UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de herramientas en software para el manejo, monitoreo y programación de la nueva versión del humanoide Robonova

Trabajo de graduación presentado por Stefano Papadopolo Cruz para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2023

UNIVERSIDAD DEL VALLE DE GUATEMALA
Facultad de Ingeniería

Desarrollo de herramientas en software para el manejo, monitoreo y programación de la nueva versión del humanoide Robonova

Trabajo de graduación presentado por Stefano Papadopolo Cruz para optar al grado académico de Licenciado en Ingeniería Mecatrónica

Guatemala,

2023

Vo.Bo.:

(f) _____

Ing. Miguel Zea

Tribunal Examinador:

(f) _____

Ing. Miguel Zea

(f) _____

MSc. Carlos Esquit

(f) _____

Ing. Miguel Zea

Fecha de aprobación: Guatemala, 5de enerode 2024.

Prefacio

Mi curiosidad desde que era niño de cómo funciona las cosas, constante necesidad de seguir aprendiendo y la mentalidad de "que tan difícil puede ser.[en] todas las áreas de la vida me han llevado al deseo de ejercer como ingeniero. El crecimiento y aprendizaje que he tenido en esta carrera ha sido una oportunidad como ninguna otra de la cual he aprendido bastante y ha incrementado estas cualidades inquisitivas en mi vida.

El estudiar una licenciatura en ingeniería mecatrónica por cinco años, aprender acerca de diseños mecánicos, eléctricos y de control, además de programación en una amplia gama de lenguajes culmina aquí: en la robótica. Esta tesis cubre todas las áreas de lo aprendido en la carrera. Desde la revisión de diseños CAD y su ensamblaje y la creación y programación de circuitos simples con un microcontrolador, hasta la creación de un programa para el control y simulación de un robot humanoide.

En esta sección quiero aprovechar para agradecer a mi asesor el Ing. Miguel Zea y a todos mis compañeros guaduandos. Sin su apoyo al momento de compartir ideas, discutir nuestras distintas opiniones y aprender y crecer juntos estos últimos años, este trabajo no hubiera sido posible. También quiero agradecer a mi pareja ya que de no ser por su constante presión, apoyo y ánimo, escribir esta tesis hubiera tomado una cantidad vergonzosa de tiempo.

Índice

Lista de figuras

Lista de cuadros

Resumen

Este trabajo graduación se basa en la revitalización de los robots Robonova-1 de la Universidad del Valle de Guatemala. El objetivo de este proyecto es diseñar una herramienta de software para monitorear y programar los Robonova de forma gráfica. El proyecto consiste en el diseño del software y coreografías básicas, la comunicación con el controlador del robot y el monitoreo en tiempo real del mismo.

Para lograr lo anterior, se tomarán como ejemplo herramientas de software como RoboBASIC de David Buckley y Coreographe de Aldebaran Robotics, ambos siendo programas para facilitar la programación de robots humanoides por medio de una interfaz gráfica. Para lograr el movimiento correcto de este robot humaniode se tomará en cuenta la biomecánica detrás de los movimientos básicos de una persona como caminar y saludar. Para analizar matemáticamente y visualizar estos movimientos se utilizarán librerías de Python específicas para el análisis físico y control robótico siendo estas pyBullet y Robotics Toolbox, respectivamente.

El proyecto consiste de tres distintos objetivos: el diseño de una interfaz gráfica de usuario, la creación de rutinas de movimiento sencillas y la conexión en tiempo real al robot Robonova. Estos tres objetivos fueron alcanzados correctamente con el programa resultante. El diseño de la interfaz de usuario se llevó a cabo utilizando la librería de diseño PyQt5 de python. Esta interfaz interactúa con una simulación del robot creada utilizando el motor físico pyBullet. El programa permite al usuario controlar directamente las juntas del robot, guardar y cargar posiciones de interes y crear rutinas de movimiento que pueden ser editadas y visualizadas en la simulación. Además de esto, también permite la conexión a través de WiFi al Robonova y su control de la misma manera que en la simulación.

Abstract

This graduation project is based on the revitalization of the Robonova-1 humanoid robots preset at Univesrsidad del Valle de Guatemala. The main objectivo of this project is to design a program for controlling, designing and monitoring the robots movements in real time.

To achieve this, various sources were used as an example to set the groundwork. Mainly, the original programming software for the Robonova, RoboBASIC, by David Buckley and the coreography design software for NAO Robots, Coreography, by Aldebaran Robotics. The biomechanics behind human movement was taken into account in order to get the correct emulation of basic motion such as greeting or walking. In order to visualize and analize this movement, various physics centric python libraries were used. Mainly, pyBullet and Robotics Toolbox.

The project consists of 3 distinct objectives: the design of a graphic user interphase (GUI), development of simple routines, and real-time connection to the robot at hand. These three objetives were all achieved in the resulting program. The design of the GUI was achieved using PyQT5 python library. This GUI interacts with a simulation of the robot created in the pyBullet physics engine. It allows the user to control the robots joints directly, save and load positions of interest and create basic movement routines that can be edited and visualized in the simulation. Along with this, the program also allows the user to connect vía WiFi to the Robonova and control it in the same manner.

CAPÍTULO 1

Introducción

La presente tesis mostrará el trabajo de investigación, programación y experimentación realizado para la creación de un programa de manejo, monitoreo y programación de la nueva versión del robot humanoide Robonova. Esta cuenta con un varios intentos previos o similares en una sección de antecedentes los cuales fueron usados como inspiración durante el diseño del programa. Además de ello, también se explica la investigación llevada a cabo en las áreas de programación, robótica y biomecánica que se llevó a cabo para la creación del programa y la emulación de movimiento humano llevado a cabo en el mismo.

El proyecto consiste de 3 distintos objetivos: diseño de la interfaz gráfica, diseño de rutinas y conexión en tiempo real al Robonova. Estos tres objetivos se presentarán en dos capítulos separados. El primero definiendo la programación, diseño y uso de la interfaz gráfica para el control de la simulación y creación de rutinas y el segundo será acerca de la conexión y el control del robot. La interfaz gráfica consistirá de dos ventanas. Una para visualizar la simulación y otra para controlar el controlarla y crear las rutinas, además de habilitar y deshabilitar la conexión al Robonova. La conexión al Robonova en tiempo real correrá en paralelo a la simulación, permitiendo observar los movimientos del robot tanto en el programa como en el mundo real.

CAPÍTULO 2

Antecedentes

Este proyecto de graduación se trabajará con los robos humanoides Robonova sin embargo, debido a que no se han trabajado proyectos con estos robots dentro de la Universidad del Valle de Guatemala, los antecedentes a continuación serán obtenidos de fuentes externas a la institución. Estos antecedentes incluyen: la programación original en RobotBASIC de los Robonova como fue diseñado por David Buckley, un proyecto de robots bailarines autónomos que utilizó a los Robonova como prototipo y las características de otros robots humanoides como NAO de Aldebaran Robotics y Atlas de Boston Dynamics.

Programación en RoboBASIC

Originalmente, los robots Robonova eran programados a través del IDE RoboBASIC. Como su nombre lo indica, la programación de estos robots se llevaba a cabo en el lenguaje de programación BASIC. Para ello contaba con comandos generales de programación en BASIC además de comandos específicos para facilitar la programación de los Robonova, como lo es el control de los servos y la asignación de grupos de motores estipulada en el manual de RoboBASIC. Además de estos comandos para el control principal del robot, el programa también contaba con comandos para funciones adicionales como lo son comandos de sonido para identificar y generar diversas frecuencias y comandos de control para LCD.

Además de la programación de alto nivel en la que se trabaja, el software de RoboBASIC también cuenta con varias interfaces para facilitar aún más el control de los robots. La primera, llamada "Catch-and-Play", permitía controlar cada servomotor de los Robonova en tiempo real a través de la interfaz gráfica que se muestra en la figura 1.

Otra interfaz útil dentro de RoboBASIC era RoboScript. Esta podia considerarse como una combinación entre la interfaz de Catch-and-Play y la programación directa en BASIC. Contaba con comandos básicos necesarios para crear una rutina del robot además de poder modificar la posición de los servos a través de una interfaz gráfica que se muestra en la Figura 2.

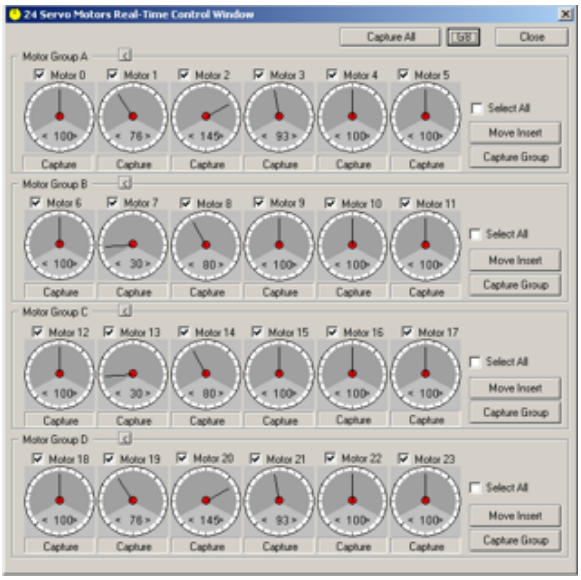

Figura 1: Captura de pantalla de la interfaz Catch and Play de RoboBASIC

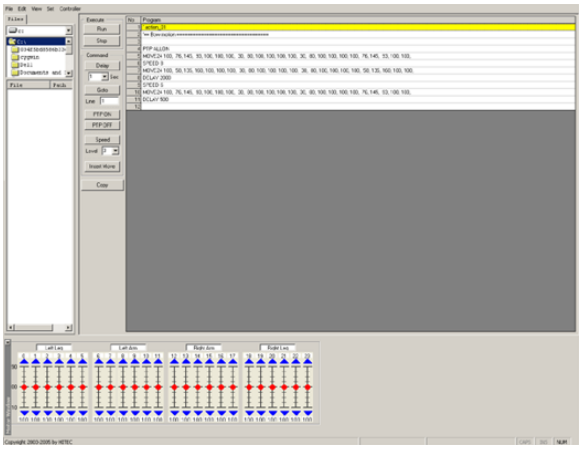

Figura 2: Captura de pantalla de la interfaz RoboScript de RoboBASIC

Por úlitmo, los Robonovas cuentan con un control remoto. Este puede ser utilizado para controlar directamente el movimiento del robot o para activar rutinas del mismo. Esta configuración es establecida a través de RoboRemocon, la última interfaz auxiliar de RoboBASIC. Cabe notar que no se lleva a cabo ningún tipo de programación en esta como se hacía en las anteriores, RoboRemocon solamente asigna las funciones y rutinas ya programadas con alguna de las herramientas ateriores y las asigna a un botón específico del control. La interfaz se muestra en la Figura 3.

Programación de robots bailarines

En [1] se presenta un proyecto cuyo objetivo era crear un programa para que los robots Robonova-1 reaccionaran a señales de audio y bailen al ritmo de la música.

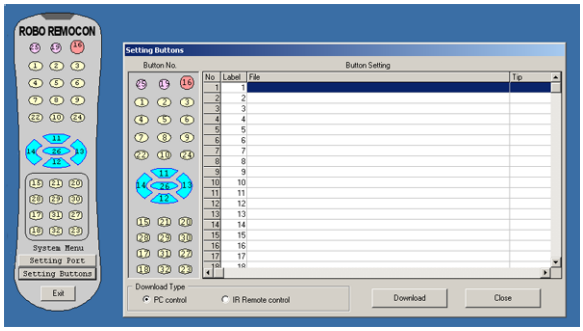

Figura 3: Captura de pantalla de la interfaz RoboReomocon de RoboBASIC

Los algoritmos de control de este proyecto se dividieron en 3 etapas: identificación del ritmo, la plataforma a utilizar (Robonova-1) y la generación de gestos y control. Para la primera etapa de identificación de ritmo, dado que el Robonova no contaba con el hardware necesario para el procesamiento de las señales, se realizó el programa en una computadora externa al robot la cual se comunicaba con el mismo por medio de Bluetooth. Este programa externo se encargaba tanto de el procesamiento de la música como la transmisión de los gestos al robot. El baile del Robonova consistía en una concatenación de diferentes gestos ya programados cuya velocidad de acción y de cambio entre gestos era regida por el ritmo analizado anteriormente.

Además de implementar el programa para hacer que el robot bailase, el equipo también creó un nuevo ecosistema para controlarlo en lugar de utilizar RoboBASIC. Lograron crear un mejor ecosistema de control que contaba con una latencia mucho menor a la obtenida al utilizar RoboBASIC (0.2 segundos) además de contar con una mayor exactitud en cuanto a las posiciónes deseadas de los servos. Como se evidencia en la Figura 4.

Atlas de Boston Dynamics

Una de las compañías más innovadoras en el área de la robótica es Boston Dynamics. En su repertorio de robots cuentan con varios robots cuadrúpedos (Spot, Big Dog,LS3, etc.), robots con ruedas (Handle) y robots humanoides como es el caso del robot Atlas. Con 28 grados de libertad y un sistema de control de estado del arte, Atlas es uno de los robots humanoides con mejor movilidad en la academia. Con este robot, Boston Dynamics demuestra los posibles usos de robots humanoides y como estos podrían revolucionar la industria. Este se puede observar en la Figura 5.

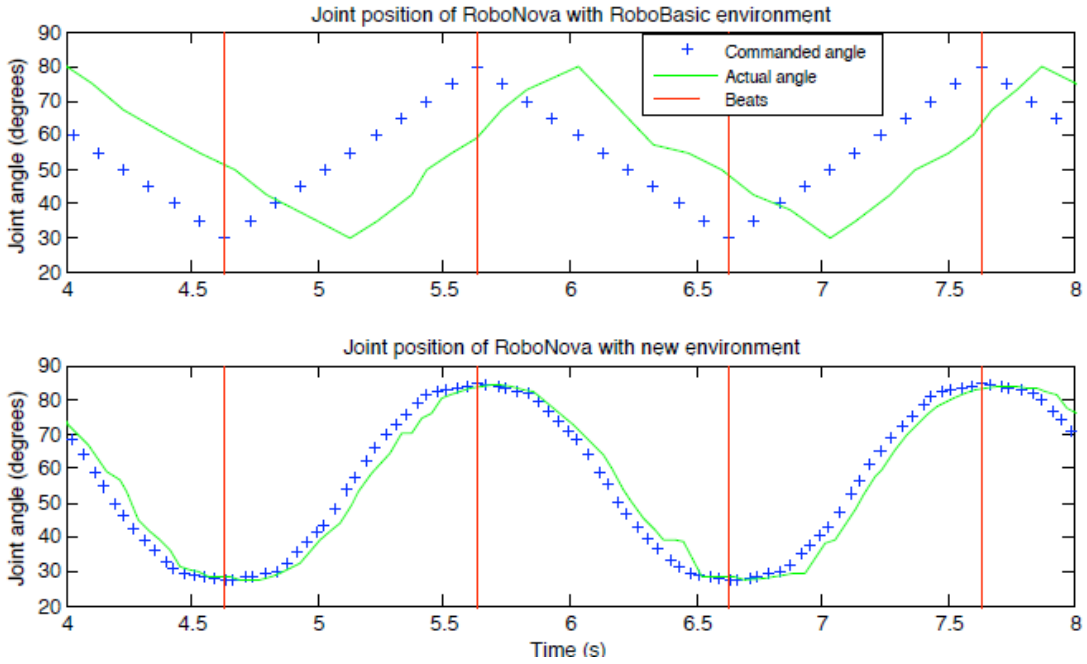

Figura 4: Comparación entre RoboBASIC y el ecosistema creado por investigadores de Drexel University

Figura 5: Robot Humanoide Atlas de Boston Dynamics

CAPÍTULO 3

Justificación

En los últimos años se han dado bastantes avances en robótica gracias a la investigación y desarrollo que varias instituciones llevan a cabo utilizando robots humanoides.Desde la manufactura de piezas mecánicas más eficientes hasta la programación de sistemas de control más avanzados y robustos, el uso de estos robots en investigación ha aportado bastante en el área de ingeniería. Por esta razón, este proyecto se llevará a cabo con el fin de revitalizar los robots humanoides Robonova-1 disponibles en la Universidad del Valle de Guatemala. Lo que se desea es actualizar su interfaz de programación además de su conexión y control de forma inalámbrica. El contar con estas herramientas aplicadas a un prototipo como el Robonova mejorará la investigación y aprendizaje dentro de la universidad en las áreas anteriormente mencionadas y abriría el campo para el futuro control de robots humanoides más sofisticados.

Como se pudo observar en los antecedentes, la plataforma RoboBASIC que se utiliza por defecto para la programación de los Robonova, a pesar de contar con diversas interfaces para facilitar la creación de rutinas, no es tan amigable y ya no cuenta con soporte, con su última actualización siendo en mayo de 2008. Además de ello, no cuenta con monitoreo en software de el estado actual del Robonova lo cual es una característica importante para el control de robots. Lo que se busca entonces es crear una interfaz de programación similar a Coreographe (la interfaz utilizada por los robots NAO). Esta debe permitir al usuario observar el estado de los robots y crear las rutinas en una interfaz comprensible y amigable.

CAPÍTULO 4

Objetivos

Objetivo general

Diseñar una herramienta de software para el monitoreo y programación de los robots humanoides Robonova-1 de forma gráfica.

Objetivos específicos

- Conectar la herramienta de software con el robot de forma inalámbrica para su programación y monitoreo en tiempo real.
- Crear una librería de subrutinas para facilitar la creación de coreografías complejas.
- Crear una interfaz gráfica amigable y sencilla para el fácil desarrollo de coreografías de los robots.

CAPÍTULO 5

Alcance

El alcance de este proyecto constituye la creación de dos programas para la conexión y control de la nueva versión del robot humanoide Robonova.

El primer programa, escrito en python, será el cliente que el usuario utilizará para controlar al robot. En él se encuentra un control directo de las juntas del robot por medio de diales, un controles para guardar y cargar posiciones específicas y una sección para la creación de rutinas del robot. El programa también cuenta con una simulación la cual toma en cuenta las consideraciones cinemáticas del robot. Este programa debe ser fácil de modificar para su utilización con otros diseños o robots, no solamente el Robonova.

El segundo programa, escrito en Arduino, consiste en la conexión del servidor (Robonova) con el cliente descrito anteriormente por medio de un modulo WiFi de un ESP32. Este programa también debe contar con el control directo de la posición de los servomotores del robot.

CAPÍTULO 6

Marco teórico

Biomecánica del movimiento humano

Para lograr que un robot bípedo emule correctamente los movimientos de una persona, primero se debe empezar por analizar la mecánica de una persona en movimiento. El área de estudio que se enfoca en esto se denomina biomecánica. Esta se enfoca en modelar a los seres vivos por medio de representaciones físicas más sencillas como masas, eslabones, uniones, resortes, etcétera, para poder implementar un análisis cinemático y dinámico de su movimiento.

Como se puede observar en la Figura 6, un modelo simple para una persona (o robot humanoide) es un conjunto de eslabones unidos por diferentes tipos de uniones. Durante su movimiento, estos eslabones se moverán de acuerdo a los grados de libertad que cada unión permita. Para modelos biomecánicos sencillos, estas uniones generalmente son revolutas con un grado de libertad. Cabe notar que, a demás de las articulaciones, también se encuentra marcado el centro de masa del modelo al rededor del cual se llevan a cabo los cálculos.

Ciclo de Gait

Al llevar a cabo movimientos que se repiten una y otra vez como es el caso de caminar y correr, estos pueden ser analizados por medio de un ciclo de Gait. Este ciclo separa los movimientos es distintas fases para su análisis. El ciclo de Gait de caminar puede ser de dos fases: la postura y el balanceo. Durante la fase de postura, la cual es el 60 % del ciclo de Gait [2], la pierna se encuentra en contacto con el piso. Este contacto se subdivide en 5 fases: *Heel strike*, ocurre cuando el talón entra en contacto con el piso; *loading response*, cuando este se prepara para cargar con todo el peso del cuerpo; *midstance*, donde todo el peso se encuentra sobre esa pierna; *heel off*, cuando el pie comienza a levantarse del piso; y por último, *toe off*, cuando ya se levantó por completo. Durante la fase de balanceo, como lo indica su nombre, la pierna se balancea con momentum hacia adelante para llegar a su nueva posición. Esta fase también puede ser dividida en 3 fases propias: aceleración, balanceo y desaceleración.

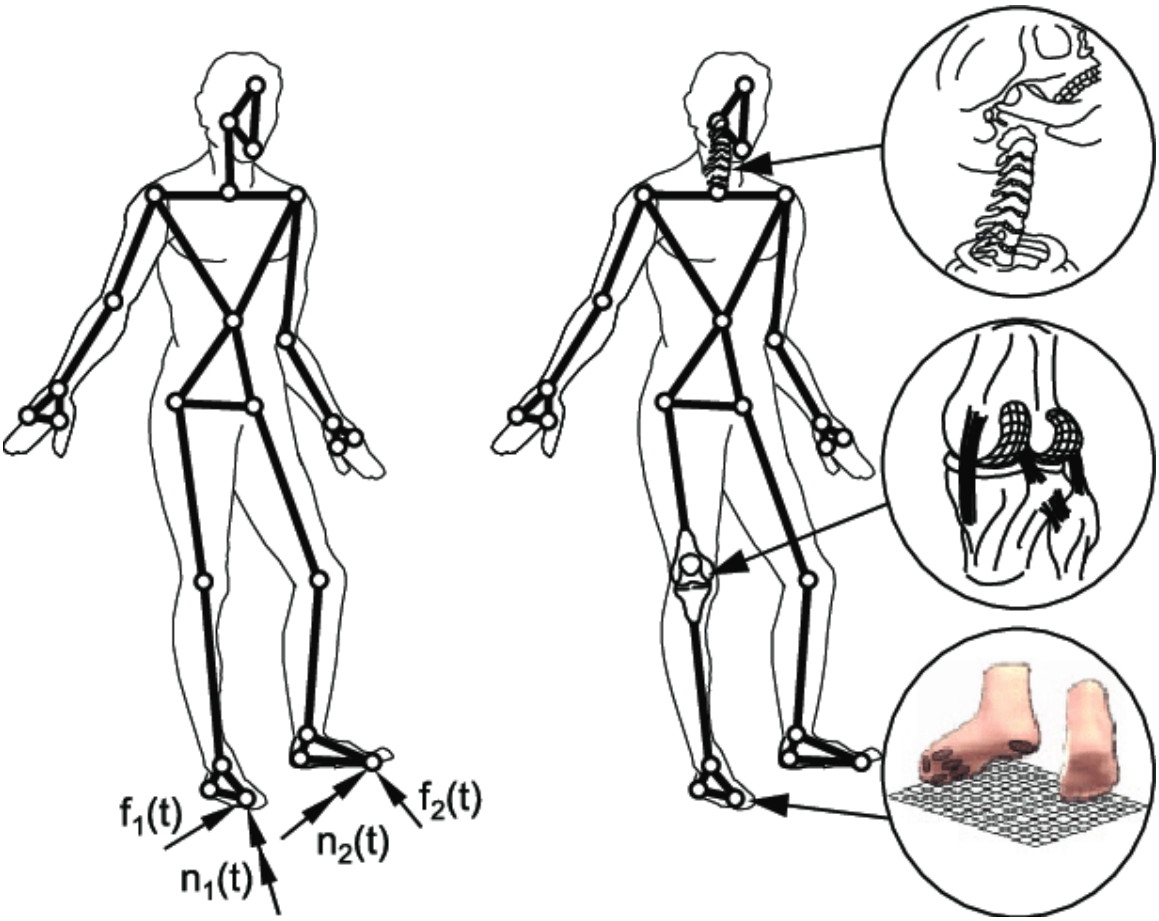

Figura 6: Modelo de juntas y eslabones de un sistema biomecánico

Para un bípedo, el proceso de caminar consiste en un ciclo de gait continuo en cada pierna, desfazados entre sí. [2]

Cinemática de robots humanoides

Un robot humanoide es un tipo específico de robot de base flotante cuya forma tiene inspiración en el cuerpo humano. Por ello, la mayoría de estos robots cuentan con 4 extremidades: 2 brazos y 2 piernas. En los robots de base flotante, generalmente, su base se encuentra en el centro de masa del mismo. Ya que no es posible actuar sobre la base directamente este tipo de robot moviliza su base de forma indirecta controlando sus extremidades.[3]

Para el control de las extremidades del robot se puede considerar cada una como un manipulador serial independiente saliendo de la base. Como se observa en [4], cada una de las extremidades tendrá su propia matriz de Denevit-Hartenberg en relación a la base flotante del robot. Al plantear el robot de esta manera, la cinemática directa e inversa de los brazos del mismo se vuelve trivial.

Este no es el caso para las piernas del robot. Debido a que estas deben sostener el peso entero del robot, además del control cinemático que se lleva a cabo, también deben tomarse

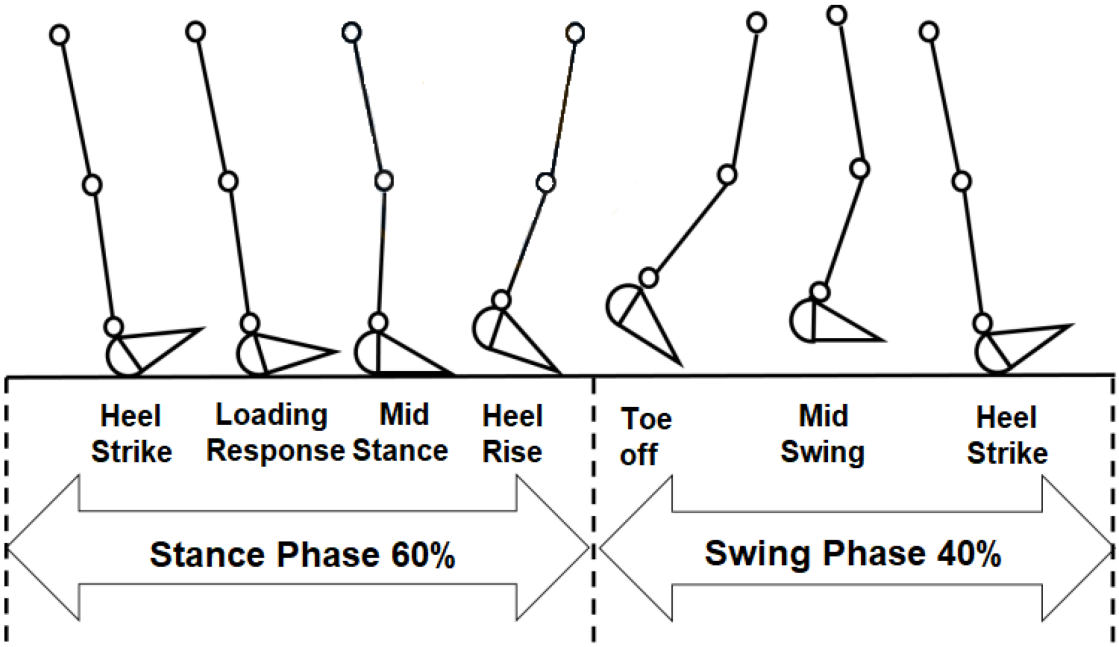

Figura 7: Ciclo de Gait

en cuenta consideraciones dinámicas al controlar las piernas. En [5], se muestran dos modelos para tomar en cuenta estas consideraciones: el modelo LIMP (Linear Inverted Pendulum Model) y el modelo ZMP (Zero Moment Point). En el modelo LIMP, las piernas actuan como una distancia desde el pivote (el suelo) hasta el centro de masa del robot, simplificando así el análisis fisico. Una vez modelado de esta forma el ZMP es el posicionamiento del centro de masa de tal manera que este no genere un torque en el plano xy evitando así que el robot pierda su estabilidad al momento de dar un paso.

PyBullet

PyBullet es una librería para Python del Bullet Physics engine. Esta librería se enfoca en el estudio de detección de colisiones y dinámica de cuerpos rígidos. La librería cuenta con simulaciones de dinámica directa, dinámica inversa, detección de colisiones y intersección de rayos. Esta librería se utiliza principalmente la simulación de sistemas de robótica antes de su implementación física además de tener aplicaciones en videojuegos y realidad virtual.

Con el área de interés siendo simulaciones de robótica, PyBullet cuenta con herramientas para crear y analizar diversos tipos de robots. Además de contar con una librería de robots de ejemplo, PyBullet cuenta con comandos para cargar la información de un robot especifico en diversos formatos como URDF, SDF, MJCF, entre otros. Una vez cargados los datos del robot se puede analizar y controlar dentro del espacio de simulación. PyBullet cuenta con funciones para obtener las propiedades físicas del robot además de funciones que le permiten al programador controlar el mismo.[6]

Además de esto, la librería también cuenta con funciones para el cálculo mátemático de la cinemática y dinámica inversa de los robots, detección de colisiones, visión de computadora

y realidad virtual.[6]

Robotics Toolbox de Peter Corke

Esta librería es una versión planteada para Python de la Robotics Toolbox para MATLAB de Peter Corke, y cuenta con herramientas para el análisis físico de robots. Utilizando todas estas herramientas, el programa puede modelar robots, analizar y crear trayectorias, llevar a cabo manipuulación simbólica de variables, analizar la cinemática diferencial del robot e incluso la dinámica del mismo.[7]

La Robotics Toolbox ya cuenta con más de 30 modelos cinemáticos de diversos tipos de robots. Además de esto, es posible utilizarla para modelar robots propios de diversas maneras. Se puede generar la matriz de Denavit-Hartenberg con las dimensiones adecuadas del robot o también se pueden cargar los datos del mismo por medio de un string ETS o un archivo URDF. Además de los parámetros cinemáticos, también se pueden agregar otros parametros como la masa, CoG, inercia de los motores, limites de las articulaciónes, etcetera, utilizando palabras clave dentro de la programación.

En cuanto a las trayectorias, el Robotics Toolbox trabaja con polinomios de quinto orden para proporcionar un *jerk* continuo. Estas trayectorias proporcionan tanto la posición, velocidad y aceleración de los actuadores para llegar al punto deseado.

Como se estipuló anteriormente, la Robotics Toolbox también computa cinemática diferencial de los robots. Puede computar los Jacobianos, Hessianos y la manipulabilidad de todos los robots.

Por último, también es posible analizar la dinámica de los robots utilizando integración numérica. Esta se puede utilizar para calcular los términos de inercia, Coriolis y gravedad por medio de dinámica inversa.[7]

Onshpe-to-robot

Onshape-to-robot es una librería creada por Rhobah, un grupo de investigadores de la Universidad de Bordeaux. Esta heramienta permite importar robots diseñados en Onshape CAD a formatos de descripción como URDF o SDF para poder utlizarlos dentro de simuladores físicos como pyBullet. [8] Esta librería crea el URDF en base a las dimensiones de de los eslabones y los tipos de juntas utliizados en el ensamble general de Onshape. Además de generar el URDF, la librería también genera los archivos .stl de todas las piezas del robot.

PyQT

PyQT es una librería de python que permite crear aplicaciones QT. Esta aplicaciones QT consisten de una interfaz de usuario utilizando diversos tipos de entradas para que el usuario interactue con el programa de python subyacente.[9]

CAPÍTULO 7

Programa en python para simulación y control de Robonova

Metodología

Partiendo desde los antecedentes, el objetivo del nuevo programa para el control de la nueva versión del Robonova fue poder recrear lo hecho por David Buckley en el programa de RoboBASIC [10] original e incluir funcionamiento adicional inspirado en la programación de los robots NAO y otras fuentes. Además de las funciones ya existentes en los antecedentes, se incluyó una simulación en pyBullet en el programa. Esto fue con el afán de facilitar la logística de programación de rutinas del robot. El incluir la simulación, también permite expandir el uso del programa para crear rutinas con diversos robots, no solamente el Robonova.

Lo primero que se trabajó fue la implementación del simulador para poder trabajar la programación del robot sin necesidad de estar físicamente conectado al mismo. Para ello se utilizó un motor físico en python. Esto permite simular los movimientos del robot bajo consideraciones cinemáticas, dinámicas y mecánicas, dando una buena aproximación a lo que sería el movimiento del robot en la vida real. Una vez creada y conectada la simulación al programa, esta se pudo utilizar como base para cumplir con todos los requerimientos de programación del robot incluidos, pero no limitados a: control de posición de las articulaciones del robot en la simulación, guardar y cargar una posición en una memoria volátil y la creación y reproducción de archivos con coreografías complejas del robot.

Resultados

Simulación en pyBullet

La construcción de la simulación en pyBullet del robot puede separarse en tres etapas: inicialización del mundo, inicialización del robot y control del robot dentro de la simulación.

La inicialización del mundo consiste en conectar la simulación a la interfaz gráfica de pyBullet y reiniciar la simulación para asegurar que esta inicie correctamente. Además de esto, también se hacen configuraciones de cámara de la interfaz de usuario, configuraciones físicas como las aceleraciones y de simulación como el tiempo entre cuadros de la simulación. Para finalizar, se carga un plano para hacer el piso de la simulación. Este mundo inicial puede ser observado en 8.

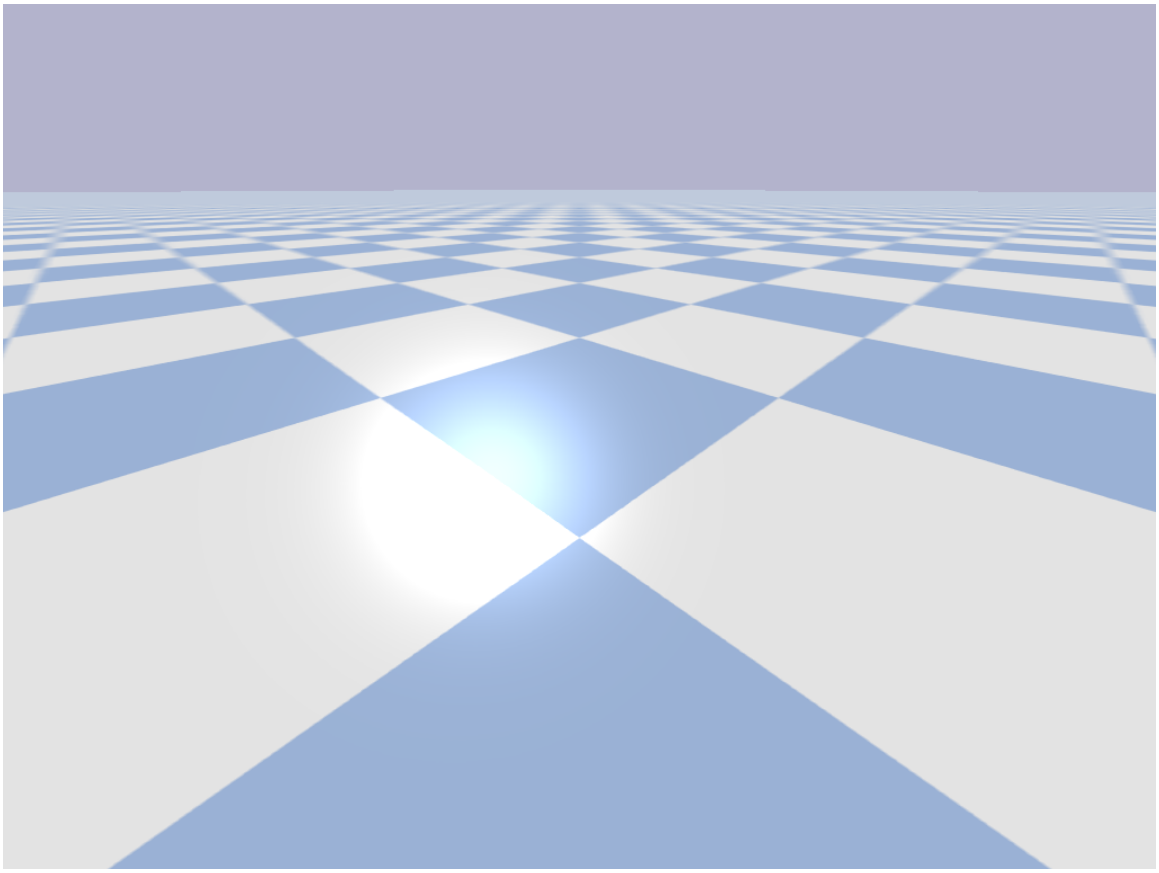

Figura 8: Mundo Inicial en pyBullet

Una vez iniciado el mundo, se carga el modelo del robot. Este puede ser observado en 9. En este caso, se cargó el URDF del Robonova. Se puede extraer la información del robot (número de juntas, nombres de las juntas, tipos de juntas, limites, etc.) utilizando las funciones de pyBullet *getNumJoints* y *getJointInfo*. Esta información es utilizada por el programa para determinar automáticamente las juntas y sus límites y crear las diales necesarias para el control de las juntas. Esto permite que la simulación pueda ser utilizada con cualquier URDF que se introduzca, no solo el Robonova.

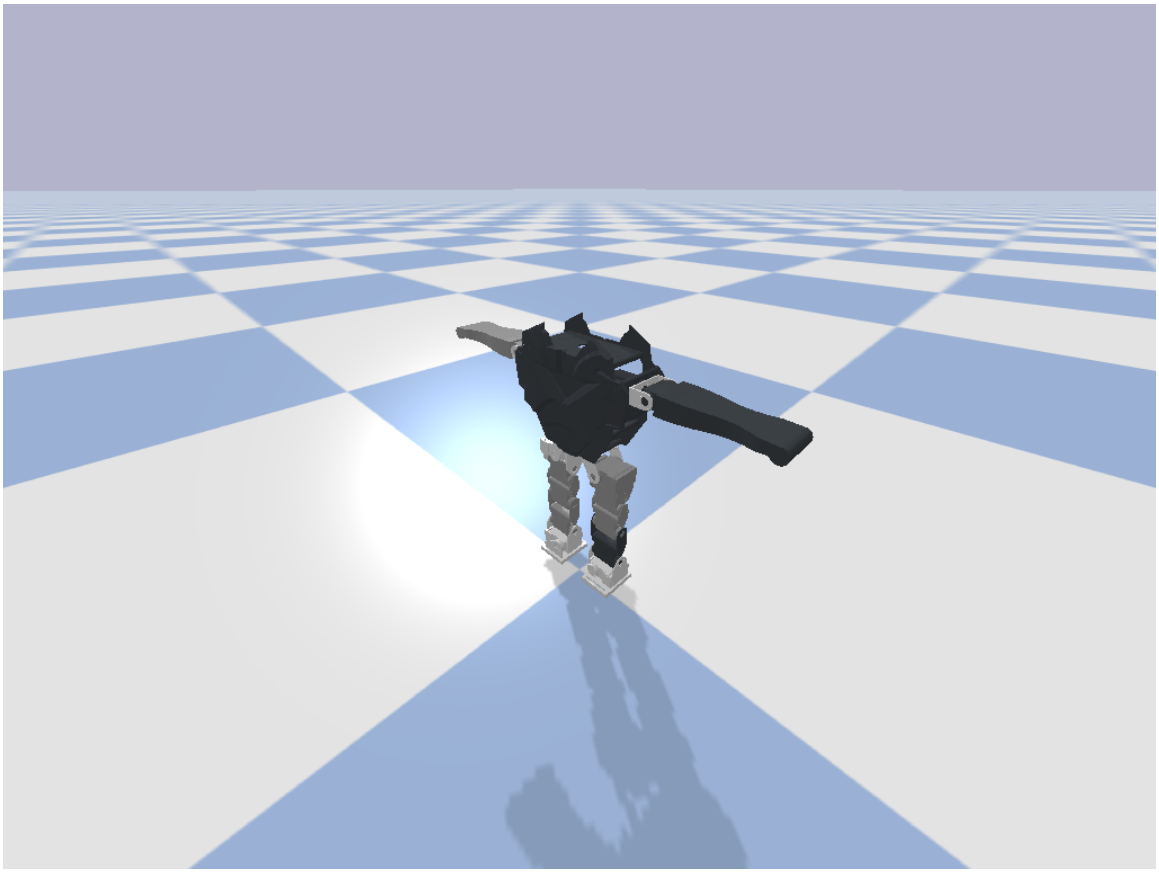

Figura 9: Modelo de Robonova en pyBullet

Por último, el control del robot dentro de la simulación se lleva a cabo con la función *setJointMotorControlArray*. Al trabajar con un robot humanoide con juntas a base de servomotores, se introduce a esta función un *array* con las posiciones angulares deseadas de los servomotores. Para poder observar la simulación a lo largo del tiempo se crea un bucle infinito con la función *stepSimuation* y la anteriormente mencionada de *setJointMotorControlArray* para poder observar y cambiar las posiciones de las juntas del robot. Una imagen del Robonova con las posiciones de sus servomotores ya manipuladas se puede observar en 10.

URDF de Robonova

Para poder observar el robot que se desea trabajar en la simulación de pyBullet, es necesario contar con el modelo en un formato estandarizado de definición de robot (URDF, SDF, etc). Grandes compañías de robótica como Faunc, Kuka, etc. proveen a sus clientes el URDF de los robots para que estos puedan simular las operaciones y determinar si es lo que necesitan. Además de esto, también existen diversas formas de crear archivos URDF de diseños propios. Desde la creación directa del archivo hasta el uso de programas que crean el URDF en base a los archivos de ensamblaje y diseño del robot.

En el caso de la nueva versión del Robonova, este fue diseñado en OnShape. Gracias

a esto, se puede utilizar el programa Onshape-to-Robot para crear automáticamente su URDF. Para ello, dentro del programa de OnShape se creó un ensamblaje del robot que contaba únicamente con los grados de libertad con los que cuenta el robot real. Luego, se creó un archivo JSON de configuración para el programa Onshape-to-Robot donde se hace referencia a este ensamblaje. Una vez creada la configuración, se corre el programa el cual se conecta a la API de OnShape y extrae toda la información necesaria del diseño para crear el archivo URDF.[1]

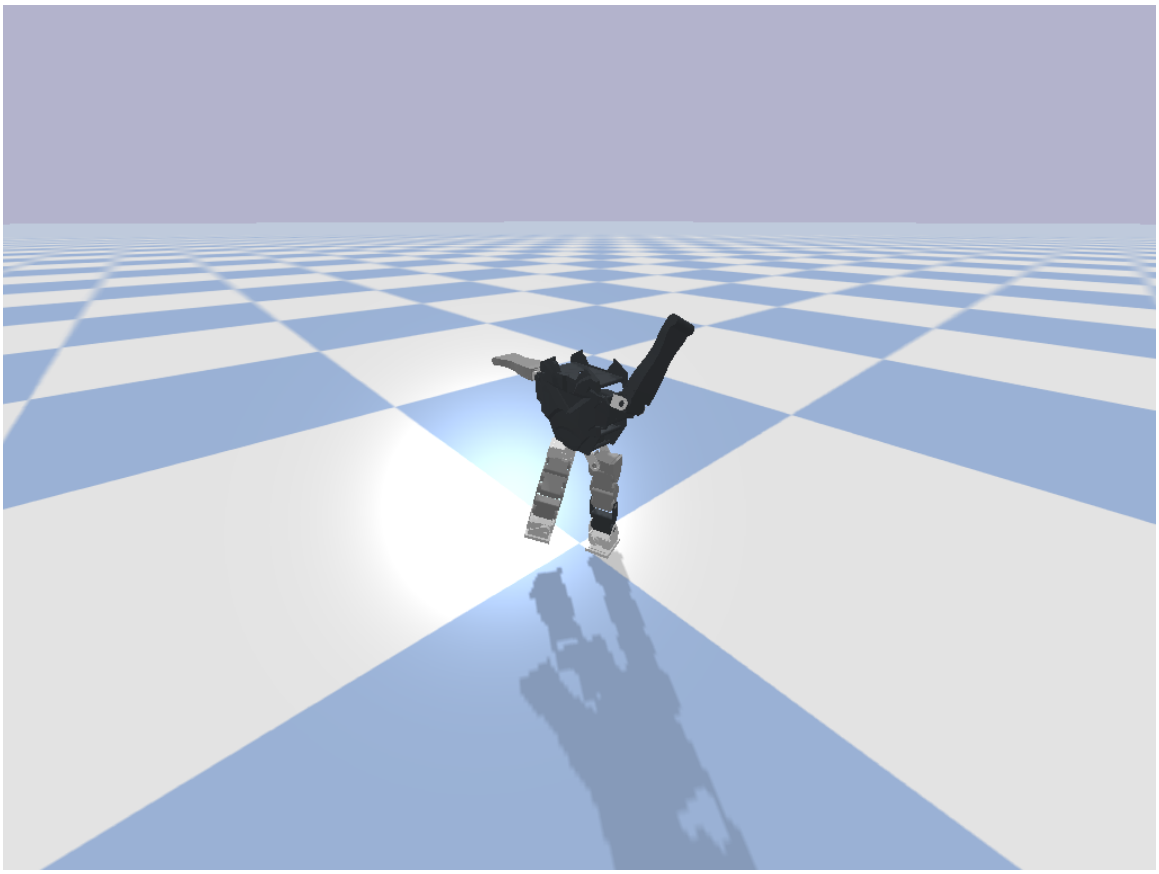

Figura 10: Simulación Robonova con posiciones de los servomotores cambiada en pyBullet

[1]El programa Onshape-to-Robot solamente puede ser utilizado en Linux. Esto debido a una integración innata del programa con ROS y que se debe configurar las claves dentro del archivo .basch para conectar al sistema operativo con la API de OnShape.

Interfaz de usuario

La interfaz de usuario fue creada utilizando la librería PyQt5.11 El objetivo de la interfaz es que el usuario pueda interactuar fácilmente con la simulación (y con el robot) para controlar los servomotores y crear rutinas para el robot. Este es el programa principal donde se correrá todo. Para poder correr todo el programa de forma simultánea se utilizó la librería *threads*. Se creó un hilo dentro del programa de la interfaz para que este, en paralelo a la misma, corra la simulación de pyBullet.

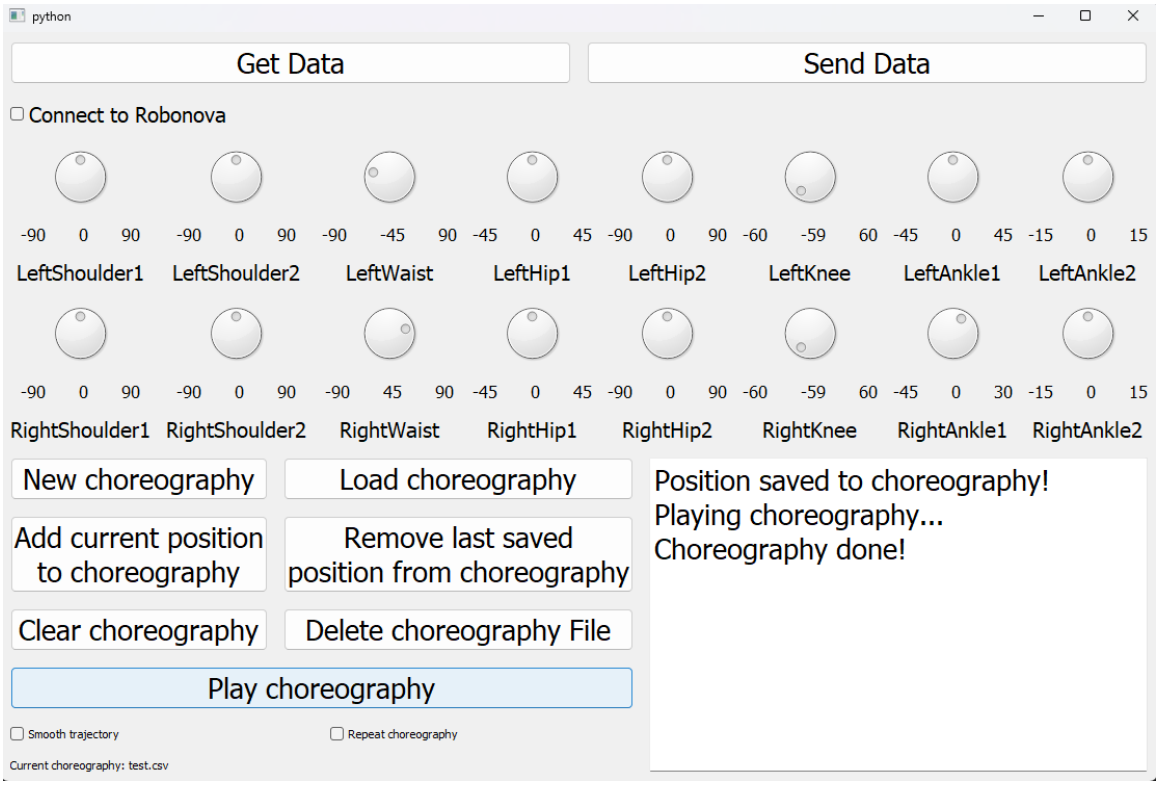

Figura 11: Interfaz de Usuario del programa de control del Robonova

PyQT5 cuenta con diversas clases para poblar la interfaz con las herramientas que utilizará el usuario. Las clases que se utilizaron fueron: *QButton* para crear los botones de pruebas y de configuración de rutinas; *QCheckbox*, para conectar o desconectar el ESP32 que controla el robot; *QDials*, para las diales que controlan los servomotores del robot; y *QPlainTextEdit*, para crear un cuadro de texto que le dé información al usuario. Para la creación de las diales, de igual forma que en la simulación, se crearon funciones para automatizar su creación. Se utilizaron las mismas funciones *getNumJoints* y *getJointInfo* para crear la cantiadad de diales exactas para controlar todos los servomotores. Esto con el afán de poder utilizar el programa con más robots. Una vez creados, todos estos *widgets* son conectados a sus funciones específicas para enviar o recibir información al programa. Las funciones son:

- *getData*

  Conectada al botón *Get Data*. Esta función toma los valores actuales del *array* de valores de servomotores y los guarda en una segunda variable. Una vez guardados,

imprime un mensaje en el cuadro de texto indicando que se guardaron los valores con éxito.

- *sendData*

  Conectada al botón *Send Data*. Toma la lista de valores guardada por la función anterior y guarda los mismos en las variables de las diales y de la posición de servomotores en la simulación. Después de eso, imprime un mensaje en el cuadro de texto indicando que se han cargado con éxito.

- *connectRobonova*

  Conectada al checkbox *Connect to Robonova*. Al ser activado sube la bandera de conexión al robot Robonova real. Más información acerca de la conexión en tiempo real en 8.

- *updateDial*

  Conectada a las diales. Se activa cada vez que una de las diales cambia su valor. Guarda ese valor en una variable que luego es introducida a la etiqueta de valor y a la posición en la lista *servoValues* que controla la posición de los servomotores en la simulación y en el robot.

- *newCoreo*

  Conectada al botón *New choreography*. Esta función usa las librerías pyQt5 y os para crear y seleccionar un nuevo archivo .csv donde será guardad la rutina. Utilizando la función *getSaveFileName* de la clase *QFileDialog* se puede crear el archivo (si este ya existe se sobrescribe) y con *os.path.basename* se extrae el nombre del archivo y se guarda en una variable. Esta variable es utilizada en la escritura al .csv en las funciones posteriores.

- *loadCoreo*

  Conectada al botón *Load choreography*. Muy similar a la función anterior, utiliza las librerías de pyQt5 y os. A diferencia de la anterior, esta utiliza la función *getOpenFileName* de la clase *QFileDialog*. Esta permite seleccionar (no crear) uno de los archivos ya existentes.

- *addCoreo*

  Conectada al botón *Add current position to choreography*. Como lo dice su nombre, esta función consiste en agregar la posición actual a la coreografía. Para ello primero hay que mover las diales que controlan los servomotores para colocar al robot en la posición deseada y luego presionar el botón. La manera en la que la función guarda la posición es bastante sencilla. Primero se abre el archivo .csv que fue seleccionado con alguna de las dos funciones anteriores. Una vez abierto, se utiliza la librería *.csv* para añadir a la siguiente fila la lista que tiene guardados los valores de todos los servomotores.

- *removeCoreo*

  Conectada al botón *Remove last saved position from choreography*. Remueve la última fila del archivo .csv activo borrando así la última posición de la coreografía. Esto se logra leyendo todo el archivo .csv y guardando los valores en una lista utilizando la

función *readlines*. Luego, se elimina la última fila de esta lista utilizando la función *pop* y se sobrescribe está lista ya modificada en el archivo .csv.

- *clearCoreo*

  Conectada al botón *Clear choreography*. Como lo dice su nombre, limpia por completo el archivo .csv para poder empezar la rutina desde cero. Muy similar a la función anterior, se sobrescribe una lista en el archivo .csv actual. La única diferencia es que en este caso se sobrescribe una con una *string* vacía para "borrar"todos los contenidos del archivo.

- *deleteCoreo*

  Conectada al botón *delete choreography*. De igual forma que se puede borrar la coreografía desde el explorador de archivos, también se puede borrar desde el mismo programa. Utilizando la función *remove* de la librería *os*, esta función toma el archivo .csv activo (que fue seleccionado con la función *newCoreo* o *loadCoreo* y lo elimina del sistema.

- *playCoreo*

  La función más importante del programa. Conectada al botón *Play choreography*. Esta función coloca los servomotores en las posiciones del archivo de la coreografía una tras otra en el orden en el que fueron guardadas. Al igual que en la función *removeCoreo*, el primer paso es leer todo el archivo de la coreografía y guardar los valores en una lista. Luego, se coloca la posición de los servomotores en las posiciones estipuladas por la fila actual de la lista una por una hasta llegar a la última fila. Existe un intervalo de tiempo estipulado al pasar de una posición a otra para darle tiempo a las revolutas de la simulación (y al Robonova físico) de llegar a la posición deseada.

  Además del funcionamiento básico para mostrar las posiciones estipuladas en al rutina una tras otra, esta función cuenta con dos configuraciones adicionales conectadas a las *chechboxes: Smooth trajectory* y *Repeat choreography*.

  Al activar la configuración de *Smooth trajectory* se toman las posiciones guardadas en el archivo de la coreografía y se hace una interpolación lineal entre las posiciones. Esto genera un movimiento más fluido lo cual será beneficiosos para el control del Robonova ya que evitará movimientos extremadamente bruscos que puedan llegar a dañar los servomotores o las juntas del robot.

  La segunda configuración, *Repeat choreography*, inicia un bucle en el cual el robot se mueve de una posición de la coreografía a la siguiente y al terminar regresa al inicio. Este bucle no se detiene hasta que se desactive esta opción. Esta opción permite visualizar de manera más fácil las rutinas sin tener que estar activándolas una y otra vez y, más importante aún, permite movimientos repetitivos como caminar y saludar — movimientos básicos que se podían hacer en RoboBASIC y se desean replicar en este programa.

Discusión

Como se puede observar con todas las funciones creadas y durante el uso del programa, este es un actualización más que adecuada al programa de control original del Robonova,

RoboBASIC.

Además de cumplir con todas las funciones de roboBASIC (control de servomotores por medio de diales, creación y carga de rutinas, enviar posiciones específicas, etc.) el programa también cuenta con características adicionales que facilitan la logística de programación de rutinas del Robonova. Como primer punto se encuentra la simulación en pyBullet incorporada en el programa. Esta simulación permite crear las rutinas y ver una aproximación bastante acertada al movimiento real del Robonova sin necesidad de tener el robot a la mano todo el tiempo. Además de esto, las funciones adicionales agregadas como la repetición de rutina y el suavizado de trayectorias permiten un mayor control en la creación de rutinas tanto para el usuario como para el programador. Al utilizar el suavizador de trayectorias, el usuario puede crear rutinas con pocos puntos de referencia y obtener una trayectoria fluida. De igual forma se puede decidir no utilizar esta opción y generar movimientos más rápidos y explosivos con el Robonova que no eran posibles con el programa de control original. Por último, la programación de la simulación y la interfaz de usuario se hicieron de forma tan general que permite trabajar con cualquier robot, contingente a que se tenga su URDF y los STLs de sus eslabones.

Gracias a todo esto, podemos determinar que, a pesar de que existen áreas de mejora las cuales serán las cuales serán estipuladas posteriormente en recomendaciones, el programa fue un éxito cumpliendo con todos los objetivos deseados y más.

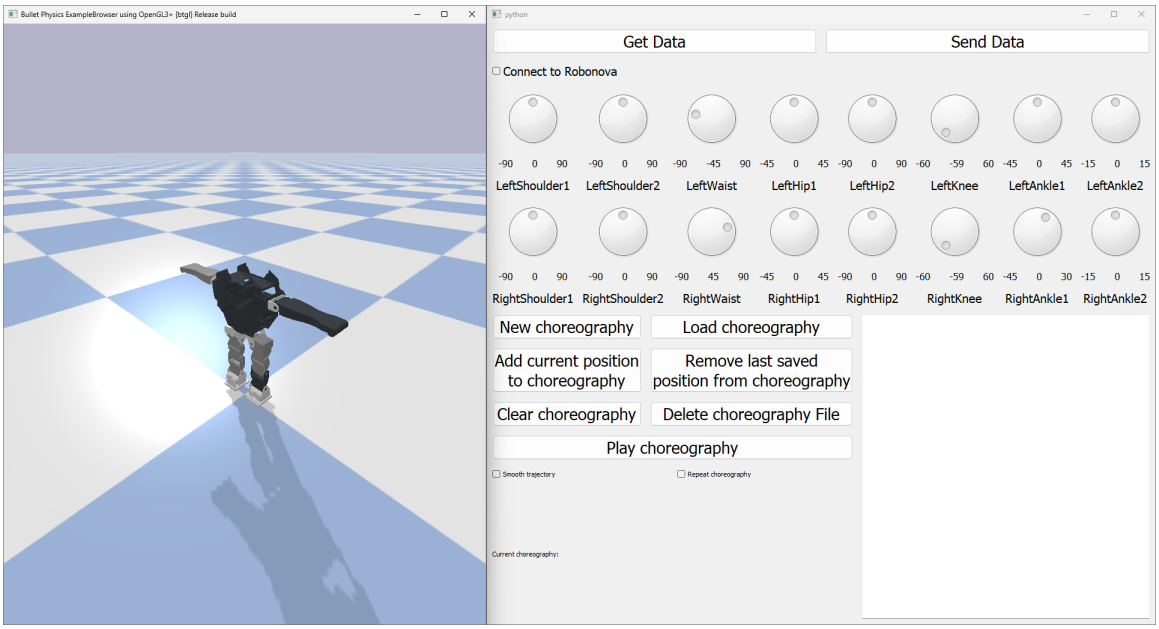

Figura 12: Captura de pantalla del programa y la simulación

CAPÍTULO 8

Control en tiempo real de Robonova

Metodología

Con el fin de controlar al robot Robonova de forma inalámbrica, se utilizaron TCP/IP Sockets para conectar el robot como servidor a un cliente en python a través del módulo WiFi de un ESP32 13. Una vez creado y cargado el programa de el servidor y control de servomotores en el ESP32 se desea poder controlar al Robonova inalámbricamente desde el programa de Python y poder enviar las rutinas al mismo sin necesidad de conectarlo cada vez que se cree o modifique una de ellas.

Resultados

Programación de servidor y controlador de servomotores en Arduino

Para crear el servidor se utilizó la librería *Wifi.h* de Arduino. Dentro del programa se guardaron los valores de SSID, contraseña y puerto de conexión a la red wifi utilizada para crear la conexión entre servidor y cliente y se creo una variable tipo *wifiServer*. Dentro del *setup* se conectó el ESP32 a la red WiFi definida anteriormente con la función *WiFi.Begin(ssid,password)*. Luego se inició un bucle que no se detiene hasta que la conexión haya sido establecida para asegurarse que el microcontrolador se conecte antes de proseguir. Una vez conectado a la red WiFi se inicializa el servidor con la función *wifiServer.begin()*. Por último, al estar trabajando con una IP dinámica, se imprime la dirección IP del dispositivo para utilizarla en la configuración del cliente.

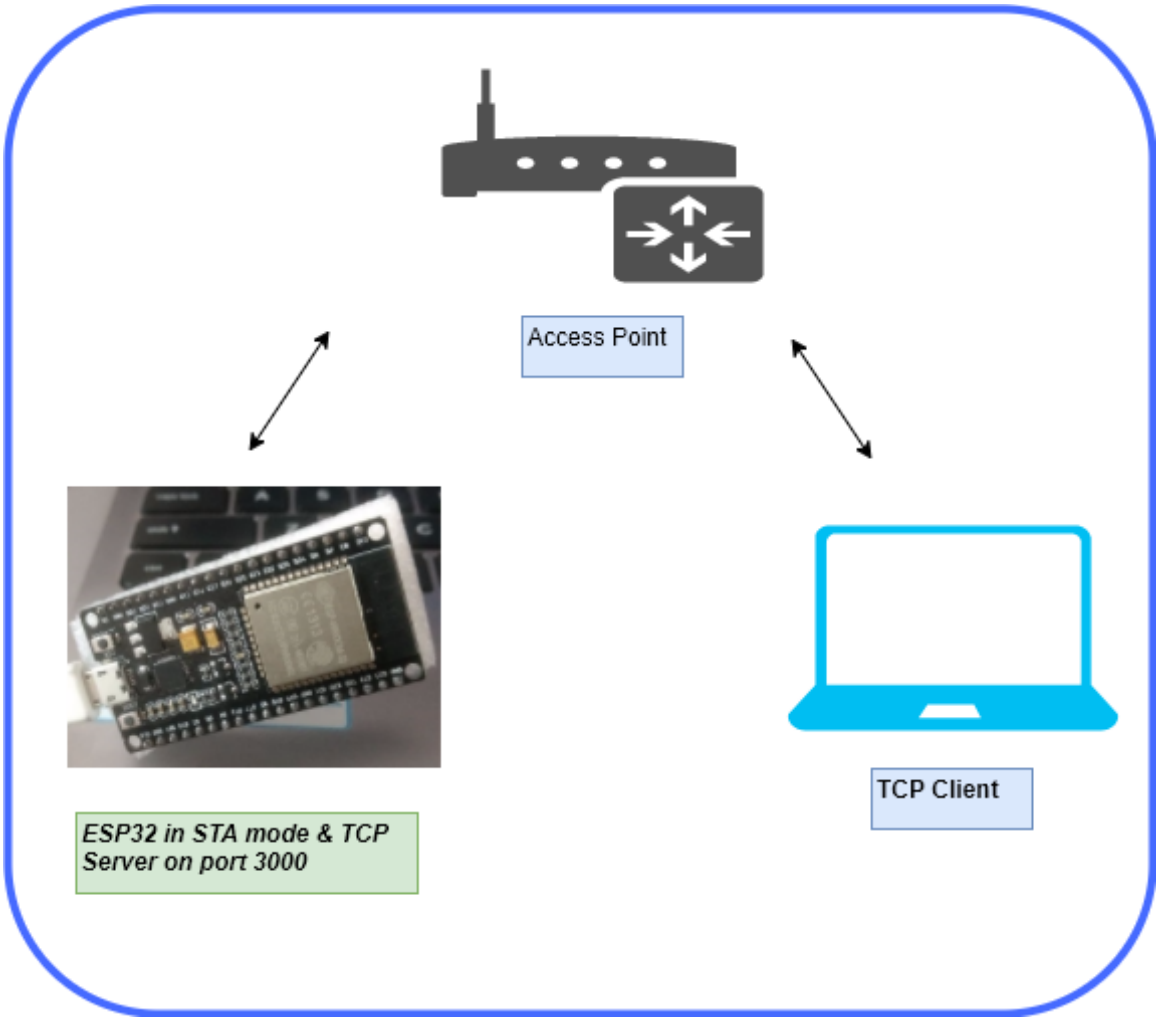

Figura 13: Conexión TCP entre servidor y cliente

Además de contar con el servidor activo, el programa del ESP32 también contiene funciones para extraer las posiciones de los servomotores de un JSON enviado por el cliente. Utilizando la librería *Adafruit_PWMServoDriver.h*, el ESP32 envía los valores extraídos del JSON a dos placas controladoras de servomotores para posicionarlos acorde a estos valores.

Progrmación de cliente en python

Debido a que ya existe un programa en python para el control de servomotores (en simulación) y creación de rutinas como se ve en 12 no se creó uno nuevo para la comunicación con el Robonova, solamente se agregó la programación para crear el socket TCP y enviar datos. Para ello se utilizaron las librerías *socket* y *json*. La primera para conectar el cliente (programa en python) a el servidor (ESP32 en el Robonva) y la segunda para crear un archivo .JSON para enviar los datos.

La creación del cliente consiste en la creación de una variable tipo *socket*. Una vez se quiera iniciar la conexión se utiliza la función *connect((ip,port))* de la librería *socket* donde se

introducen como argumentos la dirección IP del servidor (que fue determinada en la sección anterior) y el puerto por el cual se hará la conexión. Cabe notar que para la conexión suceda exitosamente tanto el cliente como el servidor deben estar conectados a la misma red.

Una vez conectado al servidor, se envían las posiciones de los servomotores en la simulación a el ESP32 para que este, con su programación incorporada coloque los servomotores en las posiciones deseadas. Para estandarizar la comunicación y facilitar el envío y recepción de los datos, se crea una biblioteca en el programa del cliente que luego es guardada en un archivo .JSON que, como se explicó anteriormente, al llegar al ESP32 es deserializado y utilizado para el posicionamiento de los servomotores. Para evitar errores durante la comunicación y reducir la utilización de recursos, este archivo .JSON únicamente se genera y se envía cada vez que exista un cambio en la posición de alguno de los servomotores del robot.

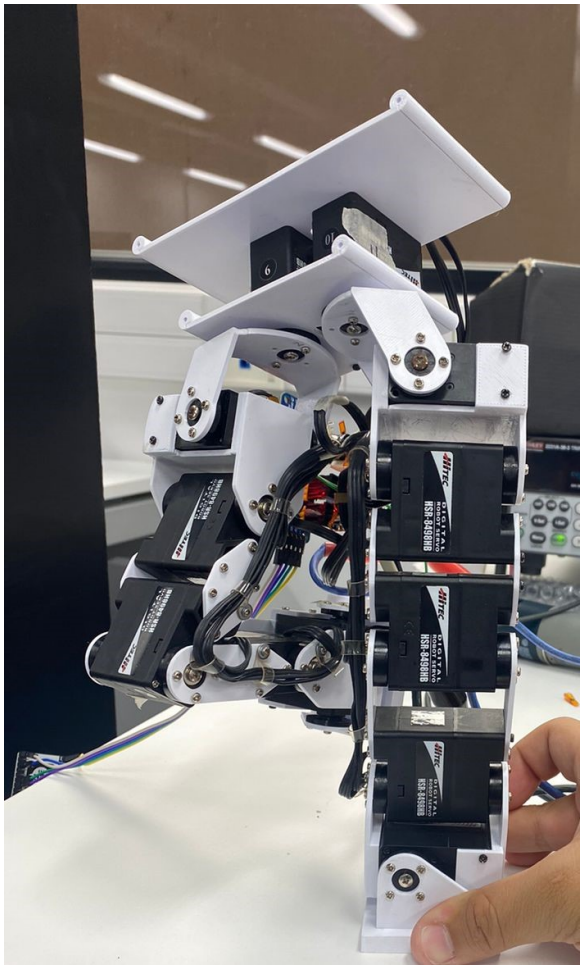

Figura 14: Prototipo de piernas del Robonova siendo controlado programa

Discusión

La principal ventaja de esta conexión entre el programa de control y el Robonova es facilidad de creación, modificación y carga de rutinas al Robonova. Ya que el programa que se encuentra en el ESP32 del nuevo Robonova solamente se encarga de conectarse por WiFi al cliente y controlar los servomotores de a cuerdo a lo que se le envía (es decir, no tiene las rutinas cargadas en su propia memoria), no es necesario conectar el Robonova a la computadora y cargar las nuevas rutinas cada vez que se quiera agregar o modificar una de ellas como era el caso al trabajar con roboBASIC. Además de esto, al encontrarse conectado por Wifi todo el tiempo, también es posible llevar a cabo las demás funciones que se podían hacer en roboBASIC como el control directo de los servomotores sin necesidad de conectar el Robonova o cambiar su programación.

En conclusión, este capítulo se enfocó en como programar y trabajar con el Robonova directamente. Tomando como base todas las funciones de roboBASIC creado por David Buckley, este programa logra llevar a cabo todas las diferentes funciones deseadas para controlar el robot aprovechando los avances en la tecnología y comunicación que se han logrado desde que este salió al mercado.

CAPÍTULO 9

Conclusiones

- Se ha creado una interfaz amigable para el control, programación, monitoreo y la creación de rutinas de la nueva versión del robot humanoide Robonova.
- La conexión en tiempo real al Robonova se logró exitosamente teniendo una latencia baja y, gracias a los servomotores utilizados y programación de control de los mismos, una resolución bastante alta.
- La interfaz creada permite al usuario crear rutinas sencilla y fluidas basado en posiciones específicas deseadas dentro de la misma.
- El agregar una simulación al programa de control del Robonova facilita la logística de programación de rutinas del mismo al no tener que conectarlo con cada edición de rutina.

CAPÍTULO 10

Recomendaciones

- Hacer el modelo dinámico del Robonova para mejorar su emulación de movimientos humanos. Al incluir la dinámica del sistema en el modelo, el Robonova podrá hacer movimientos más complejos de una forma autónoma. Las librerías de pyBullet y Robotics Toolbox incluyen bastantes funciones para facilitar la simulación y análisis de este modelo dinámico.
- Expandir las herramientas de edición de rutinas del programa. Agregar imágenes a la interfaz y botones para de una posición a la siguiente dentro de una rutina. Hacer esta la posición .ªctiva"que pueda ser reemplazada por la que se desee colocar. Esto sería ampleando lo que se tiene en el programa hasta ahora que es solamente agregar o eliminar posiciones al final de la rutina que se está trabajando.

CAPÍTULO 11

Bibliografía

[1] D. Grunberg, R. Ellenberg, I. Kim, J. Oh, P. Oh e Y. Kim, "Development of an Autonomous Dancing Robot," 2010.

[2] S. S. Ekka. "Gait definition, its phases & abnormal gait." Running Time: 59 Section: Health & Fitness. (16 de oct. de 2016), dirección: `https://physiosunit.com/gait-definition-phases-of-cycle-explained/` (visitado 20-05-2023).

[3] M. Zea, *Cinemática de robots de base flotante, fuerzas de contacto y el modelo de fricción de Coulomb*, 2023.

[4] N. Kofinas, "Forward and inverse kinematics for the NAO humanoid robot," 2012.

[5] K. Erbatur y O. Kurt, "Natural ZMP trajectories for biped robot reference generation," *IEEE Transactions on Industrial Electronics*, vol. 56, n.º 3, págs. 835-845, mar. de 2009, ISSN: 0278-0046. DOI: `10.1109/TIE.2008.2005150`. dirección: `http://ieeexplore.ieee.org/document/4633623/` (visitado 22-05-2023).

[6] E. Coumans e Y. Bai, "PyBullet quickstart guide," 2022.

[7] P. Corke y J. Haviland, "Not your grandmother's toolbox – the robotics toolbox reinvented for python," en *2021 IEEE International Conference on Robotics and Automation (ICRA)*, Xi'an, China: IEEE, 30 de mayo de 2021, págs. 11 357-11 363, ISBN: 978-1-72819-077-8. DOI: `10.1109/ICRA48506.2021.9561366`. dirección: `https://ieeexplore.ieee.org/document/9561366/` (visitado 21-05-2023).

[8] Rhoban. "Onshape-to-robot documentation." (2020), dirección: `https://onshape-to-robot.readthedocs.io/en/latest/` (visitado 01-09-2023).

[9] The Qt Company Ltd. "Qt for Python." (2023), dirección: `https://doc.qt.io/qtforpython-6/` (visitado 01-09-2023).

[10] D. Buckley. "RoboBasic-v2.2.61." (1 de ago. de 2008), dirección: `http://davidbuckley.net/DB/RoboNova/RoboBasic-v2.10.htm#Appendix1` (visitado 15-04-2023).

CAPÍTULO 12

Anexos

12.1. Programación en Python

12.1.1. Interfaz de Usuario / Programa Principal

```
from PyQt5.QtWidgets import (QApplication, QMainWindow, QPushButton,
                             QPlainTextEdit, QCheckBox, QDial, QLabel,
                             QWidget, QInputDialog, QFileDialog,
                             QVBoxLayout, QHBoxLayout, QGridLayout)
import sys
import csv
import threading
import pybullet_simulation
import GUI_Functions
import time
from numpy import deg2rad, arange
import os as os
from functools import partial
from scipy import interpolate
pybullet_simulation.servoValues = deg2rad([0,0,-45,0,0,
                                           -60,0,0,0,0,
                                           45,0,0,-60,0,0])
t1 = threading.Thread(target=pybullet_simulation.pb,args=())
t1.start()
time.sleep(2)



global currentCoreo
currentCoreo = ""
```

```
class MainWindow(QMainWindow):

    def __init__(self):
        super().__init__()

        self.p = None
        ########## SENDING AND RECEIVING POSITION BUTTONS ##########
        # Design
        self.GetDataBtn = QPushButton("Get Data")
        self.SendDataBtn = QPushButton("Send Data")
        self.connectCB = QCheckBox("Connect to Robonova")
        # Connect
        self.GetDataBtn.pressed.connect(self.getData)
        self.SendDataBtn.pressed.connect(self.sendData)
        self.connectCB.stateChanged.connect(self.connectRobonva)
        # Customize
        btnFont = self.GetDataBtn.font()
        btnFont.setPointSize(20)
        self.GetDataBtn.setFont(btnFont)
        self.SendDataBtn.setFont(btnFont)
        checkFont = self.GetDataBtn.font()
        checkFont.setPointSize(15)
        self.connectCB.setFont(checkFont)
        # Layout
        layout1 = QHBoxLayout()
        layout1.addWidget(self.GetDataBtn)
        layout1.addWidget(self.SendDataBtn)

        ##################### DIALS #####################
        # Design & Customize
        self.dials = GUI_Functions.creatDials(self,
                                              pybullet_simulation.joints_info)
        # Connect
        self.dials[0].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(0))
        self.dials[1].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(1))
        self.dials[2].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(2))
        self.dials[3].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(3))
        self.dials[4].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(4))
        self.dials[5].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(5))
        self.dials[6].itemAt(0).widget().valueChanged.connect(lambda:
                                                              self.updateDial(6))
```

```
self.dials[7].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(7))
self.dials[8].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(8))
self.dials[9].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(9))
self.dials[10].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(10))
self.dials[11].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(11))
self.dials[12].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(12))
self.dials[13].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(13))
self.dials[14].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(14))
self.dials[15].itemAt(0).widget().valueChanged.connect(lambda:
                                                    self.updateDial(15))
# Layout
dialsLayout = GUI_Functions.dialsLayout(self.dials)
##################### choreography BUTTONS ####################
# Desing
self.newCoreoBtn = QPushButton("New choreography")
self.loadCoreoBtn = QPushButton("Load choreography")
self.addCoreoBtn = QPushButton(
    "Add current position\nto choreography")
self.removeCoreoBtn = QPushButton(
    "Remove last saved\nposition from choreography")
self.clearCoreoBtn = QPushButton("Clear choreography")
self.deleteCoreoBtn = QPushButton("Delete choreography File")
self.playCoreoBtn = QPushButton("Play choreography")
self.smoothTrajectoryCB = QCheckBox("Smooth trajectory")
self.repeatCoreoCB = QCheckBox("Repeat choreography")
#global currentCoreoLabel
self.currentCoreoLabel = QLabel(
    "Current choreography: "+ currentCoreo)
# Customize
self.newCoreoBtn.setFont(btnFont)
self.loadCoreoBtn.setFont(btnFont)
self.addCoreoBtn.setFont(btnFont)
self.removeCoreoBtn.setFont(btnFont)
self.clearCoreoBtn.setFont(btnFont)
self.deleteCoreoBtn.setFont(btnFont)
self.playCoreoBtn.setFont(btnFont)
# Connect
self.newCoreoBtn.pressed.connect(self.newCoreo)
self.loadCoreoBtn.pressed.connect(self.loadCoreo)
self.addCoreoBtn.pressed.connect(self.addCoreo)
```

```
        self.removeCoreoBtn.pressed.connect(self.removeCoreo)
        self.clearCoreoBtn.pressed.connect(self.clearCoreo)
        self.deleteCoreoBtn.pressed.connect(self.deleteCoreo)
        self.playCoreoBtn.pressed.connect(
            partial(self.playCoreo, buttonPressed = True))
        self.repeatCoreoCB.stateChanged.connect(
            partial(self.playCoreo, buttonPressed = False))
        # Layout
        coreoLayout1 = QGridLayout()
        coreoLayout1.addWidget(self.newCoreoBtn, 0, 0)
        coreoLayout1.addWidget(self.loadCoreoBtn, 0, 1)
        coreoLayout1.addWidget(self.addCoreoBtn, 1, 0)
        coreoLayout1.addWidget(self.removeCoreoBtn, 1, 1)
        coreoLayout1.addWidget(self.clearCoreoBtn, 2, 0)
        coreoLayout1.addWidget(self.deleteCoreoBtn, 2, 1)
        coreoLayout1.setSpacing(15)
        #coreoLabelsLayout = QHBoxLayout()
        #coreoLabelsLayout.addWidget(self.currentCoreoLabel)
        mainCoreoLayout = QVBoxLayout()
        mainCoreoLayout.addLayout(coreoLayout1)
        mainCoreoLayout.addWidget(self.playCoreoBtn)
        coreoLayout2 = QHBoxLayout()
        coreoLayout2.addWidget(self.smoothTrajectoryCB)
        coreoLayout2.addWidget(self.repeatCoreoCB)
        mainCoreoLayout.addLayout(coreoLayout2)
        mainCoreoLayout.addWidget(self.currentCoreoLabel)
        mainCoreoLayout.setSpacing(15)

        ######################## TEXT BOX (OTPUTS) ######################
        self.text = QPlainTextEdit()
        self.text.setReadOnly(True)
        self.text.setFont(btnFont)
        tempLayout = QHBoxLayout()
        tempLayout.addLayout(mainCoreoLayout)
        tempLayout.addWidget(self.text)

        ######################## MAIN LAYOUT ######################
        mainLayout = QVBoxLayout()
        mainLayout.addLayout(layout1)
        mainLayout.addWidget(self.connectCB)
        mainLayout.addLayout(dialsLayout)
        mainLayout.addLayout(tempLayout)
        mainLayout.setSpacing(15)
        w = QWidget()
        w.setLayout(mainLayout)
        self.setCentralWidget(w)

############################## FUNCTIONS ##############################
```

```
# TEST FUNCTION (SEND, RECEIVE AND CONNECT)
def getData(self):
    global b
    b = [n for n in pybullet_simulation.servoValues]
    self.message("Current postion saved!")
def sendData(self):
    for i in range(len(b)):
        self.dials[i].itemAt(0).widget().setValue(int(b[i]))
    pybullet_simulation.servoValues = b
    self.message("Loaded saved position to simulation!")
def connectRobonva(self):
    pybullet_simulation.activeConnection = self.connectCB.isChecked()

# DIAL FUNCTIONS
def updateDial(self,a):
    dialValue =self.dials[a].itemAt(0).widget().value()
    valueLabel =self.dials[a].itemAt(1).itemAt(1).widget()
    valueLabel.setText(str(dialValue))
    pybullet_simulation.servoValues[a] = deg2rad(dialValue)
# choreography FUNCTIONS
def newCoreo(self):
    self.x = QFileDialog()
    self.filename = self.x.getSaveFileName(filter="CSV Files (*.csv)")
    if self.filename[0] != "":
        global currentCoreo
        currentCoreo = os.path.basename(self.filename[0])
        #global currentCoreoLabel
        self.currentCoreoLabel.setText(
            "Current choreography: " + currentCoreo)
    else:
        pass

def loadCoreo(self):
    self.x = QFileDialog()
    self.filename = self.x.getOpenFileName(filter="CSV Files (*.csv)")
    if self.filename[0] != "":
        global currentCoreo
        currentCoreo = os.path.basename(self.filename[0])
        #global currentCoreoLabel
        self.currentCoreoLabel.setText(
            "Current choreography: " + currentCoreo)
    else:
        pass
def addCoreo(self):
    with open(currentCoreo,"a") as csvfile:
        writer = csv.writer(csvfile)
        writer.writerow(pybullet_simulation.servoValues)
    self.message("Position saved to choreography!")
```

```
def removeCoreo(self):
    with open(currentCoreo,"r+") as csvfile:
        lines = csvfile.readlines()
        lines.pop()
        csvfile = open(currentCoreo, "w+")
        csvfile.truncate(0)
        csvfile.writelines(lines)
    self.message("Position removed from coreograhy!")

def clearCoreo(self):
    with open(currentCoreo,"r+") as csvfile:
        lines = ""
        csvfile = open(currentCoreo, "w+")
        csvfile.truncate(0)
        csvfile.writelines(lines)
    self.message("Choreography cleared!")
def deleteCoreo(self):
    if os.path.exists(currentCoreo):
        os.remove(currentCoreo)
        self.message("Choreography has been deleted!")
        self.currentCoreoLabel.setText("Current choreography: ")
    else:
        self.message("No choreography selected")


t2stop = threading.Event()
repeating = False
def playCoreo(self, **buttonPressed):
    self.message("Playing choreography...")
    with open(currentCoreo, "r+") as csvfile:
        reader = csv.reader(csvfile)
        rows = []
        for line in reader:
            if not line:
                continue
            rows.append(line)
    dt = 0.01
    if self.smoothTrajectoryCB.isChecked():
        t = len(rows)-1
        x = range(len(rows))
        y = rows
        cs = interpolate.make_interp_spline(x,y, 1)
        xs = arange(0,t,dt)
        rows = cs(xs)
    self.t2 = threading.Thread(target=GUI_Functions.repeatCoreo,
            args=(rows,self.t2stop,self.smoothTrajectoryCB.isChecked(),dt))
```

```
        if buttonPressed["buttonPressed"] == True:
            if (self.repeatCoreoCB.isChecked()
                and not(self.t2.is_alive())) == True:
                self.repeating = True
                self.t2stop.clear()
                self.t2.start()
            else:
                for i in range(len(rows)):
                    coreoPosition = [float(x) for x in rows[i] ]
                    pybullet_simulation.servoValues = coreoPosition
                    if self.smoothTrajectoryCB.isChecked():
                        time.sleep(dt)
                    else:
                        time.sleep(0.1)
                self.message("Choreography done!")

        if (not(self.repeatCoreoCB.isChecked())
            and self.repeating) == True:
            self.repeating = False
            self.t2stop.set()
            self.message("Choreography done!")

    # MISC FUNCTIONS
    def message(self, s):
        self.text.appendPlainText(s)

app = QApplication(sys.argv)

w = MainWindow()
w.show()

app.exec_()
```

12.1.2. Funciones auxiliares a la interfaz de usuario

```
from PyQt5.QtWidgets import (QApplication, QMainWindow, QPushButton,
                             QPlainTextEdit, QCheckBox, QDial, QLabel,
                             QWidget,
                             QVBoxLayout, QHBoxLayout, QGridLayout)
from PyQt5.QtCore import (Qt)
from numpy import rad2deg, ceil
import pybullet_simulation
import time
def addDial(self, name, lower_limit, upper_limit, initial_value, num):
    # Dial Widget
    self.dial = QDial()
    self.dial.setRange(lower_limit, upper_limit)
```

```
        self.dial.setValue(initial_value)
        # Limits layout/widget
        self.LowerLimitLabel = QLabel(str(lower_limit))
        self.UpperLimitLabel = QLabel(str(upper_limit))
        self.ValueLabel = QLabel(str(self.dial.value()))

        labelFont = self.LowerLimitLabel.font()
        labelFont.setPointSize(12)
        self.LowerLimitLabel.setFont(labelFont)
        self.UpperLimitLabel.setFont(labelFont)
        self.ValueLabel.setFont(labelFont)

        LabelsLayout = QHBoxLayout()
        LabelsLayout.addWidget(self.LowerLimitLabel)
        LabelsLayout.addWidget(self.ValueLabel)
        LabelsLayout.addWidget(self.UpperLimitLabel)
        LabelsLayout.setAlignment(Qt.AlignHCenter)
        LabelsLayout.setSpacing(32)

        # Name Widget
        self.dial_name = QLabel(name)
        nameFont = self.dial_name.font()
        nameFont.setPointSize(15)
        self.dial_name.setFont(nameFont)
        self.dial_name.setAlignment(Qt.AlignHCenter)

        # Main Layout
        dial_layout = QVBoxLayout()
        dial_layout.addWidget(self.dial)
        dial_layout.addLayout(LabelsLayout)
        dial_layout.addWidget(self.dial_name)

        return dial_layout

    def creatDials(self, info:list):
        from pybullet_simulation import servoValues
        dial_list = []
        for i in range(len(info)):
            dial = addDial(self,str(info[i][1],"utf-8"),
                           round(rad2deg(info[i][8])),
                           round(rad2deg(info[i][9])),
                           int(rad2deg(servoValues[i])),
                           i)
            dial_list.append(dial)
        return dial_list

    def dialsLayout(LayoutList):
        layout = QGridLayout()
```

```
    i = 0
    rows = 2
    columns = int(ceil(len(LayoutList)/2))
    for j in range(rows):
        for i in range(columns):
            if j < 1:
                layout.addLayout(LayoutList[i],j,i)
            else:
                layout.addLayout(LayoutList[i+columns],j,i)
    return layout

def repeatCoreo(rows,stop_event,smooth,dt):
    while not stop_event.is_set():
        for i in range(len(rows)):
            coreoPosition = [float(x) for x in rows[i] ]
            pybullet_simulation.servoValues = coreoPosition
            if smooth:
                time.sleep(dt)
            else:
                time.sleep(0.1)
```

12.1.3. Simulación en pyBullet

```
import pybullet
import pybullet_data
from numpy import rad2deg
# IP Socket Config
import socket
import json

def pb():
    global servoValues
    global joints_info
    global activeConnection
    activeConnection = False
    s = socket.socket()
    ip = '192.168.122.177'
    port = 8091

    # Connect to simulation
    pybullet.connect(pybullet.GUI)
    pybullet.resetSimulation()
    pybullet.configureDebugVisualizer(pybullet.COV_ENABLE_GUI,0)
    pybullet.resetDebugVisualizerCamera(cameraDistance=0.8,
                                        cameraYaw=45,
                                        cameraPitch=-30,
                                        cameraTargetPosition=[0,0,0.25])
```

```
# Set plane in simulation
pybullet.setAdditionalSearchPath(pybullet_data.getDataPath())
plane = pybullet.loadURDF("plane.urdf")

# Load robot URDF
robot = pybullet.loadURDF("Programacion/FinalVersion/robonova/robot.urdf",
                          [0,0,0.32],useFixedBase=1)

# Get Robot info and Initialize servos
numJoints = pybullet.getNumJoints(robot)
joints_info = []

for i in range(numJoints):
    x = pybullet.getJointInfo(robot,i)
    joints_info.append(x)
joint_number = list(range(numJoints))
joint_dict = {}
for i in joint_number:
    joint_dict[joints_info[i][1].decode("utf-8")] = None
# Initialize Simulation
pybullet.setGravity(0,0,-9.81)
pybullet.setTimeStep(0.0001)
pybullet.setRealTimeSimulation(1)
maxForce = [0.0 for n in servoValues]
#global old_servo_values
old_servo_values = [1000]
connected = False
# Run Simulation
pybullet.setJointMotorControlArray(robot,joint_number,
                                   pybullet.POSITION_CONTROL,
                                   servoValues, maxForce)
while True:
    pybullet.stepSimulation()
    pybullet.setJointMotorControlArray(robot,joint_number,
                                       pybullet.POSITION_CONTROL,
                                       servoValues, maxForce)
    servo2 = []

    if activeConnection == True and connected == False:
        s.connect((ip,port))
        connected = True

    for i in range(numJoints):
        x = servoValues[i]
        servo2.append(x)
    if (old_servo_values != servo2) & activeConnection == True:
        joint_dict.update({'LeftShoulder1':int(rad2deg(servoValues[0])+90),
        'LeftShoulder2':int(rad2deg(servoValues[1]))+90,
```

```
        'RightShoulder1':int(rad2deg(servoValues[2]))+90,
        'RightShoulder2':int(rad2deg(servoValues[3]))+90,
        'LeftWaist':int(rad2deg(servoValues[4]))+90,
        'LeftHip1':int(rad2deg(servoValues[5]))+90,
        'LeftHip2':int(rad2deg(servoValues[6]))+90,
        'LeftKnee':int(rad2deg(servoValues[7]))+90,
        'LeftAnkle1':int(rad2deg(servoValues[8]))+90,
        'LeftAnkle2':int(rad2deg(servoValues[9]))+90,
        'RightWaist':int(rad2deg(servoValues[10]))+90,
        'RightHip1':int(rad2deg(servoValues[11]))+90,
        'RightHip2':int(rad2deg(servoValues[12]))+90,
        'RightKnee':int(rad2deg(servoValues[13]))+90,
        'RightAnkle1':int(rad2deg(servoValues[14]))+90,
        'RightAnkle2':int(rad2deg(servoValues[15]))+90})
        old_servo_values = servo2
        data_json = json.dumps(joint_dict)
        s.sendall(bytes(data_json, "utf-8"))
        print(str(joint_dict))
```

12.2. Programación en Arduino

```
#include <WiFi.h>
#include <ESP32_Servo.h>
#include <ArduinoJson.h>
#include <Wire.h>
#include <Adafruit_PWMServoDriver.h>

//Servo servo1;
//int servoPin1 = 22;
//Servo servo2;
//int servoPin2 = 23;
// Listado de servos
/*
Servo 0 = RightAnkle2
Servo 1 = LeftAnkle2
Servo 2 = RightAnkle1
Servo 3 = LeftAnkle1
Servo 4 = RightKnee
Servo 5 = LeftKnee
Servo 6 = RightHip2
Servo 7 = LeftHip2
Servo 8 = RightHip1
Servo 9 = LeftHip1
Servo 10 = RightWaist
Servo 11 = LeftWaist
Servo 12 = RightShoulder1
```

```
Servo 13 = LeftShoulder1
Servo 14 = RightShoulder2
Servo 15 = LeftShoulder2
*/
const char* ssid = "******";
const char* password =  "*****";
const uint16_t port = 8091;

WiFiServer wifiServer(port);

StaticJsonDocument<512> doc;

int servoValues[] ={90,90,90,90,90,90,90,90,90,90,90,90,90,90,90,90};


// called this way, it uses the default address 0x40
Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver();
// you can also call it with a different address you want
//Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x41);
// you can also call it with a different address and I2C interface
//Adafruit_PWMServoDriver pwm = Adafruit_PWMServoDriver(0x40, Wire);

// Depending on your servo make, the pulse width min and max may vary, you
// want these to be as small/large as possible without hitting the hard stop
// for max range. You'll have to tweak them as necessary to match the servos you
// have!
#define USMIN  625 // This is the rounded 'minimum' microsecond
                   // length based on the minimum pulse of 150
#define USMAX  2385 // This is the rounded 'maximum' microsecond
                    // length based on the maximum pulse of 600
#define SERVO_FREQ 50 // Analog servos run at ~50 Hz updates


int Deg2US ( float grados, bool USMin_bool )
{
  float tasa_conv = ((USMAX - USMIN)/180);
  int ans = tasa_conv*grados;
  if (USMin_bool)
  {
    ans = ans + USMIN;
  }
  else
  {
    ans = ans;
  }

  return(ans);
}
```

```
float US2Deg ( int us )
{
  float tasa_conv = ((USMAX - USMIN)/180);
  float ans = (us - USMIN)/tasa_conv;

  return(ans);
}


void setServoPulse(uint8_t n, double pulse) {
  double pulselength;

  pulselength = 1000000;   // 1,000,000 us per second
  pulselength /= SERVO_FREQ;   // Analog servos run at ~60 Hz updates
  Serial.print(pulselength); Serial.println(" us per period");
  pulselength /= 4096;  // 12 bits of resolution
  Serial.print(pulselength); Serial.println(" us per bit");
  pulse *= 1000000;  // convert input seconds to us
  pulse /= pulselength;
  Serial.println(pulse);
  pwm.setPWM(n, 0, pulse);
}



void setup()
{
  Serial.begin(9600);
  Serial.println("8 channel Servo test!");

  pwm.begin();
  pwm.setOscillatorFrequency(27000000);
  pwm.setPWMFreq(SERVO_FREQ);  // Analog servos run at ~50 Hz updates

  WiFi.begin(ssid, password);
  while (WiFi.status() != WL_CONNECTED) {
    delay(500);
    Serial.println("...");
  }

  wifiServer.begin();

  Serial.print("WiFi connected with IP: ");
  Serial.println(WiFi.localIP());

/*
  // Attach servos
```

```
  servo1.attach(servoPin1, 500, 2400); // min/max Ton 500us 2400us for sg90
  servo2.attach(servoPin2, 500, 2400);

  //debug led
  pinMode(21,OUTPUT);
*/
  delay(10);
}

void loop() {
  WiFiClient client = wifiServer.available();
  if (client) {
    while (client.connected()) {

      String values;
      while (client.available()>0) {
        char c = client.read();
        values += c;
      }
      if(values != NULL){
        exchangeJSON(values);
        Serial.println(int(doc["LeftShoulder1"]));
        //servo1.write(int(doc["LeftShoulder1"]));
        //servo2.write(int(doc["LeftShoulder2"]));
        servoValues[0] = int(doc["RightAnkle2"]);
        servoValues[1] = int(doc["LeftAnkle2"]);
        servoValues[2] = int(doc["RightAnkle1"]);
        servoValues[3] = int(doc["LeftAnkle1"]);
        servoValues[4] = int(doc["RightKnee"]);
        servoValues[5] = int(doc["LeftKnee"]);
        servoValues[6] = int(doc["RightHip2"]);
        servoValues[7] = int(doc["LeftHip2"]);
        servoValues[8] = int(doc["RightHip1"]);
        servoValues[9] = int(doc["LeftHip1"]);
        servoValues[10] = int(doc["RightWaist"]);
        servoValues[11] = int(doc["LeftWaist"]);
        servoValues[12] = int(doc["RightShoulder1"]);
        servoValues[13] = int(doc["LeftShoulder1"]);
        servoValues[14] = int(doc["RightShoulder2"]);
        servoValues[15] = int(doc["LeftShoulder2"]);
      }

      for(int i=0; i<15; i++){
        pwm.writeMicroseconds(i, Deg2US ( servoValues[i], true ));
      }

      delay(10);
    }
```

```
    client.stop();
    Serial.println("Client disconnected");
  }

}

void exchangeJSON(String payload){

  DeserializationError error = deserializeJson(doc, payload);
  if (error) {
    Serial.println(error.c_str());
    return;
  }

  delay(20);
}
```