,partner,−10,15
4 6,ball,−76,57
5 6,opp_goal,−82,73
```

Las líneas 1 a 3 corresponden a la escena 1, mientras que las líneas 4 y 5 corresponden a la escena 6. El concepto de escena emula la continuidad en el tiempo del módulo de visión. El sistema de visión *stub* está desarrollado para procesar cada escena en un ciclo de visión, con una latencia configurable mediante la constante `VISION_LATENCY_MS` del sistema. El valor por defecto es de 250 ms, equivalente al tiempo de ejecución de un ciclo de visión del módulo de visión que procesa las imágenes directamente desde la cámara web. El sistema comienza los ciclos de ejecución considerando la escena 1, y luego incrementa en 1 la escena en cada ciclo de ejecución. Para cada ciclo, toma todas las líneas de valores para la escena actual. Por ejemplo, para la escena 1, en el primer ciclo toma los 3 objetos como si se observaran simultáneamente. Luego, para las escenas 2 a 5 el sistema asume que sigue viendo los mismos objetos en el mismo lugar (es decir, la escena permanece sin cambios). Finalmente, para el sexto ciclo de ejecución, observa los objetos pelota y arco rival en nuevas posiciones, pero ya no observa al jugador compañero de equipo.

Los objetos disponibles para utilizar en el módulo de visión *stub* son exactamente los reconocibles por el módulo de visión *real*:

- `BALL`: pelota.
- `PARTNER`: robot compañero de equipo.
- `OPPONENT`: robot de equipo rival.

- `MY_GOAL`: arco propio.
- `OPP_GOAL`: arco rival.
- `MY_LATERAL_POST`: poste de arco propio.
- `OPP_LATERAL_POST`: poste de arco rival.
- `RIGHT_LANDMARK`: marca de campo derecha.
- `LEFT_LANDMARK`: marca de campo izquierda.

El modo de visión del sistema puede alternarse entre *real* y *stub* en forma sencilla desde el archivo de configuración inicial `ini.config`. El modo *stub* se utiliza típicamente en escenarios de laboratorio para realizar pruebas de concepto, ajustes de calibración de componentes, o cambios de configuración. Por defecto, el sistema habilita el modo *real*, que será el utilizado en los flujos de ejecución de campo. Para habilitar el modo *stub* se debe indicar en el archivo de configuración la ubicación del archivo fuente que contiene los datos de visión, por ejemplo:

- `visionStubEnabled=1`
- `visionStubSrc=./futrob/vision/objSampleData.input`

El modo *real* toma dos tipos de configuración: la configuración propia de la cámara, y la configuración de los rangos de colores a interpretar. Por configuración de cámara se entienden los parámetros distancia focal y resolución de imagen. Ambas configuraciones están establecidas en la librería de visión desarrollada por el proyecto previo de visión de robots, *VisRob 2* [11]. Los rangos de colores a interpretar son cargados dinámicamente por el sistema en tiempo de ejecución al momento de cargar la aplicación y son tomados desde el archivo de configuración `futbotVision.conf`, que indica al sistema la relación entre los conceptos a identificar y los rangos de colores en los que se encuentra cada uno. En el fragmento 7 se incluye una porción del archivo `futbotVision.conf`.

Fragmento 7: Archivo de configuración de relación objeto-rango de colores

```
1 # regiones a detectar por el modulo de vision
2 region ball
3 {
4     id=0
5     hsiMin={11,207,81}
```

```
6         hsiMax={19,255,202}
7     }
8
9     region field
10    {
11        id=1
12        hsiMin={25,7,125}
13        hsiMax={33,31,32}
14    }
15
16    region my_goal
17    {
18        id=2
19        hsiMin={21,208,82}
20        hsiMax={24,253,145}
21    }
```

Para indicar la gama de colores entre los que se puede encontrar el objeto a identificar, se utilizan 2 ternas de valores con la representación HSV[5] de las tonalidades de los extremos: la más opaca y la más brillante.

5.4. Módulo de manejo de motores

El módulo de manejo de motores es el responsable de llevar a cabo las acciones determinadas por el manejador de acciones. Una vez que se determina una acción a ejecutar, el hilo independiente donde ejecuta el módulo manejador de acciones solicita al módulo de manejo de motores que implemente la acción interactuando con los motores en forma coordinada para producir el resultado deseado.

Cada robot cuenta con 20 motores AX-12 [25]: 18 componen el cuerpo y 2 la cabeza. La distinción tiene lugar ya que el conjunto de movimientos se encapsula en 2 categorías: movimientos del cuerpo y movimientos de la cabeza. Los movimientos de la cabeza controlan la posición de la cámara, por lo que están estrechamente ligados con el módulo de visión y ejecutan acciones que dan soporte a actividades de búsqueda y seguimiento de objetos. Por otra parte, los 18 motores que componen el cuerpo son utilizados para llevar a cabo acciones de movimiento como traslación, rotación, movimientos

[5] HSV (Hue, Saturation, Value): modelo para la representación tridimensional del color basado en los componentes de tinte, matiz o tonalidad (*hue*), saturación (*saturation*) y brillo o valor (*value*)

laterales, recuperación de caídas, impacto de pelota o atajadas, por lo que están fuertemente vinculados al módulo manejador de acciones.

El esquema arquitectónico de los componentes lógicos de la solución puede abordarse con un enfoque *top-down* donde, a partir del módulo manejador de acciones con un alto grado de abstracción se combinan movimientos que finalmente al más bajo nivel son colocados en el bus de datos de la Raspberry Pi. Para esto último el sistema cuenta con un módulo `serial.c` responsable de la comunicación a través de los puertos UART[6] de la Raspberry Pi. En la figura 31 se observa una descripción de las responsabilidades y cómo se combinan los componentes en las diferentes capas según el enfoque descrito.

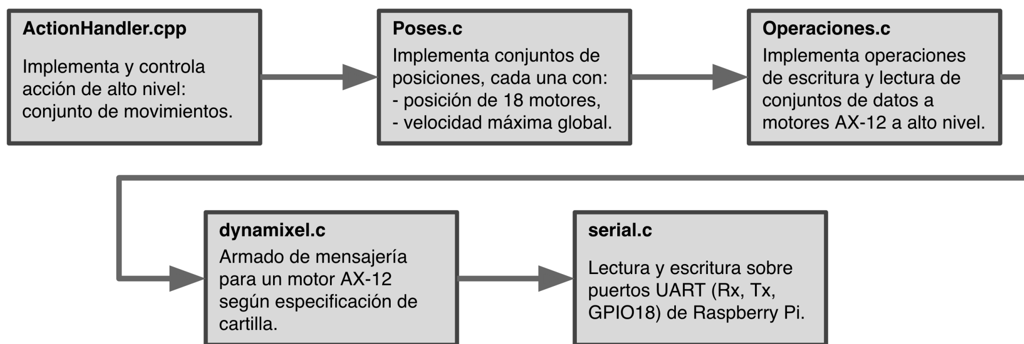

Figura 31: Diagrama de flujo de manejo de motores

Como se describe en diagrama 31, para la invocación de cada nueva posición del robot, el módulo `Operaciones.c` recibe desde el archivo `Poses.c` el valor del ángulo de destino para cada uno de los 18 motores del robot, así como una velocidad máxima para el movimiento. Se utiliza un único valor de velocidad que se extrapola hacia los 18 motores para mantener la cohesión en la traslación de los motores hacia la nueva posición. Es decir, lograr que los 18 motores se muevan al unísono, comenzando todos en prácticamente un mismo instante y terminando al mismo tiempo.

Para minimizar la cantidad de escrituras hacia los motores, el sistema mantiene en memoria un vector que guarda la posición actual de los 18 motores y es actualizada ante cada escritura de nuevas posiciones. Existe un posible desfase entre el vector de posición en memoria y las posiciones reales de los motores. Esta situación puede surgir cuando un motor no completa un movimiento. Pero, dado que las posiciones se escriben grabando el dato de cada nueva posición en caso de cambiar, el desfase se corrige en el próximo intento de escritura de posición.

[6] UART, Universal Asynchronous Receiver-Trasnsmitter: controladora de puertos serie.

El dato de la velocidad a la que debe ejecutarse el cambio de posición de los motores se extrapola hacia cada motor que cambiará su posición, debiéndose indicar para cada uno el ángulo de destino y la velocidad de rotación. Este procedimiento puede describirse en los siguientes pasos:

1. Se hace un procesamiento previo comparando el vector de posición de destino con el vector de posición actual y se establece el valor constante `NCH` (por "no change") como el ángulo de destino para los motores que no cambian de posición.

2. Se analizan los recorridos de los motores que cambian de posición y se preserva el de mayor amplitud.

3. Se recorre el vector de nuevas posiciones de motores para todos los motores que cambian de posición, y se establece la velocidad de rotación de cada uno a partir de la velocidad máxima general para el movimiento `MAX_VEL` con el siguiente cálculo:

$$v_i = |posDestino_i - posActual_i| \cdot \frac{MAX_VEL}{MAX_DIST} \tag{13}$$

Cada movimiento *complejo*, como por ejemplo caminar, es desagregado en movimientos más simples, como por ejemplo *levantar el pie izquierdo*, *apoyar el pie izquierdo mientras se levanta el pie derecho*, *llevar tronco, piernas y pies a la posición de equilibrio*. A su vez, cada movimiento *simple* está compuesto por una serie de conjunto de posiciones de los 18 motores, que determinan diferentes estados del movimiento con un enfoque *stop-motion*. Para ejecutar cada movimiento *simple* el sistema ejecuta una a una las posiciones intermedias moviendo los 18 motores en forma simultánea, comenzando y terminando a la misma vez. El sistema no comienza a ejecutar una nueva posición hasta que los 18 motores completan el movimiento para llegar a la posición actual.

Luego de escribir en cada motor el ángulo de destino y la velocidad de rotación, el sistema debe esperar a que el movimiento se complete. Es decir, que cada motor llegue a la posición final. En implementaciones previas esto se resolvía con un *busy-waiting* manteniéndose consultando constantemente la posición de cada motor hasta verificar que todos hubieran alcanzado la posición de destino. Esto degrada fuertemente el rendimiento de la solución puesto que cada operación de lectura tiene un alto costo. La solución desarrollada realiza el cálculo de tiempo de rotación en función de la velocidad para algún motor -dado que todos los motores tardan el mismo tiempo en alcanzar la posición de destino-. Para minimizar errores de cálculo por redondeo, se

toma como referencia el motor con mayor recorrido. Combinando este dato con el dato del fabricante respecto de la velocidad angular de rotación de los motores AX-12 desde la hoja de especificaciones técnicas y contrastado empíricamente (datos configurables en el archivo `ini.config` por las variables `MAX_RPM_SERVO` y `MAX_VEL_SERVO`), se realiza el cálculo que se observa en la ecuación 14:

$$
\begin{aligned}
revs_per_ms &= \left[\frac{vel \cdot MAX_RPM_SERVO}{MAX_VEL_SERVO}\right] \cdot \frac{1}{60} \cdot \frac{1}{1000} \\
revs &= \frac{distance}{360} \\
ETA &= \frac{revs}{revs_per_ms}
\end{aligned}
\tag{14}
$$

Los motores están conectados entre sí mediante un cableado con esquema de cadena margarita o *daisy chain*. En este tipo de conexiones los componentes tienen que comportarse de forma cooperativa, es decir, solamente un motor a la vez puede utilizar el el bus de comunicaciones. Cada motor tiene 2 conectores MOLEX hembra de 3 pines, donde se conecta un cable entrante y uno saliente. Un pin es conectado a Vcc, otro a tierra (GND) y el tercer pin es usado para transferencia de datos (DATA). Luego, los pines DATA de los motores están conectados a la placa controladora Raspberry Pi a través de la interfaz de hardware Pi2AX12 descrita en la sección 4.1.

Siempre que se envía un mensaje con una instrucción a un motor, el mismo responde otro mensaje con el resultado de la instrucción. Para completar exitosamente este diálogo que implica enviar el mensaje con la instrucción y leer la respuesta satisfactoriamente se tienen que completar, en orden, los siguientes eventos. En primer lugar, establecer al pin `GPIO18` de la Raspberry Pi el valor 1, e inmediatamente proceder al envío del mensaje con la instrucción a través del puerto serial (Tx). Se debe mantener el pulso sobre el pin `GPIO18` en 1 durante un tiempo para permitir que el el mensaje sea colocado correctamente sobre el bus de datos y enviado hacia los motores. Luego de transcurrido un tiempo, se debe colocar el pin `GPIO18` en 0 para permitir que el mensaje de respuesta enviado desde el motor sea recibido a través de la placa Pi2AX12 hacia la Raspberry Pi. El envío de una instrucción pude verse en el pseudocódigo del fragmento 8.

La ventana de tiempo que se debe aguardar desde que se coloca el pin `GPIO18` en 1, se realiza el envío del mensaje, y se coloca el pin `GPIO18` en 0 es clave para lograr el éxito en la comunicación. Si la ventana de tiempo que se mantiene el pin en 1 es corta, no se permitirá colocar completamente el mensaje en el bus de datos y de esta forma no se completará el envío del mensaje

Fragmento 8: Fragmento de Serial.c, implementación de la comunicación serial

```c
#define DELAY_US 90
void serial_write(unsigned char *pData, int numbytes)
{
 // Setea el pin GPIO18 en 1
 digitalWrite(1, 1);
 // Escribe en el buffer de datos
 write(fd, pData, numbytes);
 // Espera a que se envie el mensaje
 delay_us_custom(DELAY_US);
 // Setea el pin GPIO18 en 0
 digitalWrite(1, 0);
}
```

desde la Raspberry Pi hacia los motores. Por contrapartida, si la ventana de tiempo que se mantiene el pin en 1 es demasiado prolongada, el mensaje de respuesta que coloque el motor en el bus de datos desaparecerá antes de permitir su recepción hacia la Raspberry Pi, por lo que se perderá la lectura de esta respuesta. Para realizar el ajuste fino de esta ventana de tiempo, se realizó una amplia serie de pruebas de laboratorio controladas enviando mensajes en forma masiva y controlando la recepción de las respuestas para diferentes ventanas de tiempo. El rango de valores de quiebre, límites superior e inferior donde se observó que una pequeña modificación degradaba ampliamente la comunicación, fue entre 60 y 120 microsegundos. Por lo tanto, se optó por el valor medio entre los extremos: 90 microsegundos. El valor está definido en la constante `DELAY_US` en el archivo `Serial.c` que implementa la comunicación serial, parte del fragmento 8. Esta situación condiciona el tiempo que el sistema debe permanecer en espera por ambos extremos: superior e inferior. El sistema debe esperar un tiempo mayor a 60 microsegundos pero menor a 120 microsegundos para asegurar un correcto funcionamiento. La precisión en el tiempo de espera, acotado superior como inferiormente, hace que se deba contar con una implementación estricta para el tiempo de espera que se mantiene un proceso. En la sección 5.4.1 se aborda en profundidad la implementación realizada bajo el nombre *delayMicroSecondsHard*.

El uso de este mecanismo es altamente sensible, ya que si se habilita el buffer tri-estado para leer información desde los motores, si aún no se ha terminado de enviar información hacia los motores, pueden generarse inconsistencias o comportamientos inesperados. Incluso podría provocar un daño

eléctrico en alguno de los componentes del circuito. Como política de implementación defensiva, y considerando los riesgos de daño eléctrico que se podría llegar a causar a los motores, se optó por mantener el pulso de este bus permanentemente en 0. Únicamente se cambia el valor a 1 en el método que implementa la escritura hacia los motores que se observa en el fragmento 8, que setea el valor del pin `GPIO18` en 1, escribe el dato a enviar, espera 90 microsegundos, y luego vuelve a establecer el valor del pin `GPIO18` en 0.

5.4.1. Librerías utilizadas

Para completar la migración de la implementación desde la plataforma CM-510 hacia la RaspberryPi fue necesario resolver las dependencias con librerías de bajo nivel.

Por un lado, las llamadas de sistema a librerías para manejo de *delays*. Originalmente se utilizaban componentes de la librería AVR Libc [26] disponibles en el chip ATMEL de la placa CM-510 (ATMega2561). Particularmente el módulo `util/delay.h`[7], y de él las funciones de espera en milisegundos y microsegundos de la figura 9.

Fragmento 9: Funciones delay de librería AVR Libc

```
1 void _delay_ms(double __ms)
2 void _delay_us(double __us)
```

Ambas funciones fueron reimplementadas utilizando la llamada a sistema nanosleep de UNIX (*UNIX system call*), cuya especificación se observa en el fragmento 10. Estas implementaciones se incluyeron en el módulo y se utilizaron para la implementación de FutRob con las firmas que se observan en el fragmento 11.

Fragmento 10: nanosleep(2)

```
1 #include <time.h>
2 int nanosleep(const struct timespec *req,
3              struct timespec *rem);
```

Fragmento 11: Implementación para FutRob

```
1 void delay_ms_custom(unsigned int ms);
2 void delay_us_custom(unsigned int us);
```

[7] Librería AVR util/delay.h: `http://www.atmel.com/webdoc/AVRLibcReferenceManual/group__util__delay.html`

La implementación de *nanosleep(2)* asegura que suspende la ejecución del hilo que la invoca por al menos el tiempo indicado por el parámetro **req*. Es decir, si se invoca con un parámetro de 100 nanosegundos, el núcleo asegura que se interrumpirá el hilo de ejecución y se devolverá el control luego de transcurridos esos 100 nanosegundos. Lo que no asegura es que el control se devuelva exactamente luego de transcurridos esos 100 nanosegundos, por lo que el tiempo pasado como parámetro a la llamada de sistema es una cota menor para el tiempo que el sistema mantiene al hilo en espera. Más precisamente, es el mínimo. Esto se debe en parte a retrasos introducidos por el despachador de procesos del sistema operativo anfitrión (*kernel dispatch latency*).

La llamada al método de espera en microsegundos se utiliza en el sistema exclusivamente para controlar el pulso que se coloca en el pin `GPIO18` de la Raspberry Pi que controla el estado del buffer 74LS241, como se explica en la sección 5.4. Esto permite alternar entre escritura en el bus de datos hacia los motores o lectura desde el bus de datos de los motores. Dado que este es un procedimiento sensible y que interactúa directamente con un componente de hardware externo, el sistema FutRob debe asegurar que el tiempo que se mantiene el pulso es estrictamente el necesario según las pruebas de laboratorio: 90 microsegundos.

Por lo tanto, considerando que la llamada de sistema *nanosleep(2)* no permite controlar el tiempo exacto que se mantiene al hilo de ejecución en espera, se debió buscar una forma que permita indicar exactamente qué tiempo se desea detener la ejecución. Para esto, se tomo como referencia la función *delayMicrosecondsHard* de la librería WiringPi [27], que recibe como parámetro la cantidad exacta de microsegundos que se debe interrumpir la ejecución y lo implementa mediante un *busy-waiting*.

Fragmento 12: Implementación de espera en microsegundos

```
1  void delayMicrosecondsHard (unsigned int howLong)
2  {
3   struct timeval tNow, tLong, tEnd;
4
5   gettimeofday (&tNow, NULL);
6   tLong.tv_sec  = howLong / 1000000;
7   tLong.tv_usec = howLong % 1000000;
8   timeradd (&tNow, &tLong, &tEnd);
9
10  while (timercmp (&tNow, &tEnd, <))
11   gettimeofday (&tNow, NULL);
12 }
```

Además, se utilizaron otras funcionalidades de la librería WiringPi, que ofrece funciones para la entrada y salida a través del puerto serial (pines) de la placa Raspberry Pi 2. Esta librería se distribuye bajo la licencia GNU LGPLv3 y puede ser utilizada en C y en C++. Está disponible para descargar directamente desde el repositorio `git://git.drogon.net/wiringPi`. Para la instalación se debe utilizar el script `wiringPiw/build`. Es requisito previo tener esta librería instalada para poder trabajar sobre el sistema. Particularmente esta librería es utilizada para controlar el valor del pulso eléctrico en el pin `GPIO18` de la placa Raspberry Pi. Esto se realiza directamente a través de la función `digitalWrite`, donde se indica el pin sobre el que se pretende escribir y el valor del pulso a colocar en el bus. En el fragmento 13 se incluye la firma de la operación.

Fragmento 13: WiringPi - escritura serial

```
1 void digitalWrite (int pin, int value);
```

Este pin se utiliza a nivel eléctrico como entrada para la placa Pi2AX12, conectado directamente al pin `1Enable` del chip 74LS421 que regula el comportamiento del buffer tri-estado, permitiendo que la información fluya desde o hacia los motores AX-12 según su valor. Para habilitar el bus para la escritura desde la Raspberry Pi hacia los motores, se debe colocar un pulso valor 1. Por contrapartida, para permitir la lectura desde la Raspberry Pi de los pulsos enviados por los motores, se debe colocar un valor 0. En el fragmento 8 de la sección 5.4 se observa la utilización de esta librería.

5.5. Módulo de registro de actividad del sistema

Durante el desarrollo de la solución, se marcó en forma reiterada la necesidad de contar con información de registro de actividad del sistema para contrastar con la realidad observada en el comportamiento del robot. Esta información se torna especialmente necesaria en casos de calibración del sistema y principalmente desarrollo y pruebas de integración.

Es necesario tener información precisa y detallada de cuál fue el flujo de ejecución de la lógica, de cuáles fueron las entradas percibidas y los cálculos realizados, y cuáles fueron las salidas producidas. Incluso pueden ocurrir situaciones en los que la lógica se comporte adecuadamente a lo largo de todo su ciclo de vida, pero que los resultados observables del comportamiento del robot no sean los esperados, y esto pueda producirse por problemas de falta de energía de alimentación del robot, o daños en los actuadores AX-12 responsables de traducir la salida de la lógica en movimiento.

Además, dada la arquitectura del software que se desarrolló, en hilos de

ejecución independientes y concurrentes, mutuo excluidos y que utilizan memoria compartida y técnicas de sincronización, comprender el ciclo de ejecución de la lógica reviste un mayor grado de complejidad. Se debe identificar en cada instante qué hilo es responsable de cada acción del sistema y cómo interactúa con el resto de los hilos.

Esto provocó el desarrollo de un módulo personalizado para registrar en forma íntegra el comportamiento del software en un archivo de log: `logger.c`. Éste expone operaciones de inicialización y finalización del registro, y 2 operaciones que dan la versatilidad para agregar una nueva línea al log o incluir información adicional (como parámetros) en la línea actual. En el fragmento 14 se observa la firma de estas operaciones.

Fragmento 14: Logger - operaciones de registro

```
1
2  /*
3  * Agrega linea en archivo de log con las etiquetas:
4  * timestamp, archivo, operacion, linea de origen.
5  * Bloquea otras llamadas hasta que todos los nParams
6  * son escritos en el archivo.
7  */
8  void log(string srcfile, string srcfun, int srcln,
9  *             string errMsg, int nParams);
10
11 /*
12 * Agrega parametros a la ultima linea de log
13 * (con el formato "nombre = valor").
14 */
15 void logParam(string param, string value);
```

Para agregar información de contexto al registro de la información en cada línea del archivo de log, en la invocación se envían como parámetros el nombre del archivo, función y línea desde donde se invoca. Para esto, en la invocación de la operación `void log`, se utilizan las variables de entorno que ofrece el sistema. Además, se incluye el parámetro `nParams` que indica la cantidad de parámetros que se agregarán en la línea. Esto es utilizado internamente por el módulo para la sincronización, evitando grabar parámetros en líneas de invocación hechas desde diferentes fuentes externas. El log queda reservado hasta que se completa la escritura de los `nParams` parámetros. Luego, para el registro de los parámetros adicionales, se envía únicamente el par nombre de parámetro y valor. En el fragmento 15 se incluye un ejemplo para cada llamada, y en la figura 32 se incluye un extracto del archivo de log generado

durante una ejecución para ilustrar su formato.

Fragmento 15: Logger - ejemplo de invocación

```
/*
* Nueva linea que tendra 2 parametros:
*/
log(__FILE__, __func__, __LINE__, "Err msg", 2);

/*
* Agrega un par parametro-valor en la linea actual.
* Para agregar 2 parametros, se invoca 2 veces.
*/
logParam("destinatario", "255.255.255.0");
logParam("mensaje", "COMM_PARAM_START");
```

```
1   [10-22-2016  16:18:17.894918][CommModule.cpp|SendMessage|191]: MSG SENT [bytes = 16]
    [destIP = 255.255.255.255]  [msgSrc = 1]  [msgDest = 255]  [msgParam =
    COMM_PARAM_BALL_DISTANCE]  [msgValue = 162]
2   [10-22-2016  16:18:17.895172][WorldModel.cpp|VisionThread|680]: BallData [angle =
    14.4208]  [dist = 162.624]  [time = 255]
3   [10-22-2016  16:18:18.99682][ActionHandler.cpp|ExecutorThread|875]: Execute
    lookAtBall208]  [dist = 162.624]  [time = 255]
```

Figura 32: Ejemplo de archivo de log.

Como se observa en el extracto de la figura 32, el archivo de registro de actividad del sistema (log) permite tener un conocimiento integral y concentrado del ciclo de ejecución del software. A su vez, para poder realizar un análisis más personalizado del comportamiento del robot, el módulo `logger.c` permite controlar qué módulos del sistema registran información en el archivo, prendiendo o apagando cada uno de ellos a través del archivo de configuración del sistema `ini.config` con las variables:

- `logAction`: controla el registro de información del módulo de acciones y la implementación de actuadores de bajo nivel.
- `logVision`: controla el registro de información del módulo de visión.
- `logComm`: controla el registro de información del módulo de comunicación.
- `logRules`: controla el registro de información del módulo manejador de reglas.

Las variables `logEnabled` y `logfile` indican si está habilitado o no la generación del archivo de log, y en qué archivo se escribirá (indicando la ruta completa).

6. Casos de estudio

Los casos de estudio pretenden implementar ciclos de pruebas del sistema en forma integral, abarcando todas las capas de la solución y generando un resultado tangible, observable a simple vista, y que cubre las necesidades prácticas para el desarrollo de la actividad de fútbol de robots.

Se trabajó sobre cinco casos de estudio en particular para poner a prueba la implementación realizada. Dos casos involucrando un solo robot realizando una tarea de forma individual y tres casos que involucran la comunicación entre robots para realizar una tarea de forma colaborativa.

En la sección *"Media"* del sitio web de FutRob[8] se encuentran disponibles videos de los ensayos de laboratorio realizados.

6.1. Individuales

Los casos de estudio que involucran solo un robot pretenden poner a prueba algunas acciones relacionadas con la coordinación entre la información obtenida por la visión y el movimiento del robot. Los casos a estudiar son: el patear un penal estando frente a un arco con arquero para evaluar la decisión sobre la dirección a patear y el atajar un penal teniendo en cuenta la trayectoria de la pelota y anticipar precisamente dónde cruzaría la línea de meta.

6.1.1. Patear un penal

Valiéndose de las capacidades de visión para percibir los objetos del entorno, para el robot es posible determinar si se encuentra frente al arco rival. Además, es posible saber cuándo hay otros robots (de cualquier equipo) en el campo de visión del robot. La solución implementada se basa en esta información para dotar a los robots de inteligencia a la hora de patear al arco.

Cuando un robot realiza la acción de patear la pelota con dirección al arco, primero determina si hay otros robots entre el arco y su posición que puedan interferir con el tiro. Si detecta que hay otro robot, por ejemplo: el arquero rival, determina cuál es la mejor dirección para realizar el tiro. Salvo que el otro robot se encuentre exactamente en el medio del arco, siempre existe un palo del arco del cual el arquero se encuentra más alejado. El robot elegirá entonces patear en esa dirección.

En primera instancia, para patear al arco el robot debe posicionarse ante la pelota a una distancia y posición tal que le permita patearla. Es decir, se coloca de frente al arco, detrás de la pelota, y con la pelota delante de

[8]Sitio web FutRob: `https://www.fing.edu.uy/~pgfutrob/`

alguno de sus pies. Luego de alcanzada esta posición de pateo, el robot está en condiciones de determinar la mejor dirección para patear. Para esto, el robot debe tener identificada la posición del arco y obstáculos, lo que le permite determinar la dirección de remate. Luego que el robot calcula ese dato, ajusta levemente su posicionamiento nuevamente mediante combinaciones alternantes de pequeños giros y pasos laterales para colocarse sobre la recta que determina la dirección elegida, y quedar enfrentado al arco y pelota en esa dirección. Finalmente patea con la pierna que tenga la pelota al alcance. Existen implementadas las acciones para patear tanto con la pierna derecha como con la izquierda.

Para realizar pruebas que evidencien este comportamiento se definieron los siguientes escenarios, con el robot tirador enfrentado al arco:

1. Robot arquero posicionado sobre el centro del arco.
2. Robot arquero posicionado sobre el sector izquierdo del arco.
3. Robot arquero posicionado sobre el sector derecho del arco.

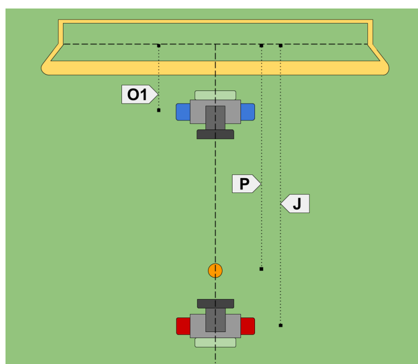

Figura 33: Diagrama de escenario patear 1

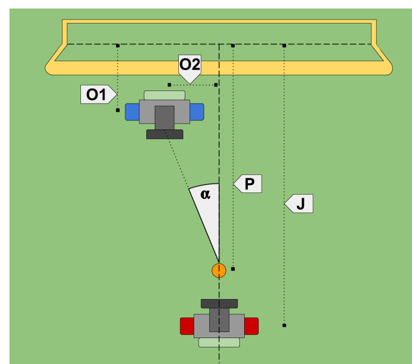

Figura 34: Diagrama de escenario patear 2

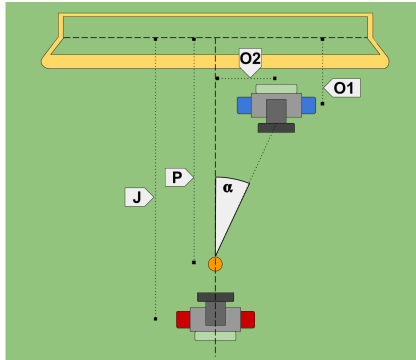

Figura 35: Diagrama de escenario patear 3

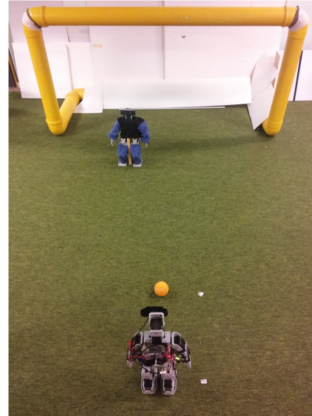

Figura 36: Imagen de escenario patear

Cuadro 3: Datos de escenarios de patear

Escenario	J	P	O_1	O_2	α
1	180 cm	140 cm	30 cm	0 cm	0°
2	180 cm	140 cm	30 cm	20 cm	−10,30°
3	180 cm	140 cm	30 cm	20 cm	10,30°

Durante la ejecución de las pruebas, el dato que se registra respecto de las posiciones observadas por el robot tirador del arquero rival y del centro del arco es el ángulo relativo a su posición. Por lo tanto en la descripción del escenario se incluye el cálculo del ángulo α en cada caso, calculado aplicando trigonometría según la ecuación 15.

$$\tan\alpha = \frac{O_2}{P - O_1} \rightarrow \alpha = \arctan\left(\frac{O_2}{P - O_1}\right) \tag{15}$$

Para completar la descripción del escenario de pruebas, se incluye en el fragmento 16 el archivo de reglas utilizado para modelar el comportamiento del robot tirador.

Fragmento 16: Reglas para patear al arco

```
1 <?xml version="1.0" encoding="US-ASCII" ?>
2
3 <RuleSet>
4  <Rule name="Look for ball" desc="">
5   <def>
6    <or>
7     <not>
```

```
8	    <value term="ballLastPosValid"/>
9	   </not>
10	   <and>
11	    <not>
12	     <value term="ballInSight"/>
13	    </not>
14	    <not>
15	     <value term="ballIsMine"/>
16	    </not>
17	   </and>
18	  </or>
19	 </def>
20	 <action name="findBall"/>
21	</Rule>
22	<Rule name="Go to ball" desc="">
23	 <def>
24	  <and>
25	   <not>
26	    <value term="ballIsMine"/>
27	   </not>
28	   <value term="ballInSight"/>
29	  </and>
30	 </def>
31	 <action name="followBall"/>
32	</Rule>
33	<Rule name="Look for goal" desc="">
34	 <def>
35	  <not>
36	   <value term="oppGoalLastPosValid"/>
37	  </not>
38	 </def>
39	 <action name="findOppGoal"/>
40	</Rule>
41	<Rule name="Look for opponent" desc="">
42	 <def>
43	  <not>
44	   <value term="opponentLastPosValid"/>
45	  </not>
46	 </def>
47	 <action name="findOpponent"/>
48	</Rule>
```

```
49  <Rule name="Kick" desc="">
50   <def>
51    <value term="goalInRange"/>
52   </def>
53   <action name="kickBall"/>
54  </Rule>
55 </RuleSet>
```

Para la ejecución de las acciones de tiro, el sistema utiliza una configuración específica dada por las siguientes constantes incluidas en el archivo `Constants.h`:

- `THRESHOLD_DIFF_ANGLE_OPP_GK_GOAL`: tolerancia en grados de distancia del arquero rival respecto del centro del arco rival. Si el arquero está a una distancia del centro del arco mayor a la indicada por esta constante, entonces el pateador asume que el arco está descubierto en el centro y decide patear al centro del arco.
- `THRESHOLD_ANGLE_KICK`: indica el tamaño del paso de giro que da el robot en caso que determine girar para ubicarse para patear.

Para la ejecución de las pruebas, los valores utilizados fueron los siguientes:

- `THRESHOLD_DIFF_ANGLE_OPP_GK_GOAL`: 10.
- `THRESHOLD_ANGLE_KICK`: 15

Por lo tanto, durante la ejecución de las pruebas, el robot decidió girar su posición de pateo únicamente en aquellos casos en que, alcanzada la pelota, la diferencia entre el ángulo del arco rival y el arquero rival es menor a 10°; es decir, el centro del arco está cubierto y se debe patear hacia un palo. En estos casos, aplicó un correctivo de 15° en la dirección que le resulta más conveniente.

Las tablas 4, 5 y 6 que se incluyen a continuación registran los casos de prueba según los escenarios 1, 2 y 3 respectivamente según fueron descritos anteriormente. La columna *¿Cayó?* registra si el robot cayó o no, y en caso afirmativo, en la ejecución de qué acción sucedió la caída. El resultado final de la acción, es decir, si pateó y en qué dirección, queda registrado en la columna *Dir. tiro*. La columna *Giros* indica si el robot giró o no, y en caso afirmativo el sentido (derecha es un ángulo positivo, apuntando para patear hacia el palo izquierdo del arquero, e izquierda es el opuesto). Los videos de esats pruebas están disponibles en la sección *"Caso de uso: Patear"* en el apartado *"Media"* del sitio web de FutRob.

Cuadro 4: Pruebas con arquero centrado

Prueba	Ángulo arco	Ángulo arquero	Ángulo tiro	¿Cayó?	Giros	Dir. tiro
1	-2.33	-4.52	12.67	No	der (1)	Arco
2	28.07	-5.2	28.07	No	der (1)	Arco
3	3.23	-0.13	-11.77	Patear	0	Arquero
4	18.3	-0.21	18.3	No	der (1)	Arco
5	23.93	-9.07	23.93	Pateando	der (1)	Afuera
6	1.96	5.62	-13.04	Caminar	-	No tiró
7	-17.06	-21.61	-2.06	Caminar	-	No tiró
8	20.91	-14.12	20.91	No	der (1)	Movió pelota
9	7.67	-25.41	7.67	Patear	0	No tiró
10	-5.47	-4.15	9.53	No	0	Arco

Cuadro 5: Pruebas con arquero hacia la izquierda

Prueba	Ángulo arco	Ángulo arquero	Ángulo tiro	¿Cayó?	Giros	Dir. tiro
1	8.88	21.99	8.88	No	0	Arco
2	-2.11	10.16	-2.11	No	0	Arco
3	-2.16	11.48	-2.16	No	0	Arco
4	3.19	13.8	3.19	No	0	Arco
5	15.2	18.63	0.2	No	0	Arco
6	-	-	-	Caminar	-	No tiró
7	19.88	2.54	19.88	Patear	der (1)	Arquero
8	-3.49	11.32	-3.49	Patear	0	Arco
9	9.29	24.33	9.29	Patear	0	Arco
10	3.77	15.46	3.77	Paso izq	0	No tiró

Cuadro 6: Pruebas con arquero hacia la derecha

Prueba	**Ángulo arco**	**Ángulo arquero**	**Ángulo tiro**	**¿Cayó?**	**Giros**	**Dir. tiro**
1	24.23	-18.37	24.23	No	der (1)	Erró pelota
2	11.93	-34.44	11.93	No	der (1)	Movió pelota
3	-10.44	-23.66	-10.44	No	0	Arco
4	22.64	-20.02	22.64	No	der (1)	Arco
5	-6.02	-38.06	-6.02	Patear	0	Arco
6	7.15	-33.31	7.15	Patear	0	No tiró
7	7.39	-2.04	22.39	Patear	der (1)	Afuera
8	-9.62	-50.2	-9.62	Patear	der (1)	No tiró
9	8.55	-0.2	23.44	Patear	der (1)	Afuera
10	2.99	-9.01	2.99	No	0	Arco

Cabe destacar que el caso planteado resulta exhaustivo considerando los aspectos de lógica y capacidades de visión, y de gran exigencia motora para la ejecución de acciones. En un caso típico el robot completa un ciclo que puede describirse en los siguientes pasos:

1. Buscar la pelota.
2. Caminar hasta la pelota con pasos normales.
3. Posicionarse delante de la pelota con pasos pequeños hasta tener posesión de la misma.
4. Buscar el arco rival.
5. Buscar el arquero rival.
6. Determinar dirección de tiro en función de la posición relativa del arquero rival respecto del centro del arco.
7. Posicionarse para patear en la dirección determinada. Puede implicar pasos laterales y giros sobre su eje.
8. Patear la pelota.

Como resultado de las pruebas se observó que el sistema de visión presenta dificultades en la identificación del objeto *arco rival*, con una marcada tendencia a ubicarlo más hacia la derecha (ángulos positivos) de lo que realmente se encuentra. A su vez, el arco es detectado únicamente a distancias mayores a $130cm$, lo que condicionó el armado del escenario colocando la

pelota a 140cm del arco para que el robot, al alcanzar esa posición, lograra capturar a través del sistema de visión la presencia del arco.

Respecto a la acción de patear específicamente, se observa que la pelota se desplaza una distancia máxima del orden de 110cm, alcanzando la línea del arquero en los escenarios planteados. Esto, en combinación con la condición de no detección del arco a menos de 130cm dificulta las posibilidades de completar un gol. El robot debería, o bien tener la capacidad de patear imprimiendo mayor fuerza sobre la pelota, o contar con un sistema de visión que le permita un rango más amplio de captura; por ejemplo un lente de tipo *gran angular*.

En relación a ejecución de las acciones en general, se observaron grandes cambios inducidos por las características del hardware: el nivel de carga de la batería, y una medida del *estrés* al que fue sometido cada actuador. Al reemplazar una batería con poco nivel de carga por una completamente cargada, los cambios en la ejecución de las acciones por parte de los motores resultó notoria a simple vista. Por otra parte, durante largas sesiones de pruebas ensayando estos escenarios, se observa una sobrecarga sobre los motores de la rodilla y cadera de la pierna de apoyo para patear; en ocasiones originando la caída del robot. Incluso durante estas pruebas se generó un daño mecánico sobre un motor de la cintura izquierda que debió ser reemplazado.

Como conclusión general, se observó que el robot tuvo un comportamiento acorde al modelado por las acciones y su inteligencia artificial, pero en ocasiones se vio limitado por las capacidades motoras que impidieron completar la acción con la presición de los cálculos realizados o evitando caídas.

6.1.2. Atajar un penal

Parte de lo implementado sobre el módulo de visión es la capacidad de detectar la trayectoria que describe el movimiento de la pelota. Para lograr dicho objetivo, en el modelo del mundo se guarda un historial de todas las posiciones de la pelota detectadas por la cámara. Con la posición actual, sumada a las posiciones anteriormente detectadas, es posible generar un modelo de la trayectoria como de explica en la sección 5.1.3. Esta información tiene varios usos. El beneficio inmediato que se desprende del conocimiento de que la pelota sigue una trayectoria rectilínea con determinada velocidad y aceleración es el hecho de poder anticiparse a la misma tanto en tiempo como ubicación. Ya sea para los jugadores que están en cancha y pretenden interceptar la pelota o patearla, como para el robot arquero en el momento de atajar la pelota.

En el caso de los robots que juegan en cancha, conocer la trayectoria, velocidad y aceleración de la pelota permitirá ir a buscarla de forma más precisa.

Si el robot no contara con dicha información y tuviera que acudir a buscar una pelota que está en movimiento, estaría constantemente corrigiendo sus pasos para tratar de alcanzar la pelota, puesto que esta cambia de posición constantemente entre cada paso. Por otro lado, si el robot es consciente que la pelota se está moviendo y con determinada dirección, velocidad y aceleración, puede calcular el lugar en el que va a estar la pelota en el futuro. Esto le da la posibilidad de moverse hacia donde va a estar la pelota y calcular la trayectoria óptima de su propio movimiento para que se cruce con la de la pelota en el menor tiempo y la menor distancia posible. Esto es posible ya el el robot conoce la velocidad a la que camina.

Para el robot arquero, la información sobre la trayectoria de la pelota le permitirá realizar una mejor maniobra para intentar atajarla. A partir de ver la pelota, el robot arquero es capaz de comenzar a calcular su trayectoria para poder determinar en primera instancia si la pelota está moviéndose con dirección al arco o no. En caso que detecte que la pelota se mueve hacia el arco, el robot procederá a calcular el momento en que la misma cruzará la línea de gol y el lugar exacto donde la pelota cruzaría la línea de meta en tanto no existan fuerzas externas ajenas al rozamiento. Finalmente, el robot arquero utiliza esta información para posicionarse en un lugar que le permita interceptar la trayectoria de la pelota cuando ésta cruce la línea de meta, lo que le permitirá atajarla.

El arquero dispone de dos movimientos para atajar la pelota. Uno en el que se agacha hacia la derecha y estira el brazo derecho para ocupar más espacio en esa dirección y otro simétrico hacia la izquierda. El tiempo que el robot pertenece en esta posición es variable. Cuando el robot detecta que la pelota se dirige al arco lo primero que hace es calcular si cruzará la línea de gol a su izquierda o a su derecha. En caso que detecte que la pelota está yendo hacia su posición actual, no será necesario que realice ninguna acción para atajarla ya que la misma se va a chocar con su cuerpo. Si el resultado de los cálculos arroja que la pelota pasará a un lado, el robot calculará la distancia a la que se encuentra de ese punto sobre el eje que une ambos palos. Como el alcance del movimiento de atajar es limitado y el arco es grande, es posible que el robot tenga que moverse lateralmente para alcanzar a la pelota cuando se incline al realizar la acción de atajar. El robot utilizará la distancia calculada para saber cuántos pasos laterales debe realizar previo a ejecutar la acción de atajar para conectar el balón.

Con el objetivo de probar lo implementado para el cálculo de la trayectoria de la pelota se realizaron pruebas donde el robot arquero debe atajar un penal. Se tuvieron en cuenta los tres escenarios más representativos que pueden ocurrir. En la primera prueba la pelota se dirige directamente hacia el robot arquero (figura 37). En este caso el robot debería permanecer en su

posición y no realizar ninguna maniobra.

En la segunda prueba, la pelota se dirige hacia un lado del arquero con una trayectoria perpendicular a la línea de meta (figura 38). En este caso, desde que el robot comienza a ver la pelota hasta que cruza la línea de gol, la misma se encuentra siempre a su izquierda o siempre a su derecha. Para atajar, el arquero deberá calcular la distancia entre su posición y el punto donde la pelota ingresaría al arco y determinar si corresponde moverse lateralmente hacia ese lado para llegar a la pelota. Cuando la pelota está a una distancia cercana al arco, el robot debe realizar la maniobra de atajar para así ocupar más espacio y aumentar las posibilidades de atajar la pelota.

Finalmente, la prueba más compleja es cuando la trayectoria de la pelota describe una diagonal que atraviesa el campo visual del robot de un lado hacia el otro. En esta prueba el robot comienza a ver la pelota a su izquierda pero luego la pelota termina cruzando la línea de gol a su derecha (figura 39). Sin el conocimiento de la trayectoria de la pelota, el robot al ver la pelota a su izquierda se movería hacia ese lado asumiendo que la pelota va a cruzar la línea de gol a su izquierda. En ese caso el robot estaría alejándose de la posición óptima para atajar. Cuando la pelota se aproxima suficientemente al arco, el robot deberá corregir su posición con la desventaja de haberse movido previamente de forma errónea en su primera reacción. Con la implementación realizada, que modela la trayectoria de la pelota, el robot sabrá desde un principio que deberá moverse hacia su derecha, a pesar de estar viendo la pelota hacia su izquierda.

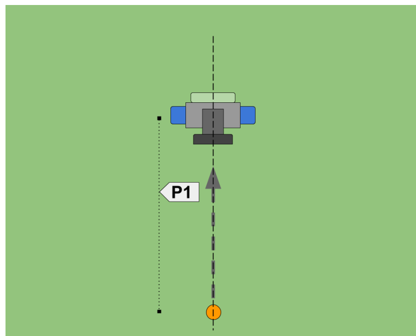

Figura 37: Diagrama atajar 1

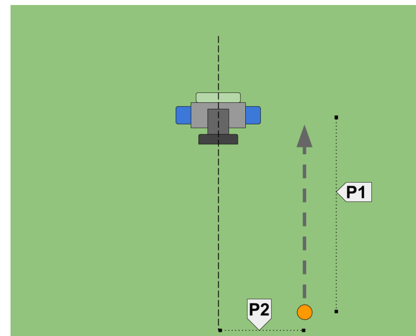

Figura 38: Diagrama atajar 2

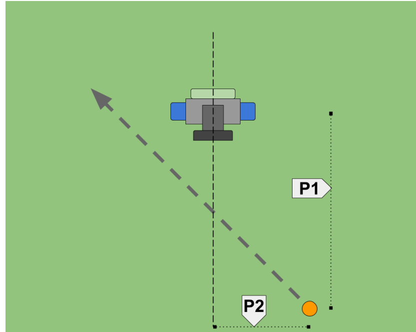

Figura 39: Diagrama atajar 3

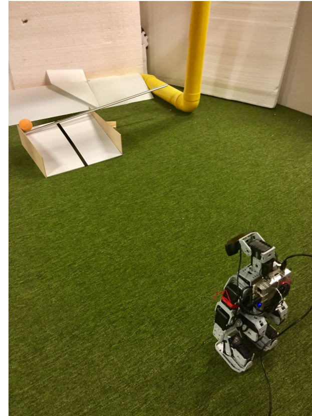

Figura 40: Imagen atajar

Las pruebas se ejecutaron varias veces cada una para solidificar la base empírica de conocimiento. La pelota se lanzó utilizando una rampa para que la velocidad y aceleración sean semejantes en todas las pruebas. En el cuadro 7 se detallan las distancias para cada escenario de pruebas. El conjunto de reglas usado en todos los escenarios es el definido en el fragmento 17

Cuadro 7: Datos de escenarios de atajar un penal

Escenario	A	P_1	P_2
1	30 cm	160 cm	0 cm
2	30 cm	160 cm	20 cm
3	30 cm	160 cm	20 cm

Fragmento 17: Regla para atajar

```
1  <?xml version="1.0" encoding="US-ASCII" ?>
2
3  <RuleSet>
4   <Rule name="Look for ball (turn head)" desc="">
5    <def>
6     <not> <value term="ballInSight"/> </not>
7    </def>
8    <action name="findBall"/>
9   </Rule>
10  <Rule name="Save ball" desc="">
11   <def>
12    <value term="ballIsGettingCloser"/>
13   </def>
```

```
14    <action name="save"/>
15   </Rule>
16   <Rule name="Look at ball (turn head)" desc="">
17    <def>
18     <value term="ballInSight"/>
19    </def>
20    <action name="lookAtBall"/>
21   </Rule>
22  </RuleSet>
```

Durante la ejecución de las pruebas, se registraron las características del comportamiento demostrado por el robot *arquero* según los escenarios 1, 2 y 3 definidos en los cuadros 8, 10 y 10 respectivamente. La rampa utilizada permite pequeñas variaciones en la trayectoria de la pelota en sucesivos lanzamientos, lo que posibilita evaluar el comportamiento del robot *arquero* ante situaciones más semejantes a un ensayo de juego real. Dada esta situación, se incluye en cada caso un campo adicional con notas producto de la observación de resultados de cada prueba.

El material de videos de estas pruebas está disponible en el sitio web de FutRob en la sección *“Caso de uso: Atajar”* dentro de la sección *“Media”*.

Cuadro 8: Pruebas de atajada en escenario 1

Prueba	**¿Cayó?**	**¿Caminó?**	**¿Inclinó?**	**¿Atajó?**	**Notas**
1	No	No	der	Si	Atajada derecha
2	No	No	No	Si	Atajada parado
3	No	No	der	No	Gol entre las piernas
4	No	No	izq	Si	Atajada izquierda
5	Si	izq (2)	No	No	Caída en paso izq

Cuadro 9: Pruebas de atajada en escenario 2

Prueba	¿Cayó?	¿Caminó?	¿Inclinó?	¿Atajó?	Notas
1	No	No	izq	No	Gol entre pierna-brazo
2	No	No	izq	No	Gol entre pierna-brazo
3	No	No	No	Si	Pelota fue al centro
4	No	No	izq	Si	Atajada izquierda
5	No	No	izq	Si	Atajada izquierda

Cuadro 10: Pruebas de atajada en escenario 3

Prueba	¿Cayó?	¿Caminó?	¿Inclinó?	¿Atajó?	Notas
1	No	No	der	Si	Atajada derecha
2	Si	No	der	No	Gol hacia derecha
3	No	der (1)	der	No	Gol hacia derecha
4	No	No	der	Si	Atajada derecha
5	No	No	der	Si	Atajada derecha

Como resultado de las pruebas, se observa especialmente cómo el robot nunca tomó la decisión de inclinarse en sentido opuesto a la intersección de la trayectoria de la pelota con su línea. Es decir, cuando la pelota cruzó a su izquierda en el escenario 2, el robot o bien se mantuvo parado o se inclinó hacia la izquierda. Análogamente para el caso 3 en que la pelota cruza la línea del *arquero* hacia su derecha, el robot o bien se inclinó hacia la derecha o se mantuvo parado. Luego, en el caso en que la pelota viene de frente al *arquero*, se dieron las 3 opciones posibles: inclinación hacia la derecha e izquierda y mantenerse parado.

A su vez, en cada situación que ameritó una inclinación lateral para atajar, el robot lo hizo a tiempo. En ocasiones no atajó la pelota porque ésta pasó entre sus extremidades, pero con el robot ya posicionado para atajar.

Es decir que el modelo de cálculo de la trayectoria de la pelota interpolando las posiciones reconocidas por la visión utilizando mínimos cuadrados resulta acertado al menos para determinar dirección y sentido del movimiento de la pelota, así como una buena estimación de su velocidad y aceleración.

6.2. Colaborativos

Los casos de estudio colaborativos pretenden poner a prueba y evidenciar la colaboración de los robot para realizar tareas de forma cooperativa. Se plantea implementar tres casos: la comunicación inicial donde los robots se anuncian incorporándose al juego indicando su identificador y rol, la ayuda en el posicionamiento del robot arquero por parte de otros miembros del equipo y la realización de un pase entre dos robots. Los dos últimos casos involucran únicamente dos jugadores, pero podrían ser realizados por cualquier par de jugadores del equipo en caso que se den las circunstancias de juego que lo ameriten.

6.2.1. Descubrimiento, salutación y anuncio

Una de las características más evidentes en implementaciones no colaborativas de sistemas multi-robot es la falta de conciencia y conocimiento del resto de los miembros del sistema. Considerando un contexto en el que dos o más agentes tienen un objetivo común y son miembros de un mismo equipo, son escasas las alternativas que permiten a cada uno de los miembros tener conocimiento del resto de sus pares.

Las dos alternativas más tradicionales son: que el equipo tenga un comportamiento estático, predeterminado al momento de comenzar la ejecución; o la utilización de sensores propios de cada agente para obtener información acerca de sus compañeros. Para el primer escenario, basta con establecer en la configuración de cada miembro del equipo cuál será su posición en el esquema de coordenadas, y una serie de atributos representativos para cada uno del resto de los miembros. Por ejemplo: un identificador de cada miembro, su posición, su rol en el equipo, su altura, sus capacidades, sus sensores, entre otros. En este caso existe la barrera del dinamismo del sistema; es decir, este tipo de enfoque aplica únicamente en casos en que el sistema es estático o puramente determinista.

Para el segundo escenario, cada miembro del equipo se vale de la información que percibe a través de sus sensores, típicamente una cámara web, para adquirir conocimiento e información del resto de los miembros del equipo. De esta forma, un miembro podrá advertir la existencia de otros dados una serie de rasgos distintivos que le permiten identificarlo como compañero de equipo.

Para el caso de los robots, los rasgos distintivos son su forma (el hecho de *ser un robot*) y el color de la vestimenta que lleva puesta. A partir de los sensores podrá determinar no solamente la presencia de compañeros y algunos de sus atributos como altura o rol, sino que también podrá evaluar constantemente su posición y los movimientos que realice. Las limitaciones en este escenario pasan por la complejidad que se agrega al procesamiento de la información sensorial y los posibles errores de percepción, y que la información acerca de los otros robots será obtenida únicamente durante el tiempo que cada robot permanece bajo el radio de percepción de los sensores. Es decir, considerando como sensor una cámara web, se tiene información de los otros miembros únicamente cuando se encuentran dentro del campo visual del robot.

Apoyados en la implementación del protocolo de comunicación entre robots a través de mensajería sobre la red inalámbrica, fue posible establecer un mecanismo de descubrimiento y anuncio interno al equipo en el que cada miembro tiene pleno conocimiento de sus compañeros durante el desarrollo del juego, incluyendo algunos de sus atributos más significativos.

Cuando un robot comienza la ejecución de *FutRob*, envía un anuncio de tipo *broadcast* hacia todos los miembros, a quienes a priori desconoce, indicando su identificador y su rol en el juego. Por ejemplo: identificador 3 y rol "jugador de campo". De esta forma, el resto de los miembros del equipo toman conocimiento que ha ingresado el jugador de campo número 3. Con el desarrollo del juego, el robot número 3 comenzará a recibir mensajes del resto de los robots del equipo. Dado que este robot desconoce al resto de los miembros del equipo, se percatará de la existencia de cada uno de sus compañeros por cada mensaje que reciba de un nuevo remitente. Esto provocará que genere una estructura en memoria para representar a este robot compañero. Inmediatamente, el robot 3 enviará a cada compañero que descubra un mensaje directo y de tipo `COMM_PARAM_ASK_ROLE` consultando por su rol.

Considerando que la mensajería fue implementada sobre el protocolo UDP que implementa una política de *mejor esfuerzo*, un robot podría no recibir el mensaje de anuncio de entrada al juego de algún robot. Consideremos al jugador 3, que ingresa al juego y envía los mensajes de anuncio a sus compañeros, y consideremos el robot 2 que no recibe estos mensajes. Luego, cuando el robot 2 reciba cualquier otro tipo de mensaje desde el robot 3, el robot 2 se percatará de la existencia del *nuevo* compañero y le enviará un mensaje de tipo `COMM_PARAM_ASK_ROLE` directo al robot 3 consultando por su rol.

El diagrama de la figura 41 ilustra el flujo de comunicación entre dos robots al momento de su conocimiento mutuo.

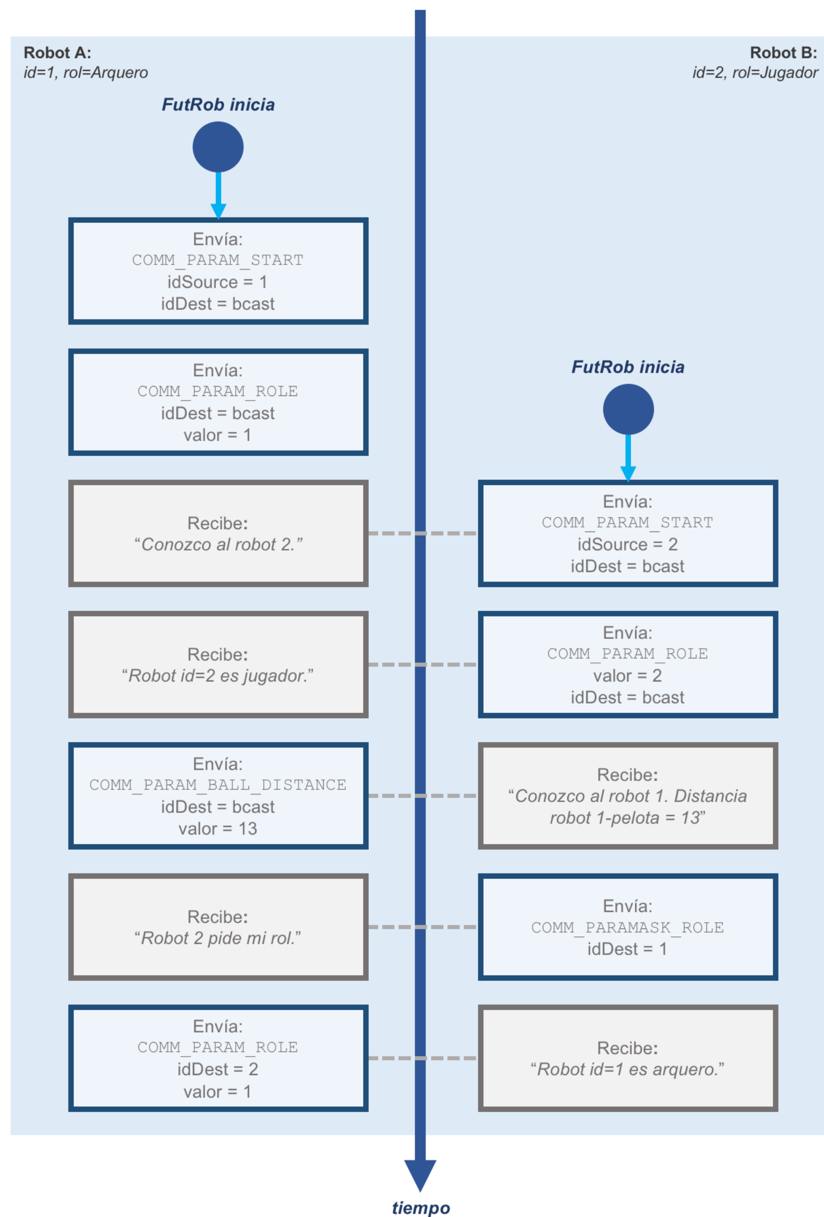

Figura 41: Flujo de comunicación de descubrimiento entre 2 robots.

Luego, y durante el desarrollo del juego, los robots compartirán la información ante las siguientes situaciones:

- *presencia de objetos representativos*: cuando un robot tiene algún objeto representativo del juego en su rango de visión como la pelota, un compañero, un rival, el arco, una marca, comunica qué objeto está viendo y a qué distancia,

- *ingreso o egreso de jugador*: cuando un robot ingresa al juego o se da de baja en forma controlada, el robot informa su identificador, su rol, y si ingresa o abandona el juego,
- *posesión de pelota*: cuando un robot tiene la pelota en sus pies, informa al resto del equipo,
- *acción ejecutada*: cuando un robot toma una nueva decisión acerca de la acción a ejecutar, informa al resto de sus compañeros.

De esta forma, los robots utilizan la comunicación para construir y mantener un modelo de la realidad en forma colaborativa. Cada dato de la realidad percibido por algún miembro del equipo, es comunicado y alimenta a la construcción del modelo global del entorno que mantiene cada robot.

6.2.2. Posicionamiento del robot arquero

Un problema común, tanto por su dificultad de resolución como por su probabilidad de ocurrencia, es el posicionamiento del robot arquero en el centro del arco. Dadas las capacidades de movimiento de los robots, particularmente los tiempos prolongados y la falta de precisión para ejecutar movimientos de giro sobre su eje, se optó por modelar el movimiento del arquero únicamente en una dimensión. El robot arquero se mueve únicamente sobre la línea de meta, línea que une ambos verticales que delimitan el arco. Es decir, el robot arquero no necesitará dar pasos al frente o hacia atrás, así como tampoco girar en ningún sentido. Por lo tanto, el movimiento del robot arquero en términos de acciones estará compuesto de pasos laterales -*galopas*- y movimientos de "atajadas" hacia los lados o hacia abajo.

Si bien diversas circunstancias de juego pueden motivar que el arquero cubra diferentes espacios del arco, es claro que en el caso general, cuando la pelota se encuentra fuera de la zona de influencia sobre el arco, el posicionamiento óptimo del arquero es en el centro del arco: sobre la línea de gol, equidistante de ambos verticales. Llamaremos a esta posición la "posición inicial" del arquero, y tomaremos la línea de meta como el eje de abscisas en la posición del arquero, siendo la posición inicial el punto de abscisa 0.

Para implementar un mecanismo colaborativo que ayude al arquero a posicionarse en el punto 0, idealmente se debería implementar una lógica que permita identificar los elementos: robot arquero, arco propio; y la posición relativa entre ellos. Como los jugadores del equipo utilizan la misma vestimenta, no hay un distintivo claro que permita diferenciar al robot arquero de un equipo por sobre los jugadores de campo. La lógica implementada

entonces realiza una abstracción del modelo y utiliza un umbral mínimo configurable para la posición relativa entre un jugador y el arco propio, objeto que sí puede identificarse unívocamente, para determinar si se trata del robot arquero o no. En simples palabras, si la visión captura en una misma escena a los objetos jugador compañero y arco propio a una distancia *cercana entre sí* (es decir, dentro del umbral), entonces asume que el jugador compañero identificado es un robot arquero.

Ante esta situación, y siempre que un robot de rol jugador observe al arco propio y un posible arquero compañero de equipo, el robot jugador envía un mensaje de tipo `COMM_PARAM_GK_POS_X` indicando en el campo *valor* del mensaje la posición relativa del robot arquero respecto del centro del arco en centímetros. Se toman las posiciones a la derecha del centro como posiciones positivas, y hacia la izquierda negativas. Luego, únicamente los robots con rol arquero procesarán este mensaje y afectarán su información relativa al posicionamiento.

El robot de rol arquero mantiene 2 variables de estado que le indican su posición respecto del centro del arco en cada dimensión, y que son ajustadas únicamente a través de la información recibida desde otros robots del equipo. El módulo de Modelo del Mundo deja disponible para utilizar por robots de rol arquero un predicado que indica si un arquero debe volver a posicionarse o no, comparando su desvío del centro contra un umbral para cada dimensión. A su vez, también ofrece una función que indica cuál es la posición relativa del robot arquero respecto del centro del arco, lo que permite implementar una acción llamada `Relocate` que hace que el arquero se posicione en el centro nuevamente. A su vez, para controlar cuándo esta acción debe ser ejecutada, se cuenta con el predicado `gkNeedsRelocate` Por lo tanto, esto desemboca en la construcción de una regla de comportamiento para el robot que contemple: si no veo la pelota o la veo pero no viene en dirección al arco o la pelota está en posesión del equipo propio, reviso si debo reubicarme. En caso afirmativo, aprovechar la instancia para moverse hacia el centro del arco.

Es importante destacar que, si bien el sistema fue implementado considerando el movimiento en una única dimensión para el robot arquero, tanto la mensajería como la visión y las estructuras de datos que mantienen la posición relativa del arquero respecto del centro del arco están preparadas para considerar el movimiento en las dos direcciones.

La implementación actual tiene una limitación y es que supone siempre que los ejes cartesianos de su sistema de visión son paralelos uno a uno con los ejes cartesianos de la posición del robot arquero, y el eje de abscisas del arquero es paralelo a la línea de meta. Para subsanar esta limitación, el sistema de visión debería devolver para el caso del arco no solamente la posición de su centro, sino que información que permita modelar la línea

de meta. Podría ser tanto otro punto cualquiera en la línea de meta, como los puntos de apoyo de los verticales, o directamente los coeficientes de la ecuación que modela la recta que une ambos postes. Esto permitiría resolver en el robot jugador la traslación de las coordenadas de visión propias hacia las coordenadas del sistema de visión del robot arquero, permitiendo que el robot jugador indique al robot arquero su posición exacta respecto del centro del arco. Actualmente este dato se envía al robot arquero bajo el sistema de coordenadas del robot jugador de campo. Si los ejes de abscisas se encuentran paralelos, y los ejes de ordenadas también, esta información se entrega en forma correcta.

Durante los ensayos de laboratorio que permitieron reproducir y evidenciar este comportamiento, se utilizaron 2 robots, un jugador y un arquero, según el diagrama de la figura 42.

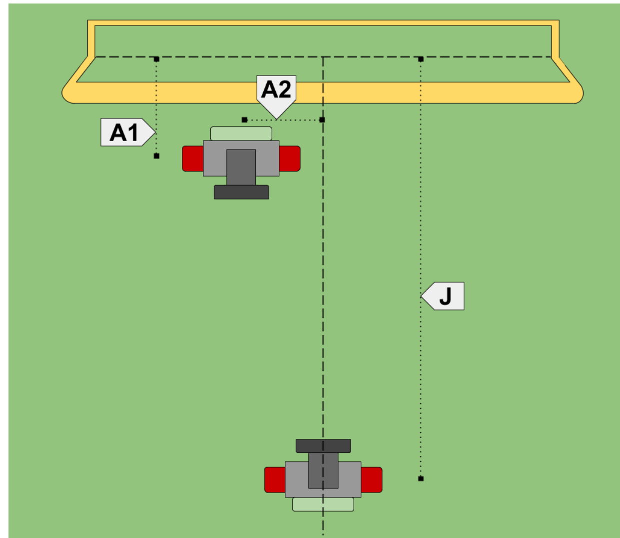

Figura 42: Diagrama de escenario de reposicionamiento de arquero.

Figura 43: Imagen de escenario de reposicionamiento de arquero.

En el cuadro se indica el detalle para las medidas fijas A_1 y J. El valor de A_2 varía según cada escenario y se registra independientemente para cada prueba, indicando el valor de posición inicial del caso y el valor de posición final luego que el robot ejecuta el movimiento.

Cuadro 11: Datos de escenario de reposicionamiento de arquero.

Escenario	J	A_1
1	180 cm	30 cm

Las reglas de funcionamiento del robot arquero abarcaron únicamente la valoración constante de la necesidad de reubicarse y, en consecuencia, la ejecución de la acción de alto nivel para reubicarse en el centro del arco. El archivo de reglas utilizado se incluye en el fragmento 18.

Fragmento 18: Regla para reposicionamiento

```
1  <?xml version="1.0" encoding="US-ASCII" ?>
```

```
2
3 <RuleSet>
4  <Rule name="Pruebas de reposicionamiento" desc="Lab">
5   <def>
6    <value term="gkNeedsRelocate"/>
7   </def>
8   <action name="relocate" />
9  </Rule>
10 </RuleSet>
```

Los umbrales de movimiento del robot se establecieron con la siguiente configuración para realizar las pruebas:

- GK_X_DIFF_THRESHOLD_MIN: 10.
- GK_X_DIFF_THRESHOLD_MAX: 40.
- GK_Y_DIFF_THRESHOLD_MIN: 40.
- GK_Y_DIFF_THRESHOLD_MAX: 80.

Se incluye a continuación en la tabla 12 los resultados de las pruebas realizadas donde se muestran las posiciones iniciales y finales del robot arquero, que corresponden a la referencia A_2 en el esquema de la figura 42. Algunos videos de estas pruebas se encuentran disponibles en el sitio web de FutRob, en la sección *"Caso de uso: Caso de uso: Reposicionamiento de arquero"* dentro de la sección *"Media"*.

Cuadro 12: Pruebas de reposicionamiento de arquero.

	Inicial	**Resultado ejecución**	
Prueba	X_i	X_f	**Se movió**
1	5	5	No
2	20	2	Si
3	30	4	Si
4	50	50	No
5	-5	-5	No
6	-20	3	Si
7	-30	1	Si
8	-50	-50	No

En todos los casos, el robot arquero se movió en el sentido esperado. Es decir, al estar posicionado sobre la izquierda del arco, recibió el mensaje del

robot jugador y se movió hacia la derecha. Análogamente para los casos en que comenzó posicionado sobre la derecha. Para los casos de desplazamiento de 5 cm, el robot permaneció en el lugar dado que la distancia se encuentra dentro del umbral mínimo de tolerancia (`GK_X_DIFF_THRESHOLD_MIN`).

6.2.3. Pase y recepción

Pensando en la implementación de comportamientos que evidencien la colaboración en un equipo de fútbol de robots, y con la intención de lograr un diferencial positivo de cara al cumplimiento del objetivo común de ganar el partido, un escenario recurrente es el de pase y recepción.

Es posible implementar un escenario acotado del pase considerando únicamente al robot que tiene posesión de la pelota. Las implementaciones no colaborativas toman información únicamente de este robot, complementando el dato de la posesión con la presencia en su rango de visión de un robot compañero. Contando con estos datos, el jugador que tiene la posesión puede intentar un tiro hacia el robot compañero. Al considerar un esquema colaborativo y apoyado en la comunicación entre robots, es posible hacer partícipe de esta situación de juego al robot receptor.

Llamémosle robot *pasador* al robot que tiene la posesión de la pelota. A partir de la evaluación del contexto actual del juego, tomando como fuentes de datos el modelo del mundo, la mensajería entre robots, la visión, y evaluando un conjunto de reglas definido, el robot *pasador* determina las siguientes condiciones: tiene la posesión de la pelota y el arco rival no está al alcance de su capacidad de tiro tanto sea por distancia o por no tenerlo a la vista. Además, el robot *pasador* determina que existe un robot compañero de equipo, llamémosle *receptor*, dentro de su rango de visión. A su vez, el robot *receptor* se declara en una posición favorable respecto del arco rival, es decir, que tiene el arco rival al alcance de tiro. El robot *receptor* da cuenta al resto de sus compañeros mediante el envío de un mensaje a través del protocolo de comunicación indicando que tiene el arco en la vista, con una trayectoria descubierta de obstáculos, y en una distancia menor a su capacidad de tiro.

Dadas estas condiciones de juego, el robot *pasador* inferirá de su conjunto de reglas que la mejor acción a ejecutar es la de intentar el pase a su compañero. Para esto, deberá hacer un tiro con el balón en su posesión y con destino hacia el robot *receptor*, cuya posición relativa a si mismo es conocida con precisión producto del resultado de la información del módulo de visión.

Por lo tanto, la acción de intentar el pase desde el punto de vista del *pasador* se reduce a, luego de saberse en posesión de la pelota, sin posibilidades de tirar al arco, y con un compañero mejor posicionado, hacer un tiro con destino a dicho compañero. Luego de tomada la decisión de ejecutar

esta acción, y mientras se encuentra ejecutándola, el robot *pasador* envía un mensaje indicando el detalle de su accionar.

Para el escenario de pruebas de esta situación de juego se desestimó la presencia del arco, considerando como factores decisivos en las reglas las condiciones: tener posesión del balón, y tener un jugador compañero dentro del alcance de tiro.

El comportamiento del robot *pasador*, descrito en la regla del fragmento 19, indica que buscará la pelota hasta encontrarla. Luego de tener la pelota a la vista, tomará posesión de la misma. Estando en posesión de la pelota, el robot recurrirá a la visión para buscar un robot compañero ejecutando la acción `findPartner`. Esta acción implica que el robot *pasador* quite la mirada de la pelota. Para controlar esta situación se utiliza el predicado *ballLastPosValid*, que indica que, si bien no se tiene la pelota actualmente dentro del rango de visión, la posición de la misma es conocida ya que al ser vista por última vez en el tiempo cercano, la pelota estaba quieta y el robot no cambió su posición. Es decir: la pelota estaba quieta, el robot *pasador* la tenía en posesión, y el robot *pasador* no cambió su posición. Finalmente, luego que encuentra un compañero al alcance de tiro, el robot pasador ejecuta la acción *passBall* para intentar un pase a su compañero.

A partir del momento en que el robot *pasador* toma la decisión de ejecutar un pase, el robot *receptor* tomará conocimiento que un compañero está intentando hacerle un pase mediante un mensaje de red enviado por el robot *pasador*. El robot *receptor*, al procesar este mensaje, y de acuerdo con su comportamiento descrito por la regla del fragmento 20, disparará automáticamente la lógica para tomar contacto visual con la pelota: la acción `findBall`. Inmediatamente intentará tomar posesión de la misma ejecutando la acción `followBall`. Consideramos por completado el comportamiento colaborativo de pase y recepción luego que el robot *receptor* logra tomar posesión de la pelota y mantenerla dentro de su campo visual ejecutando la acción `lookAtBall`.

En la figura 44 se representa la disposición de los robots involucrados en el escenario de pase.

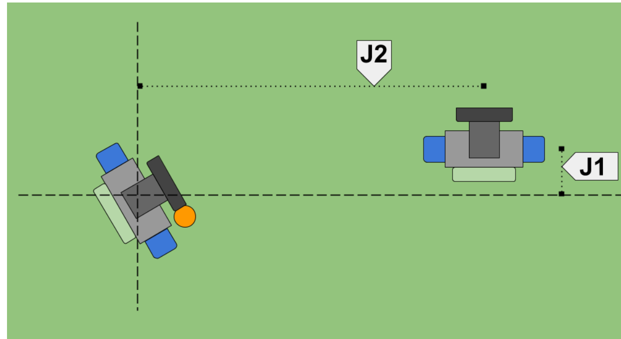

Figura 44: Esquema de escenario de pase.

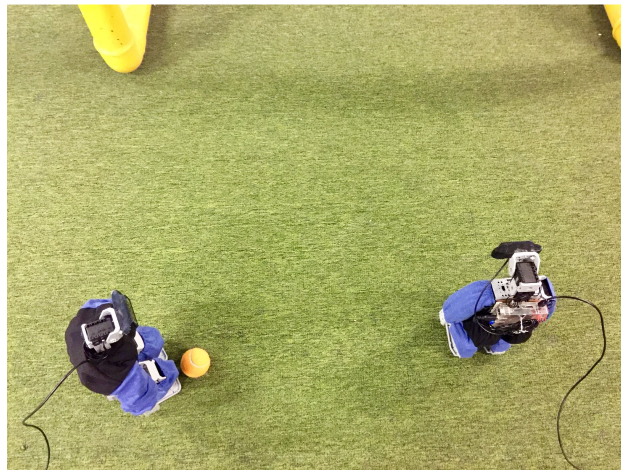

Figura 45: Imagen de escenario de pase.

Las distancias de detallan en la figura 13.

Cuadro 13: Datos de escenario de pase.

Escenario	$J1$	$J2$
1	10 cm	80 cm

Fragmento 19: Regla para la ejecución de pase

```
1 <?xml version="1.0" encoding="US-ASCII" ?>
2
3 <RuleSet>
4  <Rule name="Look for ball" desc="">
5   <def>
```

```
6      <or>
7        <not>
8          <value term="ballLastPosValid"/>
9        </not>
10       <and>
11         <not>
12           <value term="ballInSight"/>
13         </not>
14         <not>
15           <value term="ballIsMine"/>
16         </not>
17       </and>
18     </or>
19   </def>
20   <action name="findBall"/>
21 </Rule>
22 <Rule name="Go to ball" desc="">
23   <def>
24     <and>
25       <not>
26         <value term="ballIsMine"/>
27       </not>
28       <value term="ballInSight"/>
29     </and>
30   </def>
31   <action name="followBall"/>
32 </Rule>
33 <Rule name="Look for partner" desc="">
34   <def>
35     <not>
36       <value term="partnerLastPosValid"/>
37     </not>
38   </def>
39   <action name="findPartner"/>
40 </Rule>
41 <Rule name="Pass" desc="">
42   <def>
43     <value term="partnerInRange"/>
44   </def>
45   <action name="passBall"/>
46 </Rule>
```

```
47 </RuleSet>
```

Fragmento 20: Regla para la recepción de pase

```
1  <?xml version="1.0" encoding="US-ASCII" ?>
2
3  <RuleSet>
4   <Rule name="Look for ball after partner pass.">
5    <def>
6     <and>
7      <value term="partnerPass"/>
8      <not>
9       <value term="ballInSight"/>
10     </not>
11    </and>
12   </def>
13   <action name="findBall"/>
14  </Rule>
15  <Rule name="Catch the ball after partner pass.">
16   <def>
17    <and>
18     <not>
19      <value term="ballIsMine"/>
20     </not>
21     <value term="ballInSight"/>
22    </and>
23   </def>
24   <action name="followBall"/>
25  </Rule>
26  <Rule name="Stay in ball position (stare at ball)">
27   <def>
28    <value term="ballIsMine"/>
29   </def>
30   <action name="lookAtBall"/>
31  </Rule>
32 </RuleSet>
```

Durante la ejecución de las pruebas en el escenario descrito, se registraron los datos que se incluyen en la tabla 14. En el registro se distinguen los datos observados sobre cada uno de los robots: *pasador* y *receptor*.

En el sitio web de FutRob están disponibles videos de las pruebas realizadas, en la sección *"Caso de uso: Caso de uso: Pase y recepción"* dentro de

la sección *“Media”*.

Cuadro 14: Resultados de ensayos de pase y recepción

	Pasador	*Receptor*		
Prueba	**¿Pasó?**	**¿Buscó?**	**¿Encontró?**	**Pos. final**
1	Si (cayó)	Si	Si	Pelota
2	Si (cayó)	Si	Si	Pelota
3	Si	Si	Si	Pelota
4	Si	Si	No	Pos inicial
5	Si	Si	Si	Pelota
6	Si	Si	Si	Pelota (cayó)
7	Si (cayó)	Si	Si	Pelota
8	Si	Si	Si	Pelota
9	Si (cayó)	Si	Si	Pelota (cayó)
10	Si	Si	Si	Pelota

Como resultado de las pruebas realizadas, se observa que en todos los casos el robot *pasador* logró ejecutar la acción de pase y comunicar correspondientemente a sus compañeros. Sin embargo, en algunos casos luego de patear la pelota sufrió una caída. El robot *receptor* buscó la pelota en todos los casos, y logró encontrarla satisfactoriamente excepto en una ocasión en que el pase ejecutado fue demasiado corto. Finalmente, en los 9 casos en que encontró la pelota, el robot *receptor* logró hacerse de ella satisfactoriamente aunque se produjeron 2 caídas, ambas en el intento de ejecutar los movimientos finos de posicionamiento tras la pelota.

7. Conclusiones

7.1. Combinación de áreas de conocimiento

El equipo de trabajo abordó el proyecto con escasos conocimientos previos en el área de robótica. Más específicamente, ninguno de los dos integrantes había cursado asignaturas en el área de robótica. Naturalmente esta situación resultó en una subestimación tanto de alcance como de requisitos para el desarrollo del proyecto.

Las consideraciones previas al comienzo del proyecto se emparentaron principalmente con el desarrollo de una solución de software y la implementación de sistemas de inteligencia artificial. Es decir, la implementación de esquemas de lógica que modelaran el razonamiento y la toma de decisiones.

Fue durante la instancia de investigación, y trabajando sobre el primer hito del proyecto, la elaboración de un Estado del Arte en el área de robótica aplicada al juego de fútbol y robótica colaborativa, que se tomó dimensión acerca del fuerte sesgo matemático en la teoría de robótica.

Luego, y ya comenzando con el desarrollo de la solución, se tomó plena dimensión de otros componentes de relevancia medular, como las áreas de física (especialmente dinámica), y electrónica.

La rama de dinámica de la física por el comportamiento de los movimientos del robot y su constante interacción con un contexto cambiante y no determinista. Problemas de equilibrio, de obstáculos, e incluso cambios en condiciones de luz. Asimismo, comprensión del comportamiento de los objetos relevantes en el juego para la elaboración de un modelo que represente coherentemente la realidad. Por ejemplo, el movimiento de la pelota.

Finalmente, el componente de electrónica que viene dado por la construcción de un sistema con diferentes componentes de hardware que deben interactuar entre si y tomar la alimentación desde una fuente externa y común. Se debió trabajar sobre conexiones, disposición de componentes, cableado, e incluso el desarrollo completo de una placa de interfaz de hardware que permitiera la interacción entre un sistema semi-dúplex y un sistema dúplex completo.

Por lo tanto consideramos fuertemente recomendable para futuros equipos de trabajo en el área de robótica, valorar y considerar su experiencia en la rama.

7.2. Complejidad de ensayos experimentales

Al trabajar sobre un sistema que interactúa fuertemente con el contexto que lo rodea, e involucrar una serie de objetos importantes y distintivos en

el juego, cobra vital importancia el desarrollo de ensayos de laboratorio. Fue fundamental contar con un laboratorio a disposición ya que se necesita: un lugar amplio, condiciones de luz adecuadas, un sitio donde se disponga de lugares de almacenamiento para los objetos, corriente eléctrica, posibilidad de establecer una WiFi propia, estaciones de trabajo para computadoras personales, entre otras.

De todas formas, la correcta configuración del espacio de laboratorio para generar las condiciones apropiadas para el ensayo de un experimento en particular demandaron gran tiempo y coordinación con el equipo encargado del laboratorio. El tiempo principalmente debido a que se deben colocar los objetos de forma que sean útiles para el caso a probar, y especialmente la calibración de las condiciones de luz para que el módulo de visión logre interpretar correctamente los objetos participantes.

Estas situaciones dieron forma a una necesidad no identificada en etapas tempranas del proyecto, y que fue abordada a pesar de ampliar el alcance del trabajo propuesto: el desarrollo de un módulo de visión *stub*. Si bien se invirtió un tiempo importante, consideramos que el trabajo realizado para dotar al sistema de esta capacidad será de gran utilidad para trabajos futuros. Este sistema permite generar situaciones simuladas para la visión, estableciendo en forma determinista una serie de escenarios potencialmente infinito. Esto permite analizar el comportamiento del robot ante diferentes escenarios conocidos en forma previa.

7.3. Resultados condicionados por el hardware

Durante los ensayos de laboratorio se constataron fehacientemente las condicionantes del hardware sobre la performance del sistema. Esto dificultó el trabajo de refinamiento sobre la implementación de las acciones y movimientos de cada robot, ya que los motores demostraron tener notorios cambios de respuesta a lo largo de una instancia de pruebas.

Por un lado, el robot muestra grandes cambios en la ejecución de los movimientos cuando se alterna entre una batería con carga completa y una batería parcialmente cargada. A medida que se avanza sobre la ejecución de ensayos, la batería se descarga paulatinamente en forma continua. Esto impacta directamente sobre el rendimiento de los motores AX-12, y no fue posible adecuar el sistema a esta degradación gradual. Esto dificulta el ajuste fino de los movimientos del robot.

Por otra parte, el uso intensivo del sistema denota una severa demanda sobre los motores AX-12. Tanto los actuadores que ejecutan giros para llevar a cabo los movimientos, cuyo desgaste resulta natural, como aquellos motores que sostienen el peso del robot para mantenerlo en equilibrio. Parti-

cularmente, y trazando un paralelismo con los seres humanos, los actuadres que más sufren de esta última forma de desgaste son aquellos de la cintura y rodillas. Incluso los motores pueden llegar a sufrir daños eléctricos o mecánicos irreparables. Un claro síntoma de esta sobre exigencia es la temperatura que adquieren los actuadores, detectable al tacto. Producto de esta situación, idealmente se debería trabajar con más de un robot secuencialmente, es decir, utilizar un robot con una carga completa de batería y, luego de agotar la batería, utilizar otro robot y batería diferentes para permitir el "descanso" del robot anterior.

7.4. Mecanismos de diagnóstico del sistema

Durante el desarrollo del proyecto de software, se evidenció una gran dificultad a la hora de detectar problemas de funcionamiento de la solución. En otro tipo de sistemas, se cuenta con herramientas que permiten hacer un *debug* del código en forma relativamente sencilla. En cambio, en este sistema no fue viable. Esto debido a varias situaciones como la tecnología sobre la que se desarrolló (existen varios componentes en lenguaje C), a desarrollar en forma remota sobre el sistema, a que el sistema no cuenta con un display propio, a que gran parte de los insumos son no deterministas, y especialmente a la arquitectura desarrollada en 4 hilos de ejecución concurrentes.

Para paliar esta situación, se recurrió principalmente a desarrollar el código utilizando comentarios de texto mediante la salida estándar. Esto permitió tener una percepción en tiempo real de cuál es el comportamiento del sistema. Una parte significativa de estos recursos de mensajes se mantuvo en la implementación y se controlan en cada archivo habilitando la constante `DEBUG` y volviendo a compilar el proyecto.

Además, para futuros diagnósticos y considerando que el sistema debería ejecutar en forma autónoma, es decir, sin componentes externos con acceso a la salida estándar, se desarrolló un módulo de logs de sistema. El sistema permite mediante su archivo de configuración `ini.config` habilitar o deshabilitar esta capacidad. Cada registro en el log se incluye con información precisa para comprender la secuencia: una marca de tiempo (*timestamp*) con presición de milisegundos, el archivo donde se registró la línea, el método y la línea en el archivo fuente.

Estos dos recursos serán de gran utilidad para mantener y escalar sobre el sistema. El esfuerzo invertido en esta tarea, siendo una tarea satélite a la solución, entendemos que facilita mucho el diagnóstico del comportamiento de la plataforma en su conjunto.

8. Trabajo futuro

Durante el transcurso del proyecto se detectaron posibilidades de mejora para el sistema. Algunas de ellas no pudieron ser abordadas en el marco del proyecto puesto a su gran impacto en el alcance. Esta sección detalla las más relevantes como puntapié para su futuro desarrollo.

8.1. Incorporar un giroscopio

La sustitución de la placa original de los robots Bioloid (CM-510) por la placa Raspberry Pi 2 implicó, entre otras cosas, la pérdida del sensor giroscopio. La razón principal fue que la conexión del giroscopio original es analógica y las entradas de la Raspberry Pi son solo digitales. Esto hace que sea necesario agregar algún componente externo para realizar dicha conexión.

El hecho de no contar con un giroscopio hace muy difícil poder determinar cuando el robot se cae al piso. Para poder lograrlo con lo que cuenta actualmente el robot habría que usar la visión y poder identificar cuando la cámara está apuntando hacia el piso o hacia el techo, para concluir que el robot se cayó de frente o de espaldas respectivamente.

Ya que el robot suele caerse muy a menudo, sería una mejora considerable para el sistema poder saber con certeza cuando el robot se cayó para que tenga la capacidad de levantarse solo. Las acciones de levantase ya están implementadas, por lo que solo es necesario implementar la detección de caídas para darle al robot dicha funcionalidad.

Para lograrlo una posibilidad es incorporar el giroscopio original de Bioloid con la ayuda de algún componente para convertir la salida de analógica a digital y que pueda ser usado por la Raspberry Pi. Otra opción es agregar otro giroscopio con una interfaz digital para que pueda ser usado sin tener que agregar nada extra.

8.2. Mejoras de precisión en el módulo de visión

Actualmente el módulo de visión del robot realiza el procesamiento de imágenes capturando siempre a una resolución fija de la cámara web. Esto tiene ciertas limitaciones con respecto a la distancia a la que son reconocidos los objetos. Durante el transcurso del proyecto, se realizaron pruebas con varias resoluciones y se observó que es posible detectar objetos más lejanos si se aumenta la resolución. Una desventaja es que el tiempo de procesamiento de las imágenes aumenta notoriamente.

Para que el robot sea capaz de detectar objetos a distancias más lejas, y objetos cercanos con mejor precisión, sin perder velocidad de procesamiento,

se propone realizar capturas de la cámara a distintas resoluciones. La mayoría de las capturas deberían ser de una resolución suficientemente chica para que el procesamiento de la imagen no tome más de una fracción de segundo. Sin embargo, cada cierto tiempo, se puede realizar una captura a mayor resolución para que el robot vea e identifique los objetos que tiene más lejos. La información de estas capturas deberá ser tratada de manera especial para que no concluir que los objetos detectados dejan de ser visibles cuando se toma capturas con la resolución menor. También hay que tener en cuenta que es necesario realizar modificaciones al tamaño mínimo y máximo de *blob* que se quiere considerar cuando se cambia la resolución.

Otra opción puede ser que éstas capturas sean a pedido, es decir, cuando el módulo de visión no detecta un objeto con la resolución menor. Por ejemplo, cuando el robot busca la pelota, o cuando el robot tiene la pelota y busca el arco para realizar un tiro.

Además de realizar capturas a distintas resoluciones otra posible mejora al sistema de visión es la capacidad de detectar más datos sobre los objetos. Se observó que la información que brinda el sistema con respecto a los arcos es muy limitada. Actualmente no es posible saber si los arcos se están viendo de frente o si se están viendo de costado. Esto hace que cuando un jugador quiere ayudar al arquero a posicionarse se puedan cometer errores por asumir que se está viendo el arco de frente cuando puede no ser el caso. Se propone que en caso de que los dos palos del arco sean visibles, se devuelva la información sobre la posición de ambos. De esta forma, sabiendo el largo del arco es posible determinar el ángulo con el que se está viendo el mismo.

8.3. Algoritmos para refinamiento de posiciones y movimientos

Gran esfuerzo se dedicó a la calibración de los juegos de posiciones de los motores para conformar la posición de equilibrio estático del robot. Este procedimiento se realizó en forma experimental, mediante ensayos controlados en el laboratorio, hasta lograr resultados satisfactorios.

Existen múltiples factores que influyen sobre la posición de equilibrio del robot, como: el desgaste de las piezas plásticas; el desgaste de los actuadores; y leves diferencias en distribución y peso de los componentes periféricos que hacen al robot como controladora, batería, cámara, entre otros. Esto hace que para cada robot se deba realizar el proceso de calibración en forma independiente, y que al sustituir alguna pieza o ensamblar nuevamente el robot, el procedimiento de calibración deba volver a ejecutarse.

Además de la calibración del estado de reposo del robot, esta necesidad

también se traslada a cada uno de los movimientos que implementa el robot, lo que se traduce en la calibración de cada una de las posiciones intermedias que modelan ese movimiento y las traslaciones para cada actuador. Realizar esta calibración en forma experimental en el laboratorio resulta un trabajo complejo y no se tienen garantías de lograr resultados óptimos.

Si bien el desarrollo realizado contempla la configuración independiente por robot de la posición de equilibrio permitiendo ajustar cada uno de los 18 motores en el archivo `Offsets.h`, se observó que no es suficiente para ejecutar el conjunto de movimientos en forma estable en cada robot. Esto provocó que durante el proyecto se trabajara con un robot único como base para la ejecución del conjunto completo de movimientos, y luego se incluyó un subdirectorio `tunedPoses` con archivos modificados específicamente para cada una de las plataformas de robot utilizadas.

Por lo tanto se plantea como trabajo futuro el diseño e implementación de técnicas de calibración automatizadas. Estas podrían estar basadas en metodologías de aprendizaje automático o algoritmos evolutivos, o alguna heurística adaptada para el caso. El enfoque sugerido considera la naturaleza del problema a resolver, entendiendo que se trata de un modelo no determinista.

8.4. Optimizar la discretización del espacio de movimientos

El robot implementa movimientos que condicionan el modelo de la realidad a un espacio típicamente discreto. Cada movimiento del robot se compone de un conjunto de posiciones de sus motores, y lo solemos denotar como una "acción de bajo nivel". Es natural extrapolar estas acciones hacia el modelo de la realidad que comprende el robot, entendiendo cada acción de bajo nivel como la unidad atómica que tiene el robot para moverse dentro del espacio físico que lo rodea. Este nivel de atomicidad se ve afectado por las características humanoides del robot utilizado, donde cada acción de bajo nivel debe partir de una situación de reposo en equilibrio y culminar en un nuevo estado de equilibrio.

Por lo tanto, cada acción de bajo nivel del robot induce una discretización del espacio físico sobre el que actúa. Por ejemplo, para el caso del giro sobre su eje, la acción de bajo nivel más elemental que ofrece el sistema hace que el robot gire 20° hacia su derecha o izquierda; para el caso de movimiento lateral, la acción más elemental —un paso lateral— implica un movimiento de 4cm hacia la derecha o izquierda; y para el caso de desplazamiento en línea recta hacia adelante, la acción más elemental —un paso— implica un

movimiento de 8cm hacia adelante.

Dadas las acciones implementadas, podemos afirmar que el robot discretiza el espacio de traslación en línea recta hacia adelante en intervalos de 8cm. Esto implica que si el robot tiene la pelota a 17cm, podrá avanzar 2 pasos y quedar a 1cm, posición que resulta razonable para tomar posesión de la misma y ejecutar un despeje, pase o tiro al arco. Por el contrario, considerando un escenario en que el robot tiene la pelota a 13cm, la situación cambia ya que el robot ejecutará un paso que lo dejará a 5cm de la pelota, y no tendrá forma de acercarse con mejor precisión a la posición de destino. En este caso, no logrará hacerse con la posesión de la pelota puesto que la posición de la misma no se encuentra comprendida dentro de su espacio discreto de movimientos.

Existen dos enfoques para la resolución de este tipo de problemática: modelar las capacidades de movimiento sobre un espacio continuo, o implementar un refinamiento del espacio discreto en subespacios discretos de menor amplitud. El primer enfoque suele ser practicable en escenarios donde el robot se mueve sobre ruedas o vías y se puede garantizar en forma constante su equilibrio. El segundo enfoque resulta más apropiado para el tipo de sistema sobre el que trabajamos, con movimientos que deben procurar garantizar la estabilidad y el equilibrio del robot constantemente.

Para dotar al sistema de mayor efectividad, se debería analizar el conjunto de acciones de bajo nivel para evaluar cuáles, al descomponerlas en acciones de mayor precisión, dotarían al sistema de las capacidades necesarias para obtener mejores resultados. Por ejemplo, para el caso del posicionamiento del robot ante la pelota para efectuar un disparo, el sistema podría contar con una acción de bajo nivel que implemente pasos cortos —por ejemplo de 1cm— además de la acción ya existente de movimiento más *grueso*. Por ejemplo, considerando un escenario donde la pelota se encuentra a 13,5cm del robot, éste daría un “paso normal” de 8cm que lo dejaría a 5,5cm de la pelota. Luego, podría ejecutar 5 “pasos cortos” que lo acerquen 5cm y lo dejen en una posición a 0,5cm de la pelota. Luego de completada esta secuencia, el robot estaría en posesión del balón para efectuar un disparo.

8.5. Evaluación de reglas utilizando lógica difusa

Como se aborda en la sección 5.1.1, el modelo desarrollado permite la configuración del comportamiento del sistema mediante la definición de un conjunto de reglas en un archivo XML, una estructura arborescente. El intérprete de reglas implementado puede considerar modelos de comportamiento con grandes números de reglas y es capaz de evaluar predicados complejos, con cualquier combinación y agrupación de los operadores lógicos AND, OR

y NOT. El motor implementa la evaluación de predicados de lógica bivalente, es decir: falso o verdadero.

El mecanismo para definir el comportamiento a ejecutar se obtiene como resultado de la evaluación en orden de las reglas definidas en el archivo XML. Es decir, las reglas son evaluadas en orden desde primera a última, la evaluación se detiene en la primer regla válida. Esta regla definirá la próxima acción a ejecutar. Por lo tanto, el sistema impone un orden de precedencia para la evaluación de las reglas, dado por el orden en que son listadas en el archivo XML.

Queda planteado para un abordaje futuro la implementación de un motor de evaluación de reglas que implemente lógica difusa (*fuzzy logic*) en lugar de lógica bivalente (*boolean logic*). Debería evaluar el en forma simultánea el conjunto completo de reglas lógicas y asociar a cada una un valor numérico de relevancia, para finalmente definir cuál regla ejecutar. Eventualmente podría considerar información adicional luego de evaluar las reglas, valorando entre las más convenientes cuáles son las menos costosas en términos de tiempo de ejecución, consumo de energía, consumo de recursos, entre otros.

Referencias

[1] R. Canales, S. Casella, and P. Rodríguez, "Forrest: Desarrollo de un equipo de fútbol de robocup," tech. rep., Facultad de Ingeniería, UdelaR, 2006. http://www.fing.edu.uy/inco/grupos/mina/pGrado/forrest/.

[2] "Ensamblado y utilización del sistema futrob." http://www.fing.edu.uy/~pgfutrob/, 2016.

[3] RoboCup, "Robocup organization," 1994. http://www.robocup.org/.

[4] P. M. Asada, "Laboratorio de minoru asada," 1992. http://www.er.ams.eng.osaka-u.ac.jp/.

[5] P. M. Veloso, "Prof. manuela veloso, carneige mellon university, pittsburg, pa," 2015. http://www.cs.cmu.edu/~mmv/.

[6] "Robocup humanoide, sitio oficial," 2016. https://www.robocuphumanoid.org/.

[7] "Reglamento robocup fútbol humanoide." http://www.robocuphumanoid.org/wp-content/uploads/HumanoidLeagueRules2014-07-05.pdf, 2014.

[8] S. Duarte and P. Green, "Futrob - estrategias cooperativas en robótica: Estado del arte," tech. rep., Facultad de Ingeniería, UdelaR, 2015.

[9] "Uva trilearn framework - soccer simulation team." https://staff.fnwi.uva.nl/a.visser/research/ias/trilearn/2002/, 2002.

[10] N. T. Gastón Fernández, Claudia Stocco, "Visión de robots," tech. rep., Facultad de Ingeniería, UdelaR, 2003. https://www.fing.edu.uy/inco/grupos/mina/pGrado/vision2003/.

[11] G. Gismero, "Visión robótica y reconstrucción espacial con aplicaciones prácticas," tech. rep., Facultad de Ingeniería, UdelaR, 2010. https://www.fing.edu.uy/inco/grupos/mina/pGrado/vision2010/.

[12] M. Baliero and G. Pias, "Salimoo: Fútbol de robots para la liga humanoide de robocup," tech. rep., Facultad de Ingeniería, UdelaR, 2011. http://www.fing.edu.uy/inco/grupos/mina/pGrado/salimoo/.

[13] R. Dearmas, N. Fúrquez, J. Pereira, and A. Ricca, "Futbot: Fútbol robótico humanoide," tech. rep., Facultad de Ingeniería, UdelaR, 2013. http://www.fing.edu.uy/inco/proyectos/futbot13/.

[14] RaspberryPi, "Raspberrypi," 2016. http://www.raspberrypi.org/.

[15] ROBOTIS, "Manual robotis cm-510." http://support.robotis.com/en/product/auxdevice/controller/cm510_manual.htm, 2010.

[16] A. Corporation, "Atmel atmega2561 datos técnicos." http://www.atmel.com/Images/Atmel-2549-8-bit-AVR-Microcontroller-ATmega640-1280-1281-2560-2561_Summary.pdf, 2014.

[17] ROBOTIS, "Manual dynamixel ax-12." http://support.robotis.com/en/product/dynamixel/ax_series/dxl_ax_actuator.htm, 2010.

[18] "Raspbian os." https://www.raspbian.org/, 2016.

[19] R. Dearmas, N. Fúrquez, J. Pereira, and A. Ricca, "Futbot: Informe de relevamiento," tech. rep., Facultad de Ingeniería, UdelaR, 2013.

[20] "Códec h.264 en la international telecommunication union (itu)." https://www.itu.int/rec/T-REC-H.264, 2016.

[21] H. Nguyen, "Comparativa de imágenes logitech quickcam pro 9000, b910, c910, c920." http://www.ubergizmo.com/reviews/logitech-c920-webcam-review/, 2012.

[22] "Hoja de datos del buffer tri-estado 74ls241." http://ecee.colorado.edu/~mcclurel/sn74ls240rev5.pdf, 2016.

[23] RoboCup, "2015 robocup soccer humanoid league rules and setup," 2015. https://www.robocuphumanoid.org/materials/rules/.

[24] R. Dearmas, N. Fúrquez, J. Pereira, and A. Ricca, "Futbot: Pruebas de visión," tech. rep., Facultad de Ingeniería, UdelaR, 2013.

[25] "Motores ax-12," 2016. http://support.robotis.com/en/product/dynamixel/ax_series/dxl_ax_actuator.htm.

[26] "Manual de referencia de la librería avr libc." http://www.atmel.com/webdoc/AVRLibcReferenceManual/index.html, 2016.

[27] "Wiringpi," 2016. http://wiringpi.com/.