

**UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA
VALPARAÍSO - CHILE**



**“ τ -HYPERNEAT: RETARDOS DE TIEMPO EN UNA
RED HYPERNEAT PARA APRENDIZAJE DE
CAMINATAS EN ROBOTS CON EXTREMIDADES
MÓVILES”**

OSCAR ANDRÉ SILVA MUÑOZ

**MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO
CIVIL ELECTRÓNICO, MENCIÓN COMPUTADORES**

PROFESOR GUIA: MARÍA JOSÉ ESCOBAR SILVA.

PROFESOR CORRESPONDELENTE: FERNANDO AUAT CHEEIN.

Agradecimientos

Resumen

Abstract

Glosario

Índice general

1.. <i>INTRODUCCIÓN Y OBJETIVOS</i>	3
1.1. OBJETIVOS DEL PROYECTO	3
1.2. TRABAJOS A DESARROLLAR	4
1.3. EVALUACIONES A REALIZAR	5
1.4. RESULTADOS ESPERADOS	5
1.5. TÓPICOS A TRATAR	6
1.6. TRABAJOS RELACIONADOS CON LOS TEMAS A TRATAR	7
2.. <i>ALTERNATIVAS DE SOLUCIÓN PARA EL DESARROLLO DEL PROYECTO</i>	10
2.1. ALTERNATIVAS DE SOLUCIÓN	11
2.1.1. ALTERNATIVA N°1: RETARDOS DE TIEMPO EN UNA RED HYPERNEAT	11
2.1.2. ALTERNATIVA N°2: DEEP LEARNING EN REDES NEURONALES	13
2.1.3. ALTERNATIVA N°3: APRENDIZAJE REFORZADO	15
2.1.4. ENTORNO VIRTUAL DE SIMULACIÓN: VIRTUAL ROBOT EXPERIMENTATION PLATAFORM (V-REP)	17
2.2. ALTERNATIVA SELECCIONADA	18
2.2.1. CRITERIOS DE SELECCIÓN	18
2.2.2. EVALUACIÓN DE LAS ALTERNATIVAS	19
3.. <i>BASE TEÓRICA</i>	23

3.1.	COMPOSITIONAL PATTERN PRODUCING NETWORKS (CPPNs)	23
3.1.1.	DESARROLLO DE PATRONES	24
3.1.2.	COMPLEJIZACIÓN	25
3.1.3.	MÉTODO DE CODIFICACIÓN	26
3.1.4.	EVOLUCIONANDO CPPNs	29
3.2.	NEAT	30
3.3.	CPPN-NEAT	32
3.4.	HYPERNNEAT	32
3.4.1.	MAPEANDO PATRONES ESPACIALES A PATRONES DE CONECTIVIDAD	33
3.4.2.	GENERACIÓN DE PATRONES DE CONECTIVIDAD REGULARES	35
3.4.3.	CONFIGURACIONES DEL SUBSTRATO	36
3.4.4.	POSICIONAMIENTO DE ENTRADAS Y SALIDAS	37
3.4.5.	RESOLUCIÓN DEL SUBSTRATO	39
3.4.6.	EVOLUCIÓN DE LA RED CPPN	39
4..	τ -HYPERNNEAT	42
5..	<i>ROBOTLIB: LIBRERÍA PARA EL MANEJO DE ROBOTS REALES Y SIMULADOS</i>	46
6..	<i>DISEÑO DE PLATAFORMAS ROBÓTICAS EN EL ENTORNO VIRTUAL</i>	52
7..	<i>SIMULACIÓN DE CAMINATAS</i>	70
7.1.	FUNCIONES DE DESEMPEÑO	72
7.2.	ESTRUCTURA BÁSICA DE UN ENTRENAMIENTO	74
7.3.	PROGRAMA DE ENTRENAMIENTO	75
7.4.	RESULTADOS DE LOS ENTRENAMIENTOS	76
7.4.1.	QUADRATOT	77

1. INTRODUCCIÓN Y OBJETIVOS

El Proyecto “ τ -HyperNEAT: Retardos de Tiempo en una Red HyperNEAT para Aprendizaje de Caminatas en Robots con Extremidades Móviles” pretende incorporar conceptos temporales en una red neuronal HyperNEAT incluyendo retardos de tiempo adicionales a los pesos en las conexiones entre neuronas, permitiendo así generar caminatas en robots con distinta cantidad de extremidades móviles (ver Figura 1.1), a través de simulaciones en entornos virtuales, de forma más óptima y obteniendo resultados más cercanos a comportamientos encontrados en la naturaleza.



Fig. 1.1: Robots con distinto número de extremidades móviles.

1.1. OBJETIVOS DEL PROYECTO

Objetivo 1 Proponer un red neuronal usando HyperNEAT que incluya retardos de tiempo en sus conexiones.

Objetivo 2 Desarrollar el software necesario para manejar el entorno de simulación a usar en el trascurso del proyecto.

Objetivo 3 Poner a prueba la nueva red neuronal en tareas de aprendizaje y realizar benchmarks para determinar su desempeño.

Objetivo 4 Usar la nueva red neuronal en tareas de aprendizaje de caminatas en robots con extremidades móviles.

1.2. TRABAJOS A DESARROLLAR

El proyecto se inicia en base a estudios e implementaciones previas de redes NEAT y HyperNEAT para la generación de caminatas en robots con extremidades móviles en entornos virtuales de simulación, con las cuales se obtuvieron resultados exitosos. A partir de esto es que se plantea la incorporación de retardos de tiempo a una red HyperNEAT de forma de implementar τ -HyperNEAT computacionalmente. Luego se debe comparar el desempeño de τ -HyperNEAT versus el desempeño de su predecesor, el cual se espera que sea mejor, para luego entrenar y generar caminatas en los robots. La correcta generación y evolución de caminatas en un entrenamiento está sujeta a una función de desempeño en base a las variables observadas en el robot, por lo que se debe realizar un estudio exhaustivo de cuál es la función de desempeño que mejor describe a una correcta caminata. Para el desarrollo de los entrenamientos de caminatas en los robots se debe implementar un modelo para cada robot en un entorno de simulación con el fin de observar las caminatas generadas y emular correctamente las dinámicas que se presentarían en un entorno real de forma de lograr traspasar posteriormente los resultados obtenidos a los robots reales. Además se debe desarrollar el software necesario para la comunicación con el programa de simulaciones (el cual soporta comunicación por sockets), y que este permita poder exportar directamente el trabajo al entorno real.

1.3. EVALUACIONES A REALIZAR

Una vez obtenidas las caminatas con τ -HyperNEAT resta comparar los resultados con los anteriormente obtenidos solo con HyperNEAT, tanto en el aspecto visual final de las caminatas obtenidas, como además en la evolución de dichas caminatas a lo largo del proceso de entrenamiento medida a través de las variables aplicadas para la calificación del desempeño de manera de validar el mejor funcionamiento de τ -HyperNEAT. Luego se debe comparar cuan influyentes fueron los retardos de tiempo incluidos en la red HyperNEAT observando la estructura y conexiones de la red τ -HyperNEAT finalmente obtenida para obtener las conclusiones del trabajo propuesto. Finalmente se debe evaluar el desempeño final obtenido usando la red final τ -HyperNEAT sobre los robots en el entorno real para comprobar el buen funcionamiento de las caminatas en ellos.

1.4. RESULTADOS ESPERADOS

Al culminar el Proyecto “ τ -HyperNEAT: Retardos de Tiempo en una Red HyperNEAT para Aprendizaje de Caminatas en Robots con Extremidades Móviles”, se espera poder obtener caminatas naturales y armónicas en robots con extremidades móviles de manera más óptima a las obtenidas solo con una red HyperNEAT, y lograr reproducir los mismos resultados en las plataformas robóticas reales emulando correctamente las dinámicas de los sistemas. De manera más general se pretende obtener una red neuronal más robusta y eficiente que permita resolver problemas reales con dependencias temporales. Además se espera generar un software robusto que permita una correcta comunicación con el entorno de simulación a usar para proveer esta herramienta a proyectos futuros en donde se requiera de emular sistemas reales complejos.

1.5. TÓPICOS A TRATAR

Las redes neuronales artificiales (ANNs) son un campo muy importante dentro de la Inteligencia Artificial (IA). El estudio de las redes neuronales ha estado muy en boca durante las últimas décadas y su uso ha sido de gran relevancia para la solución de problemas difíciles de resolver mediante técnicas algorítmicas convencionales. Uno de estos problemas de difícil solución es la generación de caminatas en robots con extremidades móviles.

Con el objetivo de intentar emular movimientos más naturales en los robots estudiados es que investigadores han propuesto técnicas de neuroevolución, lo cual es una forma de aprendizaje de máquina que usa algoritmos evolutivos para entrenar a una red neuronal. Junto con esto y la implementación de algoritmos genéticos en redes neuronales, una de las líneas más prometedoras de la Inteligencia Artificial, se han logrado obtener los resultados esperados para este problema. Sin embargo lo que se busca en esta memoria es profundizar aún más en la solución a este problema e incorporar todos estos conceptos en una red neuronal adicionando variables temporales al modelo incorporando retardos de tiempo en cada una de las conexiones de la red.

Junto con el problema mismo que es la generación de caminatas en robots con extremidades móviles esta la tarea de simular virtualmente el modelo de cada robot, ya que la mayoría de las veces realizar pruebas en plataformas reales es inalcanzable, por elevados costos de adquisición de los equipos; muy poco práctico ya que requiere de una constante y prolongada intervención de personas; o muy peligroso, ya que cualquier problema o error podría incurrir en el deterioro del equipo o inclusive podría atentar contra la seguridad de las mismas personas que realizan los experimentos. Aun es más difícil poder traspasar los resultados del estudio realizado en un modelo simulado virtualmente al modelo real de forma directa. Es por esto que esta memoria contempla la implementación de una herramienta de software que permita trabajar con modelos virtualizados y reales de forma transparente para el usuario, de manera de dar facilidades para la implementación del modelo en un entorno virtual y luego

poder exportar todo el trabajo realizado y los resultados al entorno real con el solo hecho de ajustar los parámetros de trabajo.

Para el trabajo de simulación en el área de la robótica existen variadas opciones con distintos niveles de dificultad y costo de uso dependiendo del público objetivo para el cual está pensado. Es por esto que para el desarrollo de esta memoria se propone el uso de una herramienta de fácil acceso, tanto por el nivel de conocimiento que requiere su uso como su accesibilidad de descarga y sencillo manejo, con el objetivo de que el software a realizar este al alcance de uso de cualquier persona. Esto busca acercar a las personas a trabajar en el área de la robótica incitándolas con herramientas de fácil acceso y manejo.

1.6. TRABAJOS RELACIONADOS CON LOS TEMAS A TRATAR

En el área de redes neuronales que hacen uso de neuroevolución y algoritmos genéticos se puede observar el trabajo realizado por el investigador Kenneth O. Stanley, siendo el primer artículo de interés el que relata el desarrollo de NEAT [2], Neuroevolución a través del Aumento de Topologías, el cual supera en pruebas comparativas a redes con topologías fijas en tareas de aprendizaje reforzado. Stanley asume que el aumento en la eficiencia se debe al uso de un método de cruce entre diferentes topologías, a la clasificación por especies de redes diferenciadas por su topología y cambios en ella, y el crecimiento incremental a partir de una estructura mínima.

Una continuación del desarrollo de NEAT es HyperNEAT [1], Hipercubo basado en Neuroevolución a través del Aumento de Topologías, igualmente desarrollado por Stanley, el cual emplea una codificación indirecta llamada conectivo Compositional Pattern Producing Network (conectivo CPPNs), que puede producir un patrón de conectividad con simetrías y esquemas repetidos interpretado por el patrón espacial generado dentro de un hipercubo. La ventaja de este enfoque es que es posible explotar la geometría de la tarea mediante el mapeo de sus regularidades en la topología de

la red, desplazando con ello la dificultad del problema lejos de la dimensionalidad de este hacia la estructura misma del problema.

En el área de generación de caminatas en robots con extremidades móviles existe una vasta cantidad de investigaciones relacionadas tanto con el uso de HyperNEAT, algoritmos genéticos en general u otro tipo de técnicas que se expondrán a continuación.

Investigadores de la Universidad de Cornell el año 2004 publicaron “Evolving Dynamic Gaits on a Physical Robot” [3], en donde formularon un algoritmo genético para entrenar un controlador de lazo abierto para la generación de una caminata en un robot conformado por dos plataformas Stewart, evolucionando en búsqueda de optimizar su velocidad y su patrón de movimiento garantizando al mismo tiempo el ritmo de estos.

Otros investigadores del Centro de Investigación Ames, de la NASA, el año 2005 publicaron “Autonomous Evolution of Dynamic Gaits with Two Quadruped Robot” [4], en donde relatan cómo han desarrollado un algoritmo de evolución para generar caminatas dinámicas en dos robots cuadrúpedos, OPEN-R y ERS-110 de la marca Sony, midiendo el desempeño de las caminatas con los distintos sensores incorporados en ellos.

En el 2009, el investigador Jeff Clune junto a otros publicaron “Evolving Coordinated Quadruped Gaits with the HyperNEAT Generative Encoding” [5], en donde demuestra cómo es posible desarrollar caminatas en robots cuadrúpedos sin realizar un trabajo manual para resolver el problema usando HyperNEAT.

En el 2011, investigadores de la Universidad de Cornell junto a un investigador de la Universidad de Chile publicaron “Evolving Robot Gaits in Hardware: the HyperNEAT Generative Encoding Vs. Parameter Optimization” [6], en donde presentan la investigación de variados algoritmos para la generación automática de caminatas sobre un robot cuadrúpedo, en donde se comparan dos clases, los de búsqueda local de modelos de movimientos parametrizados y la evolución de redes neuronales artificiales usando HyperNEAT. Aquí concluyeron que las caminatas desarrolladas con Hyper-

NEAT fueron considerablemente mejores a las desarrolladas con los otros métodos de búsqueda local parametrizada, y produjo caminatas casi nueve veces más rápidas que caminatas generadas a mano.

En el año 2013, un grupo de investigadores de la Universidad de Cornell, Universidad de Oslo y Universidad de Wyoming publicaron en conjunto “Evolving Gaits for Physical Robots with the HyperNEAT Generative Encoding: The Benefits of Simulation” [7], en el cual plantean y confirman la hipótesis de que los resultados de las caminatas generadas con HyperNEAT en un entorno simulado superan con creces a las generadas en un entorno real.

Recientemente investigadores de la Universidad de la Coruña han propuesto una extensión del algoritmo NEAT propuesto por Stanley llamado τ -NEAT [8] debido a que NEAT no es siempre fiable cuando existen manejos de variables temporales dentro de la tarea a solucionar debido a la falta de elementos temporales explícitos dentro de la topología de la red NEAT. Es por esta razón que el algoritmo τ -NEAT propuesto incluye la posibilidad de incluir retardos variables en las conexiones de la red NEAT, afectando tanto a las conexiones directas como recurrentes de la red.

El software de simulación que se usará para el desarrollo de esta memoria se llama V-REP [9], Virtual Robot Experimentation Platform, desarrollado por Coppelia Robotics GmbH en Zurich, Suiza, el cual posee versiones tanto pagas como gratuitas. La versión educacional de este software es completamente gratuita y posee todas las funcionalidades del programa completo. Este software posee una API desarrollada en una gran variedad de lenguajes de programación, permitiendo así manejar todos los aspectos del programa y la simulación desde el exterior a través de sockets. Esta API será usada por la herramienta de software a desarrollar para el trabajo con los experimentos tanto en el entorno virtual como en el real.

Con respecto a la herramienta de software que se implementará para manejar transparentemente las plataformas robóticas en el entorno de simulación y entorno real no se conoce implementación alguna disponible, por lo que se cree será una gran contribución a trabajos futuros.

2. ALTERNATIVAS DE SOLUCIÓN PARA EL DESARROLLO DEL PROYECTO

En este capítulo se darán a conocer las alternativas de solución consideradas para llevar a cabo el Proyecto “ τ -HyperNEAT: Retardos de Tiempo en una Red HyperNEAT para Aprendizaje de Caminatas en Robots con Extremidades Móviles”, concentrándose en cómo se logrará la generación de algoritmos de caminata usando herramientas de Inteligencia Artificial (IA) en un entorno de simulación virtual. El programa encargado de la generación de caminatas deberá hacer uso de una herramienta externa para la comunicación con los robots a manipular tanto en el entorno virtual como en el real, tal como se muestra en la Figura 2.1

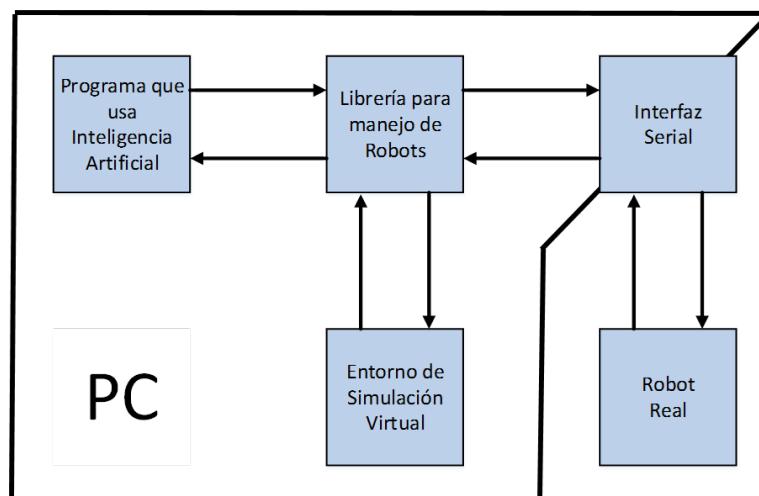


Fig. 2.1: Diagrama de Flujo del Sistema Completo para la Generación de Caminatas. El programa que controla los entrenamientos de generación de caminatas usando IA, a través de la librería para el manejo de robots, podrá comunicarse con un robot virtual dentro del programa de simulación y, por medio de la interfaz serial, comunicarse también con el robot real, para darle instrucciones de movimiento y realizar lecturas de sensores en ambos casos.

Si bien el desarrollo de este proyecto de memoria es bastante acotado debido a lo específico que es el tema, se presentarán otras opciones con las cuales es posible la generación de caminatas en robots con extremidades móviles: Deep Learning en redes neuronales, Reinforcement learning, además del método que usa HyperNEAT para la implementación de τ -HyperNEAT, propuesto para este proyecto. Con respecto al entorno virtual de simulación a usar, se optará por el software V-REP, y no se indagará en otras alternativas ya que se posee un gran manejo y experiencia en el uso de este software. Para la librería encargada de manejar los robots tanto en el entorno virtual como en el real se optará por una implementación en lenguaje C++ al igual que para la implementación de la herramienta de IA.

2.1. ALTERNATIVAS DE SOLUCIÓN

2.1.1. ALTERNATIVA N° 1: RETARDOS DE TIEMPO EN UNA RED HYPERNEAT

HyperNEAT, Hipercubo basado en el Aumento de Topologías, es un generador de codificación que evoluciona redes neuronales artificiales haciendo uso de los principios del algoritmo de Neuroevolución basado en el Aumento de Topologías (NEAT). Es una novedosa técnica para la evolución de redes neuronales de gran escala usando las regularidades geométricas del problema descritas por la red. Esta herramienta actualmente ha sido implementada para trabajo multiplataforma en lenguaje C++.

HyperNEAT básicamente es formado por dos redes neuronales: la red principal llamada substrato, y una segunda red neuronal NEAT usada para generar los pesos de las conexiones en el substrato. La topología del substrato es fija (no evoluciona), y está relacionada con la existencia de restricciones espaciales. El substrato está conformado por un número fijo de neuronas y capas: una capa de entrada, una capa de salida y capas intermedias; donde cada neurona de la red tiene una posición espacial asignada.

Dado un cierto conexionado realizado por la red NEAT en el substrato, se verifica

el desempeño de la red HyperNEAT en una tarea dada y se le asigna a dicha red NEAT una calificación en relación directa con la efectividad de la red HyperNEAT en dicha tarea. Luego de calificar a un número determinado de redes NEAT, el algoritmo de NEAT hace evolucionar dichas redes generando otras nuevas, con el objetivo de alcanzar mejores desempeños al utilizarlas para generar el conexionado del substrato de HyperNEAT. Esto se realiza ciclicamente hasta alcanzar el desempeño deseado de la red HyperNEAT.

A partir de la implementación de HyperNEAT es posible implementar el algoritmo propuesto llamado τ -HyperNEAT, el cual usa una red secundaria NEAT que no solo proporciona el peso de la conexión entre dos nodos, sino que además entrega el tiempo de retardo entre la conexión.

VENTAJAS Y DESVENTAJAS DE LA ALTERNATIVA

De acuerdo a lo antes expuesto, la alternativa de diseñar τ -HyperNEAT mediante la implementación anterior de HyperNEAT presenta las siguientes características:

- Al tener la implementación de HyperNEAT y por ende NEAT en C++, es posible realizar la implementación de τ -HyperNEAT sin mayor complicación.
- Al no usar ninguna librería externa para la implementación de las herramientas antes descritas permite la incorporación de τ -HyperNEAT en cualquier proyecto sobre sistemas operativos Windows, Linux o Mac OS.
- No existe implementación conocida de τ -HyperNEAT en particular, por lo que implementarla sería entrar en áreas inexploradas de la investigación.

Las siguientes alternativas expuestas no son factibles en base a la idea principal de este proyecto, que es precisamente incorporar retardos de tiempo en las conexiones de una red HyperNEAT para la generación de caminatas en robots con extremidades móviles, pero que si cumplirían con el objetivo de la generación de caminatas propiamente tal.

2.1.2. ALTERNATIVA N° 2: DEEP LEARNING EN REDES NEURONALES

Deep Learning [10] es una rama de Machine Learning basado en un conjunto de algoritmos que intentan modelar abstracciones de alto nivel de datos mediante el uso de varias capas de procesamiento con estructuras complejas, o de otra manera compuestas de múltiples transformaciones no lineales. Deep Learning es parte de una familia más amplia de métodos de Machine Learning basado en la representación de los datos de aprendizaje. Una observación se puede representar de muchas maneras. Una de las promesas de Deep Learning está en remplazar funciones realizadas manualmente con eficientes algoritmos para aprendizaje de características y extracción de características jerárquica de manera no-supervisada o semi-supervisada.

La investigación en esta área intenta tomar mejores representaciones y crear modelos para aprender estas representaciones a partir de datos no etiquetados a gran escala. Algunas de las representaciones están inspiradas por los avances en neurociencia y se basan libremente en la interpretación de los patrones de procesamiento y comunicación de información en un sistema nervioso, tales como la codificación neuronal que intenta definir una relación entre varios estímulos y las respuestas neuronales asociados en el cerebro.

El aprendizaje profundo es una clase de algoritmos de aprendizaje automático que:

- Utiliza una cascada de muchas capas de unidades de procesamiento no lineal para la extracción de características y transformación. Cada capa sucesiva utiliza la salida de la capa anterior como entrada. Los algoritmos pueden ser supervisados o sin supervisión y las aplicaciones incluyen análisis de patrones (sin supervisión) y clasificación (supervisado).
- Se basa en el aprendizaje de múltiples niveles de características o de las representaciones de los datos (no supervisado). Características de nivel superior se derivan de características de nivel inferior para formar una representación jerárquica.

- Son parte del campo de aprendizaje automático más amplio de representaciones de aprendizaje de datos.
- Aprende múltiples niveles de representaciones que corresponden a diferentes niveles de abstracción; los niveles forman una jerarquía de conceptos.

Los algoritmos de aprendizaje profundo contrastan con los algoritmos de aprendizaje poco profundo por el número de transformaciones aplicadas a la señal mientras se propaga desde la capa de entrada a la capa de salida. Cada una de estas transformaciones incluye parámetros que se pueden entrenar como pesos y umbrales. No existe un estándar de facto para el número de transformaciones (o capas) que convierte a un algoritmo en profundo, pero la mayoría de investigadores en el campo considera que aprendizaje profundo implica más de dos transformaciones intermedias.

VENTAJAS Y DESVENTAJAS DE LA ALTERNATIVA

De acuerdo a lo antes expuesto, la alternativa de realizar la generación de caminatas mediante la implementación de Deep Learning presenta las siguientes características:

- Permite varios niveles de abstracción que ayudarían a los robots utilizados para generar algoritmos de caminata.
- Generalmente es usado para trabajar con elementos visuales para la extracción de características y no para la generación de caminatas, por lo que su uso en dicha área no ha sido muy explorado.
- Se cree que el aumento de las capas en una red neuronal usando Deep Learning aumente el grado de complejidad de la red volviendo el problema de generación de caminatas aún más complejo.

2.1.3. ALTERNATIVA N° 3: APRENDIZAJE REFORZADO

Cuando hablamos de Aprendizaje Reforzado (RL por su nombre en inglés, Reinforcement Learning) [11], nos referimos a un área de estudio dentro del Aprendizaje de Máquinas, donde el objetivo principal es resolver problemas de decisiones secuenciales modelados como Procesos de Decisión Markovianos (MDP). Sin embargo, por mucho tiempo se ha ampliado el espectro de aplicaciones a áreas como por ejemplo Robótica o Teoría de Control Automático.

En términos simples, el objetivo del aprendizaje por refuerzos es maximizar la recompensa esperada a largo plazo, en un entorno (inicialmente) desconocido mediante la búsqueda de una secuencia óptima de acciones a tomar para cierto problema. Mientras el agente (quien aprende) decide qué acciones tomar, debe hacer un balance de dos objetivos: explotar lo que ya se ha aprendido para evitar caer en soluciones que reporten una menor utilidad, o explorar nuevas soluciones que eventualmente permitan obtener una mayor utilidad a futuro. Este compromiso usualmente se conoce como el dilema de exploración-explotación, el que ha sido extensamente tratado en la literatura, teniendo métodos de exploración indirecta (como exploración de Boltzmann) que explora todo el espacio de estado-acciones al hacer una asignación del tipo probabilista a las diferentes acciones posibles, y métodos de exploración directa que usan información estadística obtenida en experiencias pasadas. Un problema de aprendizaje reforzado, formulado como un MDP está compuesto por (S, A, T, R) donde:

- S : corresponde al conjunto de todos los estados posibles.
- A : denota el conjunto de todas las acciones que el agente puede ejecutar.
- $T: S \times A \times S \rightarrow [0, 1]$ es una función de transición de estado, la que asigna una probabilidad de que el agente en el estado s ejecutando a sea trasladado al estado s' .
- $R: S \times A \rightarrow \mathbb{R}$ corresponde a una función escalar (real) de recompensas.

- $\pi: S \rightarrow A$ es un mapeo de estados a acciones, describe la política (secuencia de acciones) que el agente tomará en cierto estado.

La Figura 2.2 muestra un esquema de la estructura que tiene un problema de aprendizaje reforzado.

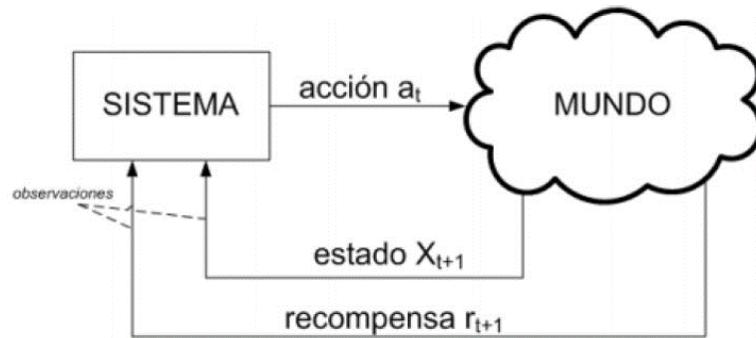


Fig. 2.2: Esquema utilizado en el contexto de aprendizaje reforzado.

VENTAJAS Y DESVENTAJAS DE LA ALTERNATIVA

De acuerdo a lo antes expuesto, la alternativa de realizar la generación de caminatas mediante la implementación de Aprendizaje Reforzado presenta las siguientes características:

- Al tener un aprendizaje constante, la caminata se adapta de forma automática en caso de que la estructura del robot cambie o el tipo de terreno varíe.
- Los métodos tabulares (basados en tablas) son imprácticos en caso de que el espacio de estados discreto sea muy grande (imposible de utilizar en el caso continuo, necesitando aproximador de funciones como redes neuronales).
- El espacio de estados discreto aumenta de forma considerable conforme el número de extremidades o el número de grados de libertad por cada extremidad aumenta.

2.1.4. ENTORNO VIRTUAL DE SIMULACIÓN: VIRTUAL ROBOT EXPERIMENTATION PLATAFORM (V-REP)

V-REP (véase la Figura 2.3) es un simulador compatible con Windows, Mac y Linux, el cual permite el modelado de un sistema completo o de solo ciertas componentes, como sensores, mecanismos, engranajes u otros en poco tiempo. El programa de control de un componente puede estar unido a un objeto ligado o a una escena para modelarlo de manera similar a la realidad. Esta plataforma puede ser usada para controlar partes de hardware, desarrollar algoritmos, crear simulaciones de automatizaciones de fábricas o para demostraciones educativas.

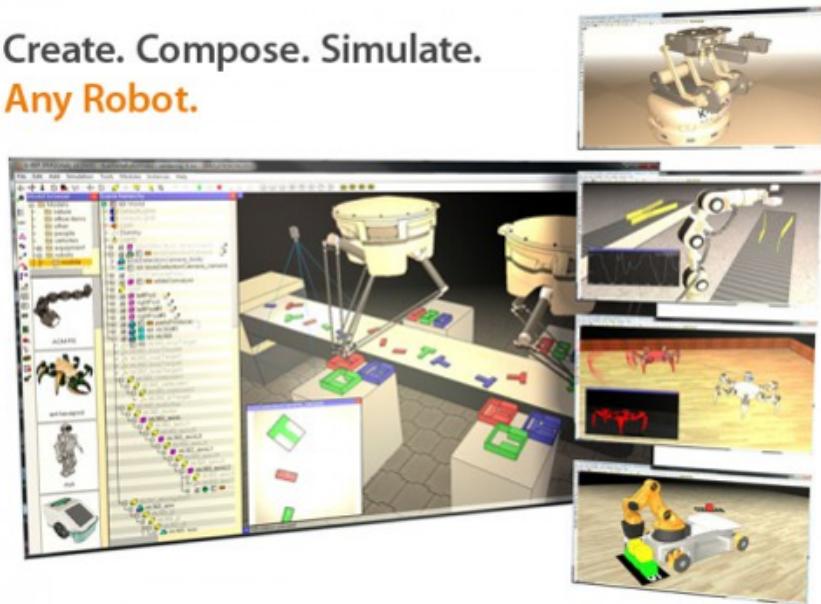


Fig. 2.3: Imagen publicitaria del software de simulación V-REP.

V-REP puede ser usado de dos formas, vía script interno programado en Lua, o vía script externo con la ayuda de la API entregada por el programa, la cual se encuentra en C/C++, Python, Java, Urbi, Lua, Matlab y Octave. Su última versión data del 17 de mayo del 2016.

2.2. ALTERNATIVA SELECCIONADA

En esta sección se seleccionará una de las tres alternativas de solución que se han expuesto para el Proyecto de Titulación en sección anterior, utilizando para ello diversos criterios que permitirán determinar la conveniencia de una u otra alternativa.

2.2.1. CRITERIOS DE SELECCIÓN

De acuerdo a lo expuesto anteriormente, se encontraron tres opciones que permiten generar caminatas en robots con extremidades móviles. Sin embargo lo acotado del problema a solucionar hará que los criterios presentados a continuación estén bastante sesgados para favorecer a la solución principal propuesta con anterioridad. Los criterios a considerar son los siguientes:

- **Cantidad de publicaciones en donde se use la alternativa para la generación de caminatas:** Mientras más publicaciones, información y referencias se tenga del trabajo realizado con la alternativa a elegir será más fácil obtener los conocimientos necesarios para implementar la solución.
- **Simplicidad en la programación:** Puesto que el tiempo para realizar el proyecto es limitado, una programación fácil ayuda a ocupar este recurso en forma óptima. Esto también puede verse afectado por el conocimiento previo que se tenga de la alternativa a elegir.
- **Conocimientos previos de la alternativa a elegir:** Conocimientos previos del tema a trabajar facilitaran en gran medida la implementación de la solución a efectuar.
- **Complejidad computacional de la alternativa a elegir:** Una variable importante a considerar es la complejidad computacional que puede tener la alternativa a elegir, lo cual puede limitar en gran medida a la generación de las caminatas debido a los retardos de procesamiento que se puedan generar.

La ponderación relativa que tienen cada uno de los criterios en la decisión final de la alternativa escogida, de acuerdo a las condiciones, recursos y tiempo del que se dispone son las siguientes:

Criterio de Selección	Porcentaje de Relevancia
Cantidad de bibliografía	30 %
Simplicidad en la programación	15 %
Conocimientos previos	35 %
Complejidad computacional	20 %
Porcentaje Total	100 %

Tab. 2.1: Ponderación de cada uno de los criterios de evaluación

Se ha dado un mayor porcentaje de relevancia a los conocimientos previos que se puedan tener de la alternativa a elegir debido a que daría más facilidad para comprender y solucionar el problema por su mejor manejo. Luego sigue la cantidad de bibliografía disponible, debido a que es fundamental poder informarse de los desarrollos realizados por otros investigadores con el fin de nutrir el trabajo a realizar y para efectuar comparaciones de desempeño. Posteriormente le sigue la complejidad computacional, punto realmente crítico si es que se desea obtener buenos resultados de las simulaciones. Finalmente, la simplicidad en la programación tiene cierta relevancia y se debe tomar en consideración, sin embargo no es un criterio demasiado crítico, ya que se dispone de basto conocimiento de ello.

2.2.2. EVALUACIÓN DE LAS ALTERNATIVAS

Para evaluar cada una de las alternativas se utilizará el siguiente sistema de puntuación:

Sistema de Puntuación				
Muy Deficiente	Deficiente	Aceptable	Bueno	Óptimo
0.1 - 0.2	0.3 - 0.4	0.5 - 0.6	0.7 - 0.8	0.9 - 1

Tab. 2.2: Sistema de puntuación utilizado para evaluar las alternativas

La puntuación de cada alternativa se detalla a continuación.

Retardos de tiempo en una red HyperNEAT (τ -HyperNEAT)

- **Cantidad de publicaciones en donde se use la alternativa para la generación de caminatas: Puntuación = 0.9** (Se pueden encontrar un gran número de publicaciones del tema).
- **Simplicidad en la Programación: Puntuación = 0.8** (Se posee un buen manejo de programación en C++ para cualquiera de las alternativas, pero al tener mayor manejo del tema se facilita aún más la programación).
- **Conocimientos previos de la alternativa a elegir: Puntuación = 0.9** (Se posee conocimientos bastos en el tema debido a trabajos realizados previamente).
- **Complejidad computacional de la alternativa a elegir: Puntuación = 0.7** (La complejidad computacional se ve perjudicada por tratarse de dos redes involucradas en el cálculo de la alternativa, sin embargo no perjudica mayormente al problema en sí).

Deep learning en redes neuronales

- **Cantidad de publicaciones en donde se use la alternativa para la generación de caminatas: Puntuación = 0.5** (Cantidad de bibliografía promedio).
- **Simplicidad en la Programación: Puntuación = 0.7** (Se posee un buen manejo de programación en C++ para cualquiera de las alternativas, pero no se posee

gran manejo del tema).

- **Conocimientos previos de la alternativa a elegir: Puntuación = 0.4** (Se poseen conocimientos regulares del tema).
- **Complejidad computacional de la alternativa a elegir: Puntuación = 0.8** (No posee gran complejidad computacional manteniendo baja la cantidad de capas de la red).

Aprendizaje reforzado

- **Cantidad de publicaciones en donde se use la alternativa para la generación de caminatas: Puntuación = 0.4** (Se encuentran un numero de publicaciones bajo el promedio).
- **Simplicidad en la Programación: Puntuación = 0.5** (Se posee un buen manejo de programación en C++ para cualquiera de las alternativas, pero no se posee ningún manejo del tema, complicando la tarea de programar los algoritmos).
- **Conocimientos previos de la alternativa a elegir: Puntuación = 0.6** (Se posee muy poco conocimiento del tema).
- **Complejidad computacional de la alternativa a elegir: Puntuación = 0.4** (Debido a la gran complejidad de los robots por su cantidad de grados de libertad implica que exista un espacio de estados discretos muy grande volviendo la solución impráctica).

En la Tabla 2.3 se evalúa cada una de las alternativas, de acuerdo a la ponderación que tiene cada criterio. Ella muestra, que con una puntuación total del 84.5 %, la opción más conveniente es la de Retardos de tiempo en una red HyperNEAT (τ -HyperNEAT), luego con un 55.5 % la de Deep learning en redes neuronales y finalmente con un 48.5 % la de Aprendizaje reforzado. **Con todo, la alternativa selec-**

cionada para la generación de caminatas en robots con extremidades móviles es Retardos de tiempo en una red HyperNEAT (τ -HyperNEAT).

Criterio de Selección	Puntuación		
	τ -HyperNEAT	Deep L.	Reinforcement L.
Cantidad de bibliografía	0.9	0.5	0.4
Simplicidad en la programación	0.8	0.7	0.5
Conocimientos previos	0.9	0.4	0.6
Complejidad computacional	0.7	0.8	0.4
Puntuación Total	84.50 %	55.50 %	48.50 %

Tab. 2.3: Tabla de Evaluación de cada alternativa

3. BASE TEÓRICA

En este capítulo se presentará una introducción teórica de la herramienta de neuroevolución HyperNEAT y sus componentes, necesario para la implementación de τ -HyperNEAT.

3.1. COMPOSITIONAL PATTERN PRODUCING NETWORKS (CPPNs)

Un concepto base en la codificación en la naturaleza es que un pequeño número de genes pueden codificar un gran número de subestructuras dentro de un fenotipo a través de la reutilización de genes. En biología, los genes en el ADN representan estructuras extremadamente complejas con miles de millones de partes interconectadas. Sin embargo el ADN no posee miles de millones de genes, sino que de algún modo solo 30 mil genes codifican todo el cuerpo humano.

La reutilización de genes se hace posible debido a que una estructura o fenotipo puede poseer un gran número de patrones, y estos patrones un sin número de regularidades, tal como ocurre en la naturaleza. De no existir regularidades no se hace posible reproducir distintas partes de una estructura a partir de la misma información, perdiendo gran ventaja en la codificación.

En el siguiente sección se introducirán importantes características del desarrollo de patrones.

3.1.1. DESARROLLO DE PATRONES

Identificar las características generales de los patrones presentes en la naturaleza es un prerequisito importante para describir como esos patrones pueden ser generados algorítmicamente. A continuación se mencionarán características generales de patrones observados en organismos de la naturaleza que también pueden ser observados en fenotipos evolucionado artificialmente.

- **Repetición** Múltiples instancias de la misma subestructura es un sello distintivo de los organismos biológicos. Desde las células de todo el cuerpo hasta las neuronas del cerebro, las mismas estructuras se repiten una y otra vez en un único organismo.
- **Repetición con variaciones** Frecuentemente estructuras se encuentran repetidas pero no de forma completamente idéntica. Esto se ve de forma frecuente en toda la naturaleza, como por ejemplo en las vertebras de una columna o en los mismos dedos de una mano, cada una de sus componentes posee la misma estructura pero con distintas variaciones.
- **Simetría** A menudo las repeticiones ocurren a través de las simetrías, como cuando los lados derecho e izquierdo del cuerpo son idénticos, produciéndose una simetría bilateral.
- **Simetría imperfecta** Mientras que un tema simétrico general es observable en muchas estructuras biológicas, muchas veces no son perfectamente simétricas. Tal simetría imperfecta es una característica común de repetición con variaciones. El cuerpo humano es simétrico en general, pero no es equitativo en ambos lados; algunos órganos solo aparecen en uno de los lados, y un lado es generalmente dominante sobre el otro.
- **Regularidades elaboradas** Durante muchas generaciones, regularidades son a menudo elaboradas y mucho mas explotadas, como por ejemplo las aletas de

los peces con simetría bilateral temprana que con el tiempo se convirtieron en los brazos y manos de mamíferos.

- **Preservación de regularidades** Durante generaciones, determinadas regularidades son estrictamente preservadas. Simetrías bilaterales no producen fácilmente simetrías de tres vías, y animales cuadrúpedos raramente producen crías con distinto numero de extremidades.

Usando esta lista, fenotipos y linajes producidos por codificaciones artificiales pueden ser analizados en base a características presentes naturalmente, dando una indicación de si una codificación particular esta capturando propiedades y capacidades esenciales de un desarrollo natural.

La siguiente sección describe un proceso mediante el cual los patrones representados por un conjunto de genes pueden llegar a ser cada vez más complejos.

3.1.2. COMPLEJIZACIÓN

EL proceso de complejización permite a la evolución descubrir fenotipos más complejos de los que sería posible descubrir a través de la optimización de un conjunto fijo de genes.

En la búsqueda de la solución a un problema particular, cuya dimensión es desconocida a priori, mientras más dimensiones tenga el espacio de solución seleccionado, más difícil se hace descubrir esta solución. En otras palabras, soluciones más complejas son más difíciles de evolucionar que otras mas simples. Es por esto que se busca reducir la complejidad del espacio de búsqueda mediante la codificación de un fenotipo complejo en un genotipo de dimensiones significativamente menores.

La clave para que la evolución pueda superar el problema de la complejidad es que no se inicia la búsqueda en un espacio de la misma complejidad que la solución final. Nuevos genes son ocasionalmente añadidos al genoma, permitiendo a la evolución complejizar funciones por sobre el proceso de optimización. La complejización permite a la evolución comenzar con fenotipos simples partiendo por un espacio de

búsqueda dimensionalmente más pequeño para trabajar sobre este de manera incremental, opuesto a la idea de trabajar directamente a partir de sistemas más elaborados desde el comienzo.

Nuevos genes comúnmente aparecen a través de duplicación de otros genes, que es un tipo especial de mutación en donde uno o más genes de los padres son copiados en un genoma hijo más de una vez. El principal efecto de la duplicación de genes es el incremento de la dimensionalidad del genotipo, con lo que se podrán representar patrones fenotípicos cada vez más complejos. Por lo tanto, complejización y la codificación con reutilización de genes trabajan conjuntos para producir fenotipos complejos.

3.1.3. MÉTODO DE CODIFICACIÓN

Con el fin de poder codificar un fenotipo determinado es que ayuda concebirlo como una distribución de puntos en un espacio cartesiano multidimensional. Visto de esta manera, un fenotipo puede ser descrito como una función de dimensión n , donde n es el número de dimensiones en el mundo físico. Para cada coordenada, la presencia o ausencia de un punto es una salida de la función que define el fenotipo. La figura 3.1 muestra como un fenotipo bidimensional puede ser generado a través de una función de dos parámetros

Considere una distribución de puntos o marco de coordenadas en un eje cartesiano de izquierda a derecha, donde la concentración de puntos aumenta hacia la derecha. Las coordenadas de estos puntos a lo largo del eje podrían estar definidas simplemente por una función $f(x) = x$. Si consideramos una distribución de puntos en donde estos estuvieran concentrados hacia el punto medio entre ambos extremos y que disminuyen su concentración a medida que se acercan a los extremos, tal como se presentaría una simetría bilateral, esta podría ser descrita simplemente como una función Gaussiana $g(x) = \frac{1}{\sigma\sqrt{2\pi}}e^{-(x-\mu)^2/2\sigma^2}$. Si se quisiera representar segmentación, podría hacerse a través de funciones periódicas. La función $h(x) = \sin(x)$ podría representar puntos repetidos de forma equidistante a lo largo de un eje de coordenadas.

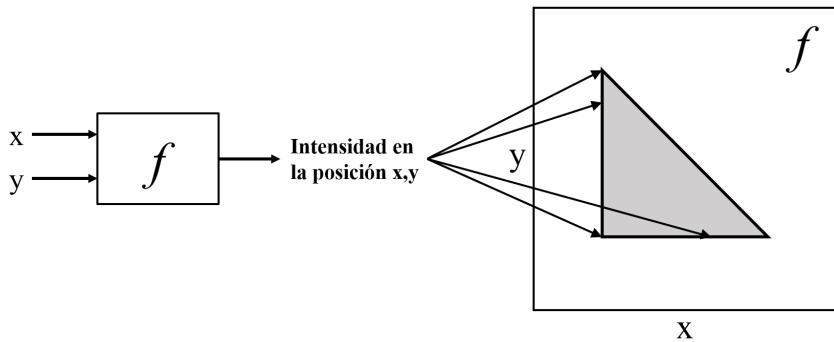


Fig. 3.1: Una función produce un fenotipo. La función f toma los argumentos x e y , las cuales son coordenadas en un espacio bidimensional. Cuando todas las coordenadas son dibujadas con una intensidad correspondiente a la salida de la función f , el resultado es un patrón, el cual puede ser visto como un fenotipo cuyo genotipo es f . En este ejemplo, f produce un fenotipo triangular.

Diferentes marcos de coordenadas podrían interactuar en el proceso de desarrollo para producir patrones con regularidades mas complejas. Del mismo modo, marcos representados por funciones pueden interactuar y componer regularidades complejas. Por ejemplo, una simetría bilateral con segmentación a lo largo de un eje de coordenadas de izquierda a derecha puede producir dos grupos de segmentos con polaridades opuestas. Esta distribución podría ser representada fácilmente poniendo como entrada de una función periódica la salida de una función simétrica, realizando una composición de funciones, como se muestra en la figura 3.2. Así también, una serie de composiciones de funciones pueden unirse para formar una nueva composición de manera de producir marcos de coordenadas mas elaborados.

“Compositional Pattern Producing Networks” (CPPNs) es un método de codificación que permite describir directamente relaciones estructurales de una topología a través de una composición de funciones.

Una manera natural de representar una composición de funciones es a través de un grafo de funciones interconectadas, como se muestra en la figura 3.3. Es así como el sistema de coordenadas inicial de una estructura puede ser provisto como entrada del grafo. El siguiente nivel de nodos puede ser visto como una descripción inicial del primer sistema de coordenadas de dicha estructura. Niveles mas elevados de nodos

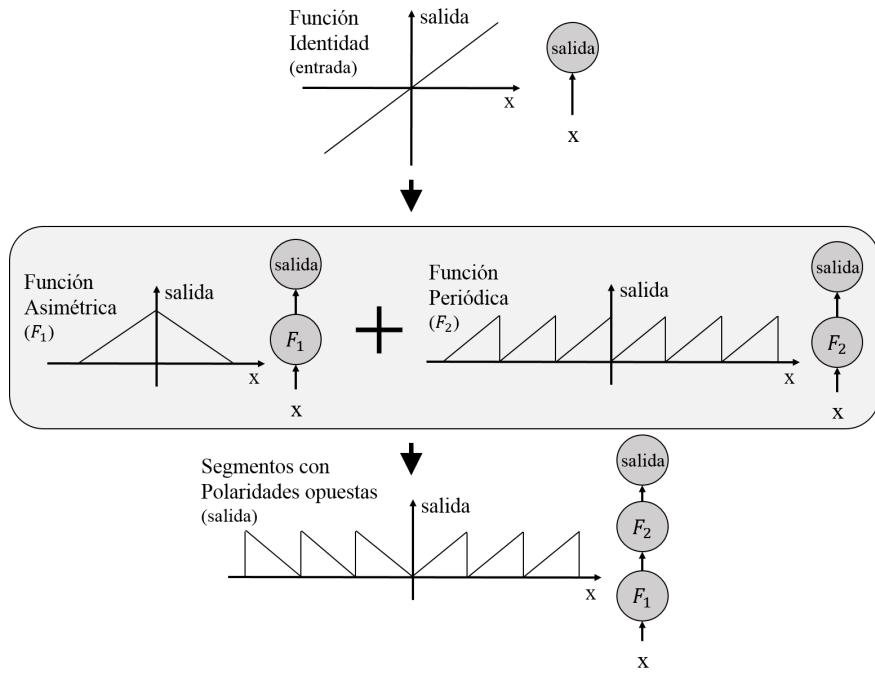


Fig. 3.2: Composición de funciones. Este ejemplo ilustra como una simple composición de funciones en una dimensión puede producir patrones con múltiples regularidades. Una representación en forma de red de cada composición es mostrada a la derecha de los gráficos de cada función. La entrada asimétrica inicial x a la entrada de una composición de una función simétrica (F_1) y una función periódica (F_2) produce dos grupos de segmentos con polaridades opuestas.

establecerán sistemas de coordenadas cada vez más refinados. Finalmente las salidas finales corresponderán a las transformaciones de todas las capas anteriores, entregando una codificación del sistema de coordenadas provisto.

Es interesante observar que un grafo de dicha composición es muy similar a una ANN con topología arbitraria. La única diferencia entre ambas es que las ANNs generalmente usan funciones sigmoides (y a veces funciones Gaussianas) como función de activación de cada nodo, mientras que un grafo puede usar cualquier variedad de funciones canónicas en cada nodo.

Finalmente podemos referirnos a “Compositional Pattern Producing Networks” (CPPNs) para describir una composición de funciones en forma de grafo que busca reproducir patrones regulares.

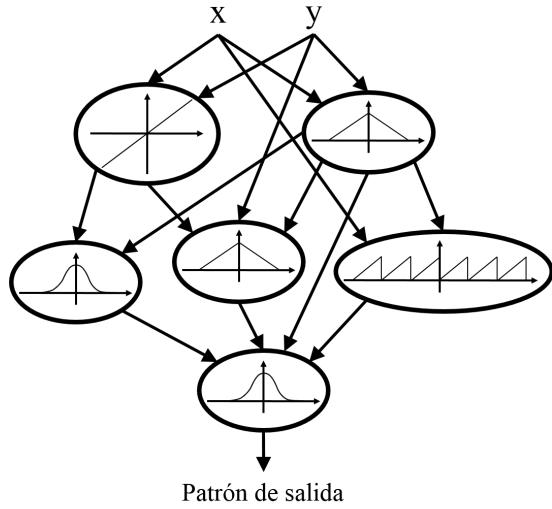


Fig. 3.3: Composición de funciones en forma de grafo. El grafo determina cual función esta conectada con cual. A cada conexión se le asigna un peso, el cual multiplica a la salida de la función del nodo entrante. Si múltiples conexiones entran a una misma función, esta recibe como entrada la suma de las salidas de cada una de las funciones entrantes multiplicadas por sus respectivos pesos de conexión. Se debe tener en cuenta que la topología no tiene restricciones y puede representar cualquier relación posible. Esta representación es similar al formalismo de las ANNs tradicionales con funciones de activación y topologías arbitrarias.

3.1.4. EVOLUCIONANDO CPPNs

Casualmente CPPNs y ANNs son prácticamente lo mismo desde una perspectiva estructural. Muchos métodos para evolucionar topologías y pesos de conexión en ANNs ya existen, por lo que es posible extender de manera fácil estos métodos para evolucionar CPPNs añadiendo pequeños cambios. También hay métodos para evolucionar programas genéticos, los cuales pueden ser muy similares a CPPNs.

Anteriormente se describió una propiedad esencial de la evolución en la naturaleza que describe la gradual complejización del genoma. La idea principal es iniciar con un pequeño genoma al cual se le añaden nuevos genes en el trascurso de la evolución. En CPPNs esto significa que se añadirán nuevas conexiones o nodos de funciones a la red, permitiendo así complejizar la red para poder establecer regularidades fundamentales en los principios de su desarrollo para luego hacerlas más elaboradas a lo largo de la evolución.

El método a usar para evolucionar las CPPNs será “NeuroEvolution of Augmenting Topologies” (NEAT). NEAT es capaz de evolucionar ANNs increíblemente complejas a lo largo de generaciones, y sobrellevar los desafíos que conlleva evolucionar una gran población de redes de diversas topologías mediante el uso de marcas históricas. Ya que NEAT comienza trabajando sobre redes pequeñas y solo expande su espacio de solución cuando logra un beneficio de esta expansión, se vuelve capaz de encontrar ANNs significativamente más complejas, de forma contraria a métodos que evolucionan topologías fijas.

A pesar de que NEAT originalmente fue diseñado para evolucionar ANNs, solo se requiere de pequeñas modificaciones para evolucionar CPPNs, ya que como se mencionó, estas son muy similares.

3.2. NEAT

Esta sección explica el fundamento del método de neuroevolución NEAT, su funcionamiento, y como este puede evolucionar CPPNs.

A diferencia de muchos otros métodos para evolucionar ANNs (métodos de neuroevolución) existentes que tradicionalmente evolucionan redes con topologías fijas o generadas arbitrariamente de forma aleatoria, NEAT es la primera que inicia la evolución a partir de una población de pequeñas y simples redes neuronales, aumentando su complejidad a lo largo de las generaciones, destacando un comportamiento cada vez más sofisticado.

Antes de describir como extender el algoritmo NEAT para evolucionar CPPNs es necesario describir las tres ideas principales de las cuales NEAT se basa. Primero, con el fin de permitir que las estructuras de las redes se vuelvan más complejas a través de las generaciones, se hace necesario un método para realizar un seguimiento de qué gen es cuál. De otra forma no es claro en posteriores generaciones cuál individuo es compatible con cual, o cómo estos genes pueden ser combinados para producir genes hijos. NEAT soluciona este problema asignando una marca histórica única a cada

nueva pieza de la estructura de la red que aparezca a través de mutaciones estructurales. La marca histórica es un número asignado a cada gen correspondiente a su orden de aparición durante el curso de la evolución. Estos números son heredados durante el entrecruzamiento sin cambios, y permitiendo a NEAT realizar entrecruzamientos de genes sin la necesidad de un costoso análisis topológico. De esta forma, genomas con diferentes estructuras o tamaños mantienen la compatibilidad a lo largo de la evolución, solucionando el problema planteado anteriormente de comparación de diferentes topologías en una población en evolución.

Segundo, NEAT separa en especies a los individuos de una población, por lo que los organismos compiten principalmente dentro de sus propios nichos en lugar de con toda la población. De esta manera las innovaciones topológicas en los genes son protegidas y se les da tiempo para que optimicen sus estructuras antes de competir con otros nichos dentro de la población. NEAT utiliza las marcas históricas en genes para determinar a qué especies pertenecen diferentes individuos.

Tercero, otros sistemas que evolucionan redes con topologías con nodos interconectados con pesos o costos en sus conexiones comienzan una evolución con una población con topologías aleatorias. NEAT al contrario, comienza con una población uniforme de redes simples sin capas intermedias, difiriendo solo en los pesos de sus conexiones inicializados aleatoriamente. Diversas topologías se acumulan gradualmente durante la evolución, lo que permite diversos y complejos patrones fenotípicos para ser representados. No está indicado un límite del tamaño que puede llegar a alcanzar una topología. Así, NEAT puede iniciar la evolución desde una estructura mínima, y aumentar su tamaño sobre un número determinado de generaciones.

Nuevas estructuras son introducidas como sucesos de mutaciones estructurales, y solo sobrevivirán si es que resultan ser beneficiosas a través del cálculo de su desempeño. De este modo, otra ventaja de la complejización es que NEAT busca a través de soluciones dimensionalmente mas pequeñas, reduciendo significativamente el número de generaciones necesarias para encontrar la solución, y asegurando que la red no se volverá mas compleja de lo necesario. En efecto, entonces, NEAT realiza una búsqueda

da de una solución compacta y apropiada a través del incremento de la complejidad de estructuras ya existentes.

3.3. CPPN-NEAT

CPPN-NEAT es una extensión de NEAT que permite evolucionar CPPNs. Mientras que redes en el método NEAT original solo incluían nodos intermedios con funciones de activación sigmoides, los nodos en CPPN-NEAT son creados asignándoles aleatoriamente una función de activación proveniente de un grupo canónico determinado de funciones (que incluye por ejemplo la función Gaussiana, Sigmoide, y funciones periódicas). Además se define la función de distancia de compatibilidad para determinar si dos redes pertenecen a la misma especie, la cual incluye la información de por cuantas funciones de activación difieren dos individuos distintos. Esta característica permite que en la separación de individuos en especies se tome en cuenta el número de funciones de activación que difieren entre individuos. Ya que CPPN-NEAT es una mejora de un preexistente efectivo método de neuroevolución, proporciona una base fiable para el estudio de la evolución de formas cada vez mas complejas, como ANNs de gran escala o otras estructuras tipo grafo con simetrías y patrones complejos de repetición. La siguiente sección presenta un enfoque que permite a CPPN representar y evolucionar simplemente este tipo de redes.

3.4. HYPERNEAT

Si CPPNs son evolucionadas para representar patrones de conectividad, el problema se dirige en encontrar la mejor interpretación de sus salidas para describir efectivamente la estructura que se intenta representar. EL patrón de dos dimensiones mostrado en la figura 3.1 presenta un desafío: ¿ Cómo pueden patrones espaciales describir la conectividad de una red ? Esta sección explica como patrones espaciales generados por CPPNs pueden mapear de manera natural patrones de conectividad de una red

mientras que al mismo tiempo dan solución a problemas asignados a dicha red desde su propia dimensionalidad.

3.4.1. MAPEANDO PATRONES ESPACIALES A PATRONES DE CONECTIVIDAD

Existe un mapeo eficaz entre los patrones espaciales y de conectividad que pueden elegantemente explotar las geometrías de las estructuras. La idea principal es entregar a la entrada de una red CPPN las coordenadas de dos puntos que definen una conexión en lugar de entregar solo la posición de un único punto como se mostró en la figura 3.1. La salida de la red CPPN es interpretado como el *peso* de la conexión en lugar de la intensidad de un punto. De esta forma las conexiones pueden ser definidas en términos de la ubicación de sus nodos terminales, teniendo así en cuenta la geometría de la red.

CPPN en efecto calcula una función $\text{CPPN}(x_1, y_1, x_2, y_2) = \omega$ de cuatro dimensiones, en donde el primer nodo se encuentra en la coordenada (x_1, y_1) y el segundo en (x_2, y_2) . A partir de esta función se retorna el peso de cada una de las conexiones entre cada nodo en la red. Por convención, una conexión no se realiza si la magnitud del peso resultante de la función CPPN, que puede ser positiva o negativa, se encuentra por debajo de un umbral mínimo ω_{min} . Las magnitudes de los pesos por sobre este umbral son escalados entre cero y una magnitud máxima ω_{max} , definida para cada red como uno sobre el número de nodos de la capa formada por el menor número de ellos. De esta forma, los patrones producidos por la red CPPN pueden representar cualquier topología de red.

Por ejemplo, considere una malla de 5×5 nodos. A cada nodo se le asigna una coordenada correspondiente a la posición de este dentro de la malla (nombrada como *substrato* en la figura 3.4), donde la coordenada $(0, 0)$ corresponde al centro de la malla. Asumiendo que estos nodos y sus posiciones son dadas *a priori*, un patrón de conectividad entre nodos en un espacio bidimensional es producido por una red CPPN que toma dos coordenadas (fuente y destino) como entrada, y que retorna como salida

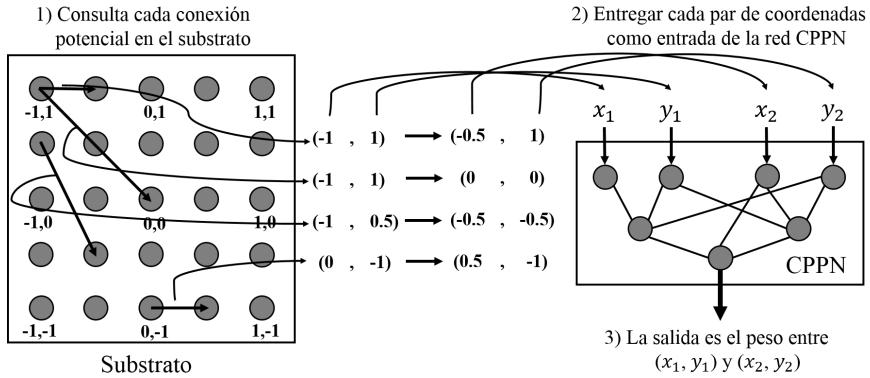


Fig. 3.4: Interpretación del patrón geométrico de conectividad de un hipercubo. En una malla de nodos, llamada *substrato*, a cada uno de los nodos se le asigna una coordenada tal que el nodo central se encuentra en el origen. (1) Cada conexión potencial es consultada para determinar si existe, y de existir, cuál sería su peso; la línea negra en la parte inferior derecha del substrato indica una posible conexión a ser consultada. (2) Por cada consulta, la red CPPN toma como entrada las coordenadas de los dos puntos terminales y (3) entrega como salida el peso de la conexión entre ellos. Luego de que todas las conexiones han sido determinadas, un patrón de conexiones y pesos de conexión resultan en función de la geometría del substrato. De esta forma, la red CPPN produce patrones de regularidad de conexiones en el espacio.

el peso de la conexión entre estos nodos. La red CPPN calcula de esta manera entonces cada conexión potencial en la malla. Ya que los pesos de las conexiones son de un modo una función de las posiciones de los nodos de fuente y destino, la distribución de pesos en las conexiones a lo largo de la malla exhibirá un patrón que estará en función de la geometría del sistema de coordenadas de la malla.

El patrón de coordenadas producido por una red CPPN será llamado *substrato* de forma de distinguirlo verbalmente de la red CPPN misma, el cual tendrá su propia topología interna.

Ya que la red CPPN en este caso representa una función de cuatro dimensiones, el patrón de conectividad bidimensional expresado por la red CPPN es isomorfo al patrón espacial presente en un hipercubo de cuatro dimensiones. Esta observación es importante ya que esto significa que patrones espaciales con simetrías y regularidades corresponden a patrones de conectividad también con dichas simetrías y regularidades. Así, tal como CPPNs generan patrones espaciales regulares, por extensión se

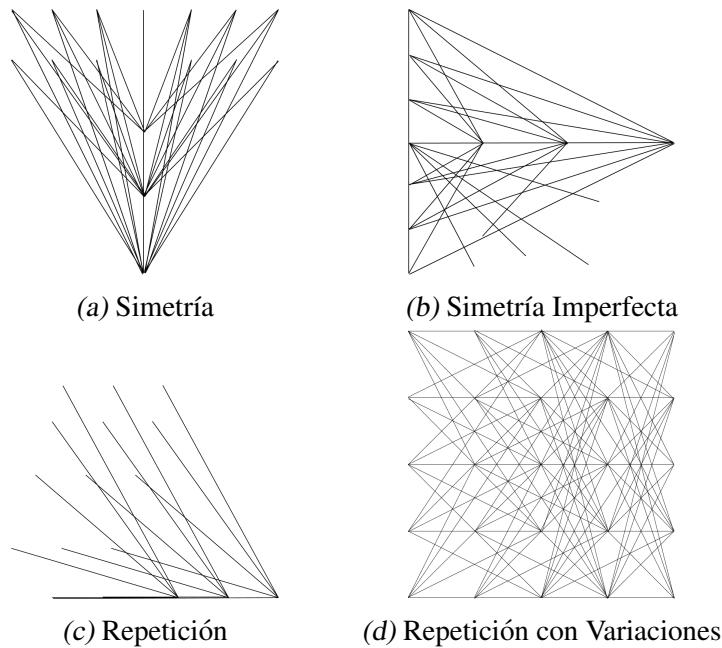


Fig. 3.5: Patrones de conectividad producidos por CPPNs. Estos patrones, producidos a través de la evolución, exhiben importantes casos de patrones de conectividad: (a) simetría bilateral, (b) simetría imperfecta, (c) repetición, y (d) repetición con variaciones. Que estos patrones sean generados fácilmente y representados de forma compacta sugiere el poder de esta codificación.

puede esperar que puedan producir patrones de conectividad con las correspondientes regularidades. La siguiente sección muestra dichas capacidades.

3.4.2. GENERACIÓN DE PATRONES DE CONECTIVIDAD REGULARES

Subestructuras descubiertas en la red CPPN producen importantes regularidades en las conexiones del substrato utilizando como entradas los valores de las coordenadas de los puntos en el eje x y en el eje y . Así por ejemplo, una simetría a lo largo del eje x puede ser simplemente descubierta mediante la aplicación de una función simétrica (por ejemplo Gaussiana) a las coordenadas x_1 y x_2 (figura 3.5a).

Una simetría imperfecta es otro importante patrón observado. Una red CPPN puede producir simetrías imperfectas por la composición de dos funciones simétricas juntas a un marco de coordenadas asimétrico. De esta manera, la red CPPN puede producir diferentes grados de simetrías imperfectas como el ejemplo de la figura 3.5b.

Otro patrón importante es el de repetición, particularmente el de repetición con variaciones. Tal como funciones simétricas producen simetrías, funciones periódicas, como la función seno, produce repeticiones (figura 3.5c). Patrones con variaciones son producidos por una composición de una función periódica con un marco de coordenadas sin repeticiones, tal como su propio eje (figura 3.5d). Así, simetría, simetría imperfecta, repetición y repetición con variaciones son regularidades que pueden ser representadas de forma compacta y por lo tanto fácilmente descubiertas por CPPNs.

La siguiente sección ahondará en las configuraciones que puede adoptar el substrato.

3.4.3. CONFIGURACIONES DEL SUBSTRATO

La disposición de los nodos que CPPN conecta en el substrato puede tomar otras formas distintas a la malla plana mostrada en el ejemplo demostrativo de la figura 3.4 y en la figura 3.6a. Configuraciones de substratos diferentes son probablemente más adecuadas para distintos tipos de problemas.

Por ejemplo, CPPNs también pueden producir patrones de conectividad tridimensionales por la representación espacial de patrones de la función $\text{CPPN}(x_1, y_1, z_1, x_2, y_2, z_2)$ en un hipercubo de seis dimensiones (figura 3.6b), tal como es teóricamente la topología de los cerebros biológicos.

También es posible restringir configuraciones de substrato a determinadas topologías con el fin de aprender acerca de su viabilidad de forma aislada. Por ejemplo, Churchland [12] llama a una simple capa bidimensional de neuronas conectada a otra capa bidimensional un “*state-space sandwich*”. El sándwich es una estructura tridimensional restringida en la cual una capa solo puede establecer conexiones en una sola dirección hacia otra capa. Por lo tanto, debido a esta restricción, se puede expresar por una única función $\text{CPPN}(x_1, y_1, x_2, y_2)$ de dimensión cuatro, donde (x_2, y_2) se interpreta como una locación en la capa de destino en lugar de estar en el mismo plano de coordenadas de (x_1, y_1) . De esta manera, CPPNs pueden hacer búsqueda de patrones útiles dentro de un substrato con configuración state-space sandwich (figura 3.6c).

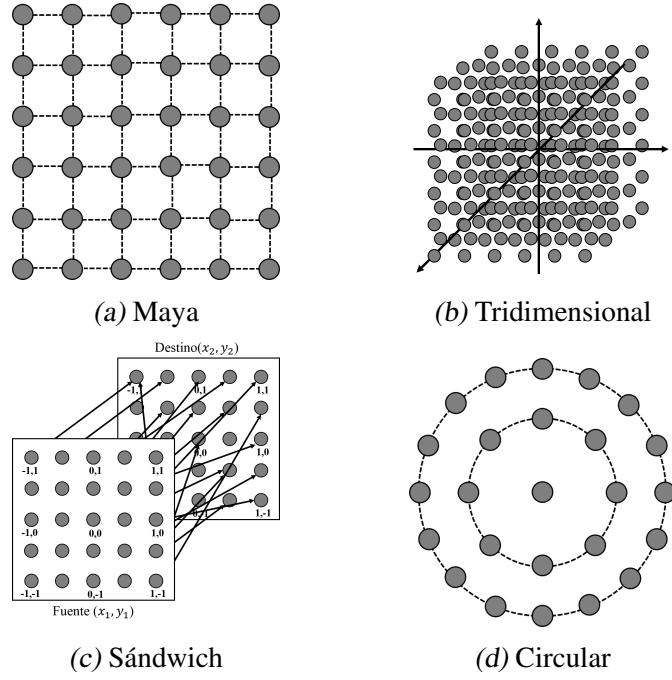


Fig. 3.6: Alternativas de configuración del Substrato. Estas figuras muestran (b) una configuración tridimensional de nodos centrados en $(0,0,0)$, (c) una configuración sándwich de dos capas en la cual una capa fuente de neuronas se conecta directamente a una capa destino, y (d) una configuración circular. Diferentes configuraciones son probablemente mas adecuadas para problemas con diferentes propiedades geométricas.

Finalmente, los nodos en una capa no tienen que estar necesariamente distribuidos en forma de malla. Por ejemplo, nodos dentro de un substrato que controla entradas radiales, como por ejemplo un robot con sensores de distancia situados a su alrededor, podrían estar dispuestos mejor en un esquema con simetría radial, como se muestra en la figura 3.6d, de modo que el patrón de conectividad se pueda establecer con un sistema de coordenadas polares perfecto.

3.4.4. POSICIONAMIENTO DE ENTRADAS Y SALIDAS

Parte de la configuración del substrato es la determinación de cuales nodos son entradas y cuales salidas. La flexibilidad de asignar entradas y salidas a coordenadas específicas en el substrato crea una oportunidad de explorar relaciones geométricas ventajosamente.

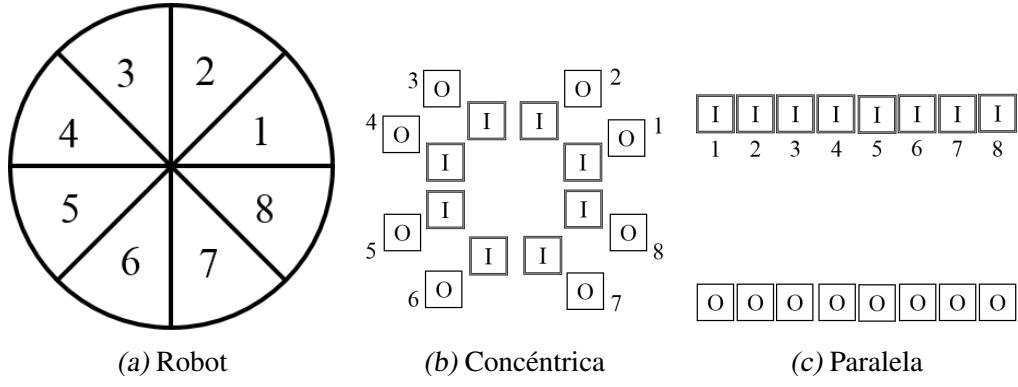


Fig. 3.7: Posicionando entradas y salidas. Un robot (a) es descrito con ocho sensores lidar como entradas de estímulos, y ocho actuadores para respuestas motoras situados de forma radial equiangularmente distanciados. En (b), las entradas (de etiqueta *I*, correspondientes a sensores lidar) y salidas (de etiqueta *O*, correspondientes a actuadores) están situadas en la misma diagonal en los ocho sectores indicados en (a). En (c), entradas y salidas son situadas horizontales y paralelamente unas de las otras en dos filas. Ambos arreglos crean una relación geométrica entre cada entrada y su correspondiente salida. De esta forma es posible dar ventaja a la evolución desde el comienzo.

En muchas aplicaciones de ANNs, las entradas son dibujadas por un grupo de sensores en un arreglo geométrico en el espacio. A diferencia de los algoritmos tradicionales de aprendizaje de ANNs que no son conscientes de tal geometría, una red CPPN si sera consciente de ello y puede usar esa información a su favor.

Mediante la disposición de las entradas y las salidas en una configuración específica en el substrato, regularidades en la geometría pueden ser explotadas por la codificación. Esto permite ser creativos para probar diferentes configuraciones geométricas que aporten distintas ventajas. Por ejemplo, la figura 3.7 describe dos métodos en los cuales las entradas y las salidas de un robot circular pueden ser configuradas, en donde ambas crean una oportunidad de explotar diferentes formas de relaciones geométricas.

En un arreglo, los sensores de la periferia del robot son situados en un círculo centrado en el origen del substrato, y las salidas forman una circunferencia concéntrica alrededor de este (figura 3.7b). De esta forma, si la red CPPN descubre simetrías radiales o bilaterales, esta puede usar este sistema de coordenadas para crear un patrón repetitivo que capture regularidades en las conexiones entre las entradas y las sali-

das. Un arreglo alternativo es situar las entradas y las salidas en dos líneas paralelas en donde la posición de cada entrada y salida está correlacionada con el ángulo de ubicación en el robot en cada una de sus filas respectivamente (figura 3.7c). De esa forma, la evolución puede explotar las similitudes de las posiciones horizontales de las entradas y salidas. Ambos métodos representan correspondencia a través de una regularidad geométrica diferente.

A través del arreglo de las neuronas en una configuración específica en el substrato, regularidades en la geometría pueden ser explotadas por la codificación.

3.4.5. RESOLUCIÓN DEL SUBSTRATO

En contraposición con la codificación de un patrón específico de conexiones entre un conjunto específico de nodos, una red CPPN en efecto codifica un concepto de conectividad general, es decir, los fundamentos de las relaciones matemáticas producen un patrón particular. La consecuencia es que la misma red CPPN puede representar un concepto equivalente en diferentes resoluciones o densidades de nodos. La figura 3.8 muestra dos conceptos de conectividad en diferentes resoluciones.

Para substratos con neuronas, la importante implicancia de esto es que la misma funcionalidad de la ANN puede ser generada en distintas resoluciones. Sin más evolución, una red CPPN previamente evolucionada puede ser usada para especificar el patrón de conectividad de un mismo tipo de substrato con una resolución mayor, generando de este modo una solución para un mismo problema con una resolución más alta.

3.4.6. EVOLUCIÓN DE LA RED CPPN

La propuesta de este método base para la implementación de τ -HyperNEAT es evolucionar una red CPPN usando NEAT para generar un patrón de conectividad en otra red principal que posee características geométricas propias del problema a solucionar. Este método base es llamado HyperNEAT ya que se usa a NEAT para evolucionar.

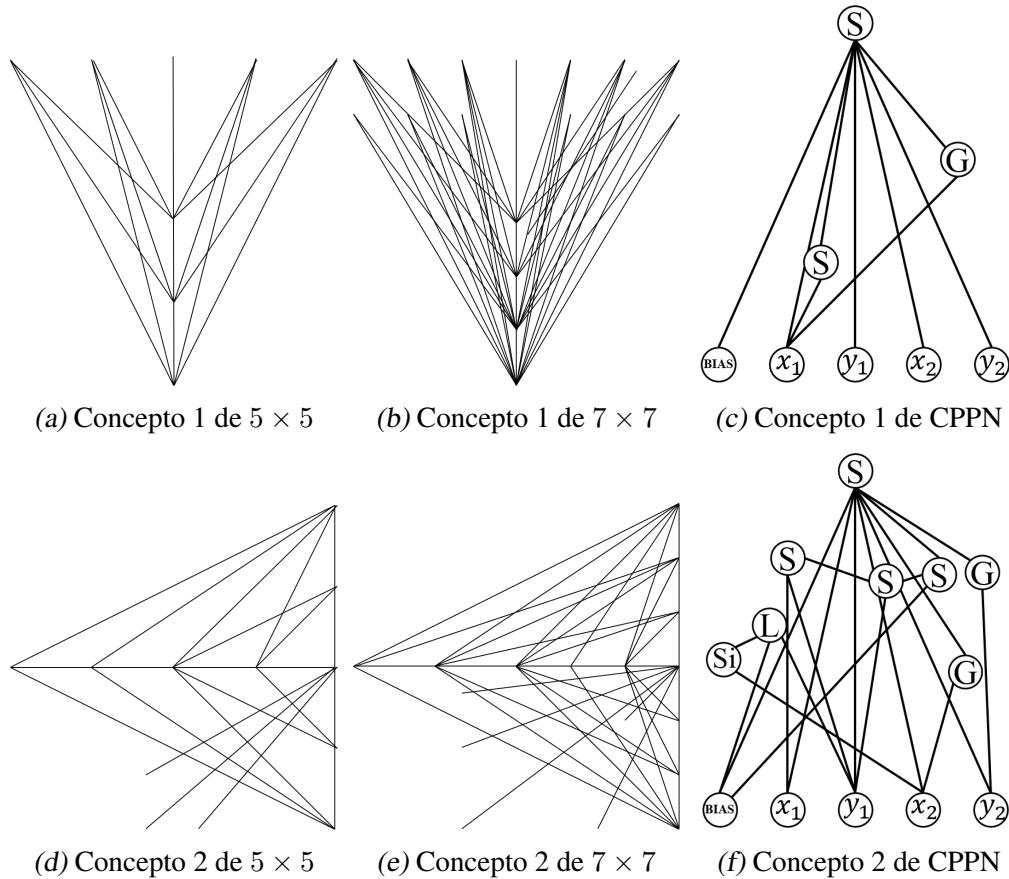


Fig. 3.8: Conceptos de conectividad equivalentes en diferentes resoluciones del substrato.

Se describen dos conceptos de conectividad generados en el proceso de evolución. La red CPPN que genera el primer concepto a resoluciones de 5×5 (a) y 7×7 (b) es mostrado en (c). La red CPPN en (f) genera de forma similar el segundo concepto en ambas resoluciones (d) y (e). Esta ilustración demuestra que CPPNs representan un concepto matemático en lugar de una sola estructura. Así, la misma red CPPN puede producir patrones con el mismo fundamento conceptual en diferentes resoluciones del substrato en diferente densidad de nodos. Las funciones de activación de las redes CPPN mostradas están denotadas por *G* para funciones *Gaussianas*, *S* para *Sigmoides*, *Si* para *Senos*, y *L* para *Lineales*.

nar una red CPPN que busca representar patrones espaciales en un hiperespacio. Cada punto en el patrón, delimitado por un hipercubo, es interpretado como una conexión en un grafo dimensionalmente más pequeño.

El esquema básico del algoritmo del método HyperNEAT se muestra a continuación:

1. Elegir una configuración de substrato, es decir, el posicionamiento de cada nodo y la asignación de nodos de entrada y salida.
2. Inicializar una población de redes CPPN con pesos asignados de forma aleatoria.
3. Repetir hasta encontrar una solución:
 - a) Por cada miembro de la población de redes CPPN:
 - (I) Usar la red CPPN para determinar el peso de cada posible conexión en el substrato. Si el valor absoluto de la salida de la red CPPN sobrepasa la magnitud de un umbral, se crea la conexión con el peso dado por la salida de la red CPPN escalada apropiadamente.
 - (II) Se usa el substrato como una ANN en la tarea a resolver para determinar su desempeño y asignar el resultado a la red CPPN.
 - b) Reproducir las redes CPPN acorde al método NEAT para producir una nueva población de CPPNs correspondientes a una nueva generación.

En efecto, como HyperNEAT incorpora nuevas conexiones y nodos a la estructura de la red CPPN, esta descubriendo nuevas dimensiones globales de variaciones en los patrones de conectividad a través del substrato. Al principio es posible descubrir una simetría global, para luego descubrir un concepto mas elaborado de las regularidades del sistema. Cada nuevo nodo y conexión en el substrato representan una nueva manera en que un patrón a generar pueda variar dándose nuevas regularidades. Así, HyperNEAT es una propuesta para evolucionar patrones de conectividad a gran escala en ANNs.

4. τ -HYPERNEAT

Una vez comprendido el funcionamiento del método de neuroevolución HyperNEAT es posible extenderlo para implementar el nuevo método τ -HyperNEAT propuesto. Este método poseerá la misma estructura que su predecesor, con una configuración definida del substrato y una población inicial de organismos CPPN de topología básica únicamente diferenciados por la aleatoriedad de la asignación de los pesos de sus conexiones. Sin embargo, la topología básica inicial de estos organismos será diferente que para el caso de HyperNEAT. Además de que una red CPPN entregue el peso como resultado de la consulta de una conexión entre dos nodos dentro del substrato, entregará un segundo valor correspondiente al porcentaje de retardo (con respecto a un retardo máximo) asignado a la conexión entre dichos nodos. El retardo de conexión es implementado en el substrato por medio de un *buffer*, el cual poseerá un largo proporcional al retardo.

La figura 4.1 muestra un esquema del método τ -HyperNEAT propuesto, el cual es muy similar al esquema de la figura 3.4. Una red CPPN se encarga de generar un patrón de conectividad entre todos los nodos ubicados en el substrato, tomando como entradas las coordenadas de los nodos (fuente y destino), y retornando el peso y el porcentaje de retardo en la conexión entre estos nodos. De esta forma, la red CPPN calcula entonces cada conexión potencial en el substrato. Tal como ocurre en el método HyperNEAT en donde la distribución de los pesos en las conexiones a lo largo del substrato exhibe un patrón en función de las geometrías del sistema de coordenadas del substrato, los retardos también estarán distribuidos en función de este.

En este caso, la red CPPN calcula una función $\text{CPPN}(x_1, y_1, x_2, y_2) = [\omega, \tau]$ de cuatro dimensiones, al igual que en el caso de HyperNEAT, en donde el primer nodo

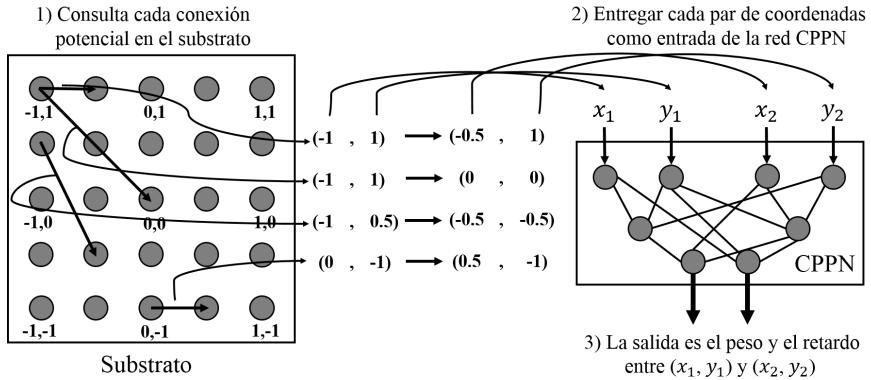


Fig. 4.1: Interpretación del patrón geométrico de conectividad de un hipercubo usando τ -HyperNEAT. Al igual que como funciona el método HyperNEAT (1) cada conexión potencial es consultada para determinar si esta existe, y de existir, cual sería su peso y su retardo asociado. (2) Por cada consulta, la red CPPN toma como entrada las coordenadas de los dos puntos terminales y (3) entrega como salida el peso y el retardo de la conexión entre ellos. Luego de que todas las conexiones han sido determinadas, un patrón de conexiones, pesos y retardos de conexión resultan en función de la geometría del substrato.

se encuentra en la coordenada (x_1, y_1) y el segundo en (x_2, y_2) . Sin embargo, a diferencia de su predecesor, a partir de esta función se retorna un vector de largo dos, compuesto por el peso y el retardo de cada una de las conexiones entre cada nodo en la red. Al igual que en HyperNEAT, una conexión no se realiza si la magnitud del peso resultante de la función CPPN, que puede ser positiva o negativa, se encuentra por debajo de un umbral mínimo ω_{min} . Las magnitudes de los pesos por sobre este umbral son escalados entre cero y una magnitud máxima definida para la red. El resultado correspondiente al retardo no tiene ninguna implicancia al momento de decidir si una conexión es o no factible, y solo entrega información adicional del comportamiento de cada conexión. El porcentaje de retardo obtenido es multiplicado por el retardo máximo τ_{max} (número entero positivo correspondiente al tamaño máximo asignable al buffer de retardo) dado para la red, y es aproximado al número entero más próximo, obteniendo como resultado el tamaño del buffer de retardo. La figura 4.2 muestra como es el comportamiento del buffer a cada iteración de la red.

Cada nodo del substrato recibirá como entrada los valores de salida de cada uno

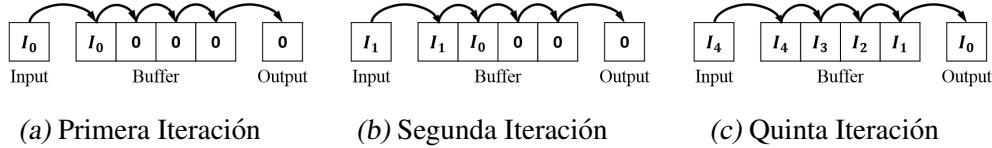


Fig. 4.2: Funcionamiento del buffer de retardo. Cada nodo dentro del substrato posee un buffer, con un largo dado por una red CPPN, que retardará el flujo de información a través de la red. En las imágenes se muestra un ejemplo de un nodo asignado con un buffer de tamaño 4, al que se le asignan por defecto todas sus casillas con valor cero. En la primera iteración (a), la suma de todas las entradas al nodo es pasada por una función de activación previamente asignada dando como resultado el primer valor de entrada I_0 al buffer, colocándose al inicio de este y desplazando todos los demás valores contenidos en el buffer hacia la derecha. Luego el resultado de salida del nodo para la primera iteración es el valor de la última casilla del buffer antes del desplazamiento (cero). En la segunda iteración (b), se entrega una nueva entrada I_1 al buffer volviendo a desplazar cada valor contenido en el buffer hacia la derecha, obteniéndose como salida el valor de la última casilla del buffer antes del desplazamiento (cero aun). Al llegar a la quinta iteración (c), se ingresa un nuevo valor a la entrada del buffer y se obtiene a la salida el valor I_0 ingresado en la primera iteración.

de los demás nodos conectados a él, multiplicados por sus respectivos pesos. Estos valores son sumados para posteriormente pasar el resultado por la función de activación del nodo e ingresar al inicio del buffer, desplazando los demás valores contenidos en él hacia la derecha. El valor que queda fuera del buffer producto del desplazamiento se convierte en la salida final del nodo.

Todos estos retardos a lo largo de la red permitirán incorporar variables temporales en la solución del problema a solucionar, incorporando dinámica al sistema. Finalmente, el esquema básico del algoritmo del método τ -HyperNEAT se muestra a continuación:

1. Elegir una configuración de substrato, es decir, el posicionamiento de cada nodo, la asignación de nodos de entrada y salida, y establecer un retardo máximo.
2. Inicializar una población de redes CPPN con pesos asignados de forma aleatoria.
3. Repetir hasta encontrar una solución:
 - a) Por cada miembro de la población de redes CPPN:

- (I) Usar la red CPPN para determinar el peso y el retardo de cada posible conexión en el substrato. Si el valor absoluto de la salida correspondiente al peso de la red CPPN sobrepasa la magnitud de un umbral, se crea la conexión con el peso escalado apropiadamente y el porcentaje de retardo dado por la salida de la red CPPN.
 - (II) Se usa el substrato como una ANN en la tarea a resolver para determinar su desempeño y asignar el resultado a la red CPPN.
- b) Reproducir las redes CPPN acorde al método NEAT para producir una nueva población de CPPNs correspondientes a una nueva generación.

Ya definidos el método HyperNEAT y el método propuesto τ -HyperNEAT es posible realizar pruebas de desempeño en la tarea de generación de caminatas en robots con extremidades móviles, pero para esto es necesario diseñar una herramienta de comunicación que permita al programa que usa alguno de estos métodos de neuroevolución comunicarse con el programa de simulación en donde se ejecutarán los entrenamientos de los robots en la generación de caminatas. Además esta herramienta debe permitir una comunicación con los robots reales para posterior al entrenamiento poder recrear las caminatas efectuadas en el entorno virtual del programa de simulación. En el capítulo siguiente se explicará el diseño e implementación de esta herramienta.

5. ROBOTLIB: LIBRERÍA PARA EL MANEJO DE ROBOTS REALES Y SIMULADOS

RobotLib es una herramienta diseñada para la comunicación e interacción con entornos de simulación y entornos reales de manera transparente y sencilla. RobotLib esta implementada en lenguaje C/C++ y funciona a modo de librería externa, entregando a disposición del usuario los recursos necesarios para poder obtener y entregar información a todos los entornos de trabajo (simulados o reales), inclusive de manera simultanea, de una forma simple y segura.

RobotLib esta compuesta por un conjunto de clases que se dividen en dos grupos, *componentes* del entorno y *controladores*. Los componentes del entorno son los objetos a manipular, como piezas, motores o sensores. Los controladores son objetos que permiten al usuario ejecutar acciones con los componentes del entorno, como por ejemplo posicionar piezas dentro de un entorno virtual, asignar posiciones a motores, o obtener lectura de sensores. Así, para que un usuario pueda manipular un componente de algún entorno de trabajo debe crear un objeto controlador para dicho entorno y un objeto correspondiente al tipo de componente, y además, indicarle al objeto controlador el objeto componente que se desea manipular a través de él. También es posible que un mismo componente sea controlado en distintos entornos de forma simultanea, creando tantos objetos controladores como sean necesarios para cada entorno, creando el objeto componente que se desea manipular, e indicarle a cada objeto controlador el componente a manipular. Esta última estrategia es muy útil cuando se desea, por ejemplo, manipular un robot en un entorno virtual y en el entorno físico real de forma simultanea. A continuación se explicará de forma breve cada una de las clases fun-

damentales que componen RobotLib. Para utilizar RobotLib esta debe ser descargada desde el repositorio <https://github.com/osilvam/RobotLib.git>.

Los objetos controladores implementados son dos: el objeto *RobotVREP* y *USB2Dynamixel*. *RobotVREP* es el controlador implementado para manipular objetos dentro de software de simulación VREP [9], el cual fue el software elegido para realizar los entrenamientos de generación de caminatas en los robots con extremidades móviles. *USB2Dynamixel* es el controlador implementado para controlar los motores de los robots a utilizar, los cuales son de la marca Dynamixel [13], a través del dispositivo de comunicación serial *USB2Dynamixel* diseñado exclusivamente para dichos motores.

Los objetos componentes implementados son tres: el objeto *Object*, *Joint* y *CollisionObject*, en donde los últimos dos objetos heredan del primero. El objeto *Object* es la base de cualquier objeto existente en un experimento, y almacena un *UniqueObjectId* asignado al objeto. El *UniqueObjectId* es un número único asignado a cada objeto al momento de ser creado, y permite que cada controlador pueda asociar dicho número con el número identificador interno respectivo de cada objeto, sin la necesidad de que un objeto deba almacenar tantos números identificadores como controladores lo estén controlando. En un programa de simulación un *Object* puede representar, por ejemplo, un obstáculo dentro del escenario de simulación, y usando las funciones disponibles en *RobotVREP* es posible obtener su posición, orientación y velocidad, y asignarle una nueva posición y orientación. En un entorno real un *Object* puede representar, por ejemplo, una IMU (del inglés *Inertial Measurement Unit*), que con el procesamiento adecuado puede entregar, a través del controlador respectivo, su posición, orientación, velocidad y aceleración. Para hacer uso de cualquier *Object* en algún entorno (simulado o real) solo debe indicarse al controlador respectivo que dicho *Object* estará bajo su control, haciendo uso de la función *Controlador.addObject(Object *)* implementada en cada controlador¹, en donde el argumento entre paréntesis corresponde al puntero del objeto a controlar. Funciones similares como *addJoint(Joint *)* y *addCollisionObject(CollisionObject *)* son usadas para trabajar con objetos *Joint* y *CollisionObject*

¹ No esta implementado en el controlador *USB2Dynamixel* debido al no uso de sensores físicos.

respectivamente con el controlador RobotVREP; y addMotor(Joint *, int) con el controlador USB2Dynamixel, en donde el segundo argumento corresponde al ID del motor dynamixel físico. Una vez asignado el Object al controlador es posible, por ejemplo, obtener la posición de dicho objeto en coordenadas cartesianas dentro del entorno de simulación (V-REP) usando la función *Controlador:getObjectPosition(Object *)*.

El objeto Joint es usado para representar cualquier tipo de motor que se deseé utilizar, como un servomotor, un motor paso a paso, o uno de giro continuo. Si un Joint representa, por ejemplo, un servomotor de un brazo robótico, es posible asignarle una posición angular, o consultar cual es su posición actual para lograr mover el brazo. Si un Joint representa un motor de giro continuo, es posible variar su velocidad angular de giro, para por ejemplo, controlar un carro motorizado con ese tipo de motores. Cualquiera de dichas acciones debe realizarse usando las funciones disponibles en el controlador correspondiente, al cual previamente le fue asignado el control de dicho Joint con la función *addJoint(Joint *)*. De esta forma, a modo de ejemplo, si se desea asignar a un servomotor real una posición angular de π radianes ², se deben seguir los siguientes pasos:

- Informar al Joint la posición que necesita alcanzar, usando la función *setJointNextPosition((double)3.1415)*, en donde el argumento entre paréntesis corresponde a la posición angular a adoptar de tipo doble flotante. Si se desea asignar una posición angular a más de un Joint se debe realizar esta misma operación con cada uno de ellos.
- Solicitar al controlador, que en este caso corresponde a un controlador USB2Dynamixel, que actualice la posición de cada uno de los motores usando la función *Controlador:move()*.

De esta forma, es posible mover todos los motores que se deseen de forma simultánea, como lo es en el caso de la generación de caminatas en donde todas las

² por defecto el objeto Joint recibe y entrega valores en radianes, pero es posible elegir otro tipo de unidad disponible para evitar que el usuario deba realizar conversiones adicionales.

extremidades de un robot deben moverse al mismo tiempo.

El objeto CollisionObject es usado para representar objetos que eventualmente puedan colisionar durante un experimento. En un entorno real un CollisionObject puede representar, por ejemplo, un sensor de tacto que al hacer contacto con otros objetos pueda indicar una posible colisión. En el software V-REP, un CollisionObject corresponde a una interacción entre dos entidades dentro del escenario que posean propiedades de colisión activas, pudiendo detectar si existe colisión entre estas. De esta forma, si por ejemplo se quisiera detectar una colisión entre el piso del escenario de simulación y el torso de un robot, se deben seguir los siguientes pasos:

- Crear un objeto dentro del software de simulación que relacione el par de entidades de posible colisión.
- Crear un objeto CollisionObject correspondiente al objeto del punto anterior en el programa de entrenamiento.
- Solicitar al controlador, que en este caso corresponde al controlador Robot-VREP, que verifique si existe una colisión relacionada con el objeto del punto anterior usando la función *Controlador.readCollision(CollisionObject *)*, en donde el argumento entre paréntesis corresponde al puntero del objeto que relaciona el par de entidades a colisionar. Esta función retornará un valor booleano verdadero si existe una colisión entre el par de entidades, o falso si no existe colisión.

Con el uso de los objetos mencionados anteriormente es posible efectuar cualquier tipo de experimento relacionado con robótica móvil, por lo que RobotLib es una herramienta indispensable para el estudio de la robótica para cualquier rama de investigación. A continuación se muestra un programa de prueba, escrito en C/C++, en donde se acelera un motor de giro continuo unido a una hélice sobre la cual hay un cilindro (figura ??) que es expulsado por efecto de la aceleración centrífuga.

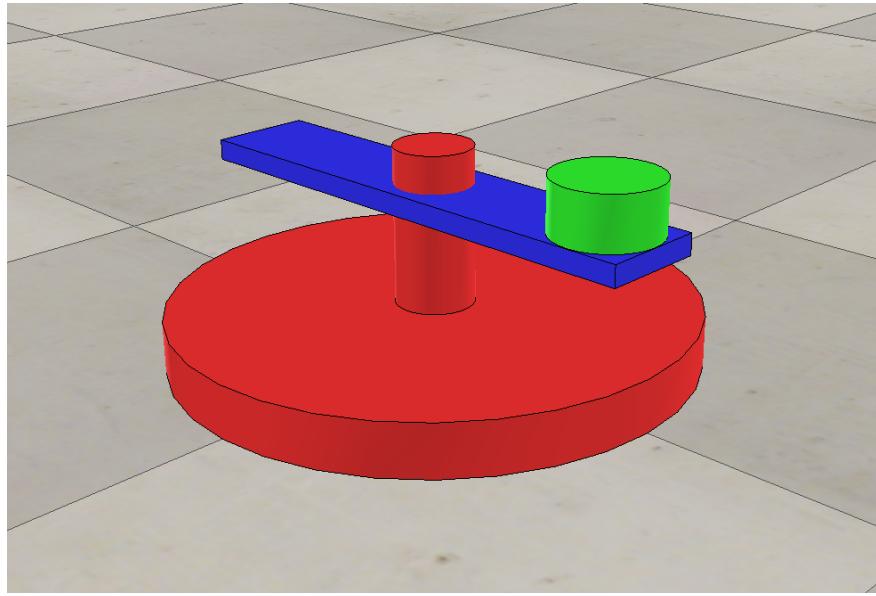


Fig. 5.1: Escenario de prueba de la librería RobotLib. El escenario de simulación de la figura muestra una hélice azul montada en una base que gira con el uso de un motor o *Joint*, y un cilindro verde sobre un extremo de la hélice. El experimento, realizado como ejemplo de uso de la librería RobotLib, tiene como objetivo hacer girar la hélice y acelerarla hasta expulsar el cilindro de sobre ella producto de la fuerza centrífuga generada sobre el cilindro producto del giro.

```

1 #include <unistd.h> // For use usleep
2 #include <ROBOTLIB> // For use the RobotLib library
3
4 using namespace std;
5
6 int main(int argc, char* argv[])
7 {
8     RobotVREP vrep(false); // Create a controller
9     Joint * joint = new Joint((char*)"DEG", (char*)"joint"); // Create
10    a motor
11    CollisionObject * cylinder = new CollisionObject((char*)"Collision");
12    // Create a collisionable object
13    vrep.addJoint(joint); // Add the motor to controller
14    vrep.addCollisionObject(cylinder); // Add the collisionable object
15    to controller
16
17    double vel = 0.0;
18    int notCollision = 0;
19
20    vrep.startSimulation(simx_opmode_oneshot_wait); // Start the
21    simulation
22
23    while (notCollision < 2)
24    {
25        notCollision = (vrep.readCollision(cylinder)) ? 0 : notCollision
26            + 1; // Read collision state

```

```

22     vel = vel + 0.2;
23     vrep.setJointTargetVelocity(joint, vel); // Set the joint
24         velocity
25     usleep(500000);
26
27     while (vel >= 0.2)
28     {
29         vel = vel - 0.2;
30         vrep.setJointTargetVelocity(joint, vel); // Set the joint
31             velocity
32         usleep(500000);
33
34     vrep.stopSimulation(simx_opmode_oneshot_wait); // Stop the
35         simulation
36     delete(joint);
37     delete(cylinder);
38
39     return(0);
40 }
```

Como se aprecia en la linea 4 del código anterior, una vez instalada la librería RobotLib, es posible utilizarla solo incluyéndola como allí se muestra. El programa crea un objeto RobotVREP para controlar los objetos dentro del software V-REP, un objeto Joint para el motor que hace girar la hélice, y un objeto CollisionObject que relaciona el contacto entre la hélice y el cilindro, y añade estos dos últimos al controlador. Luego, se inicia una simulación, y se aumenta la velocidad de giro del Joint hasta que se detectan dos “no colisiones”³ consecutivas relacionadas con el CollisionObject, producto de la caída del cilindro desde la superficie de la hélice. Una vez confirmada la caída del cilindro de la superficie de la hélice se reduce la velocidad del Joint hasta detenerlo para finalmente detener la simulación. En el capítulo siguiente se mostrará como realizar el diseño de las plataformas robóticas a usar para la generación de caminatas dentro del software de simulación V-REP.

³ Por posibles inexactitudes de la escena y los objetos dentro de ella se producen breves detecciones de no colisión entre objetos que si están colisionando, por lo que se verifican dos no colisiones consecutivas para asegurar que los objetos no estén colisionando realmente.

6. DISEÑO DE PLATAFORMAS ROBÓTICAS EN EL ENTORNO VIRTUAL

Tal como se mostró en el ejemplo de uso de RobotLib, para realizar un experimento en un entorno virtual de simulación es necesario recrear un sistema (en este caso una plataforma robótica) dentro del escenario de simulación. Esto puede realizarse creando piezas simples, como planos, discos, cubos, cilindros o esferas, como se hizo para el sistema usado en el ejemplo de uso de RobotLib; pero cuando se trata de sistemas mas elaborados, como lo son las plataformas robóticas a utilizar en los experimentos de generación de caminatas, se hace necesario realizar procedimientos mas avanzados. El procedimiento necesario para recrear las plataformas robóticas en el escenario de simulación del software V-REP se detalla a continuación.

1. **Importar el diseño 3D de una plataforma robótica:** Con el objetivo de emular de mejor manera las características de cada plataforma robótica es que se debe importar el diseño realizado de ellas (en algún programa de diseño 3D) en el simulador. La importación del diseño debe hacerse a través de la barra de menú superior del simulador (*File→Import→Mesh...*) como se muestra en la figura 6.1a, aceptándose formatos con extensión *obj* ,*dxf* o *stl*. Luego debe indicarse el archivo que se desea importar (6.1b), para ser cargado y visualizado por el programa (6.1c).
2. **Dividir el modelo importado en piezas individuales.** Luego de la importación del modelo es posible observar en el simulador una sola pieza correspondiente a la estructura completa del robot, y para hacerla dinámica es necesario dividir el modelo en piezas individuales. Esto es posible de realizar a través del menú des-

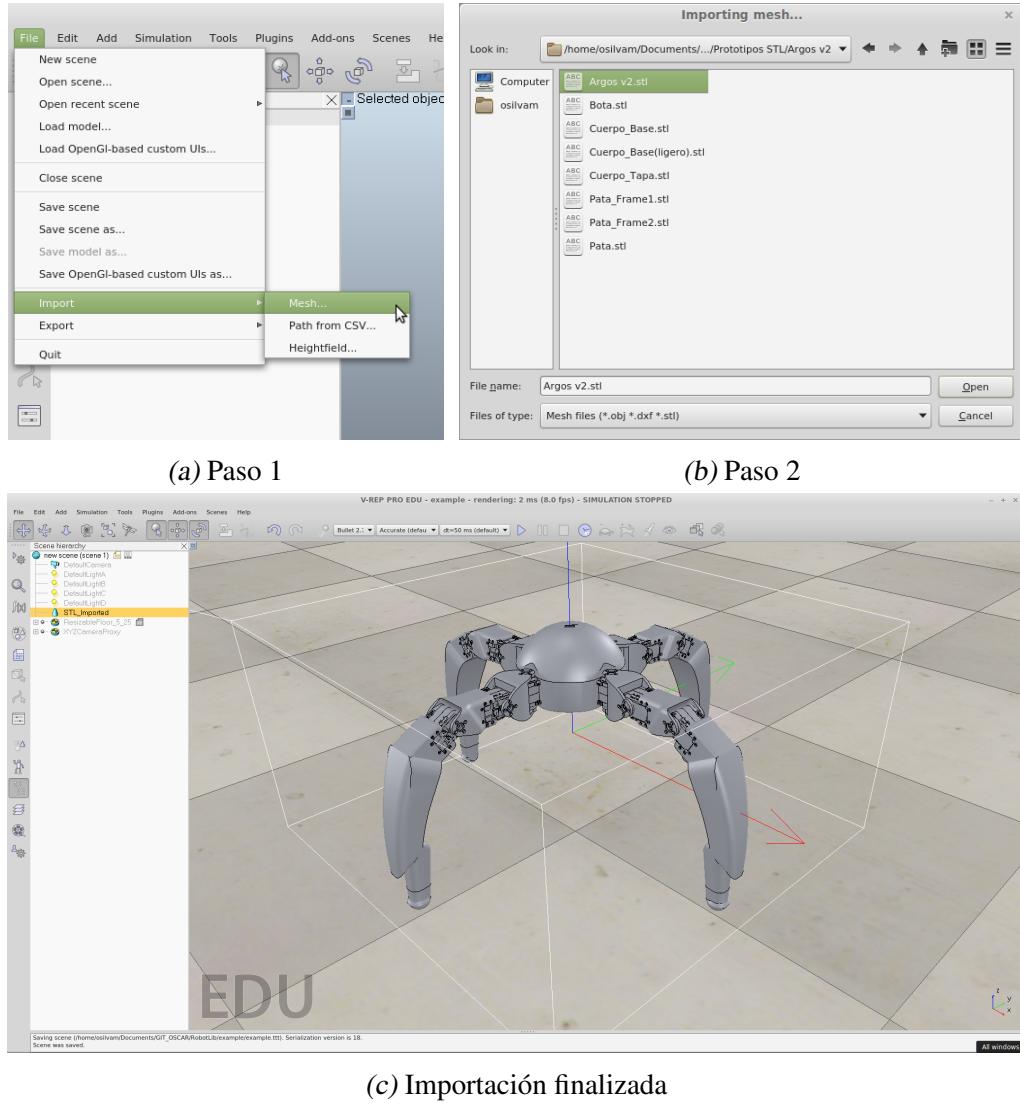


Fig. 6.1: Importación de modelo 3D en V-REP. La importación de un modelo 3D hacia V-REP debe realizarse siguiendo dos simples pasos: acceder a la ruta mostrada en (a), seleccionar el modelo 3D a importar (b) en alguno de los formatos permitidos, y abrirlo para luego ser visualizado en el simulador como se muestra en (c).

plegado por medio del click derecho sobre la pieza a dividir (*Edit→Grouping / Merging→Divide selected shape*) como se muestra en la figura 6.2, obteniendo así todas las piezas que componen el modelo original por separado.

3. Selección de piezas a utilizar. Ya que el modelo del robot es simétrico, y todas sus extremidades son iguales y están compuestas de las mismas piezas, basta



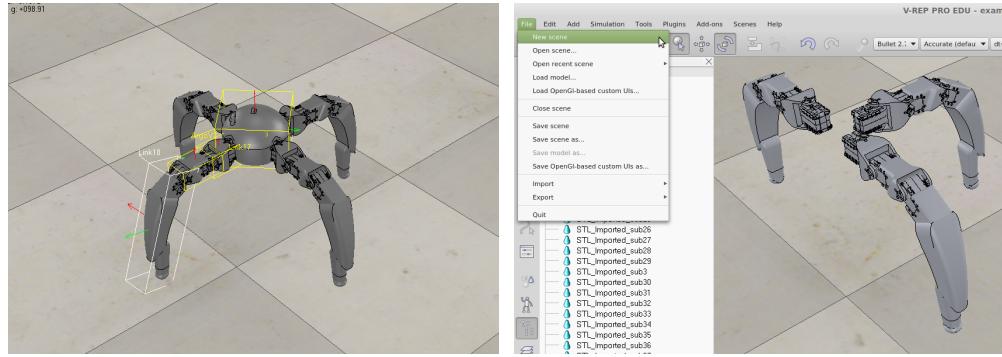
(a) Robot en una pieza

(b) Robot dividido en muchas piezas

Fig. 6.2: División del modelo. Haciendo click derecho sobre una entidad correspondiente a una importación de un modelo tridimensional es posible dividirla en partes individuales, para posteriormente articular dichas partes para generar un modelo dinámico que pueda interactuar con el escenario de simulación.

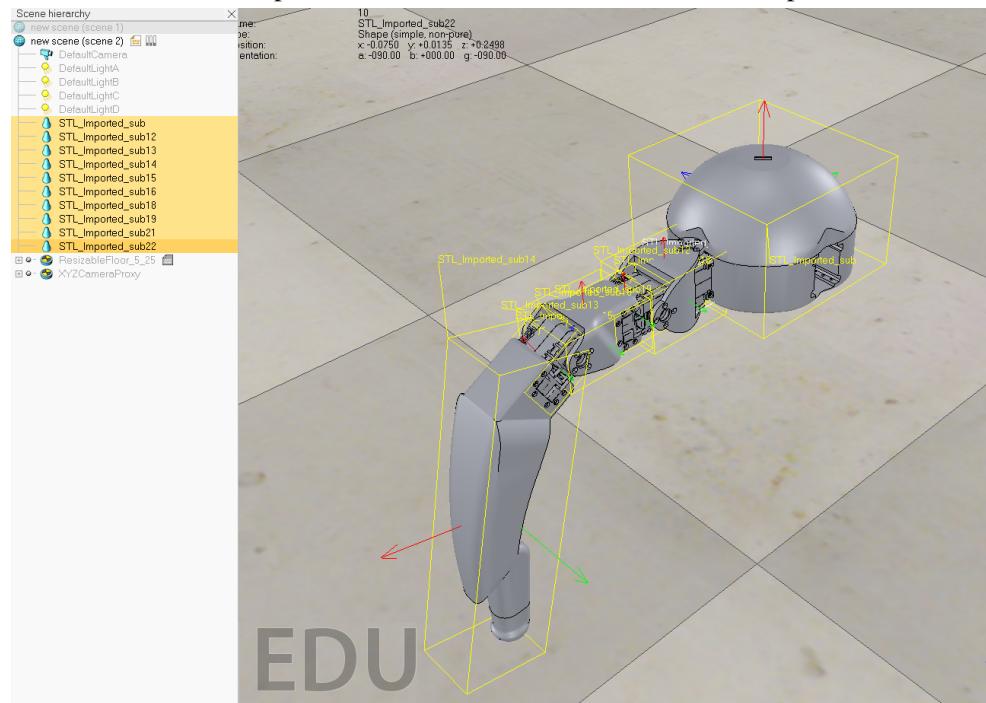
con trabajar sobre una de las extremidades del robot para generar las restantes. Por esta razón es que solo serán útiles las piezas que componen una de las extremidades y el torso central del robot. Para trabajar únicamente con las piezas útiles deben seleccionarse cada una de estas (figura 6.3a) y cortarse (CTRL+X puede utilizarse para esta acción). Luego debe abrirse una nueva escena través de la barra de menú superior del simulador (*File→New scene*) como se muestra en la figura 6.3b, y pegar en ella (CTRL+V puede utilizarse para esta acción) las piezas seleccionadas y cortadas anteriormente (figura 6.3c). Luego de esto será posible manipular de forma fácil y rápida las piezas de utilidad para armar el modelo dinámico del robot.

4. **Generar piezas convexas a partir de piezas importadas.** Utilizar piezas importadas de manera directa (sin ningún tipo de simplificación) involucra una cantidad extremadamente grande de cálculos para el procesamiento y ejecución de una simulación debido a que están formadas por un gran número de triángulos, siendo necesario transformarlas en estructuras más simples. Un método que permite realizar una simplificación útil es el que permite generar una pieza convexa y sin agujeros a partir de otra pieza más compleja, ya sea importada o creada dentro del simulador. Estas estructuras convexas estarán confor-



(a) Selección de piezas útiles

(b) Extracción de piezas útiles

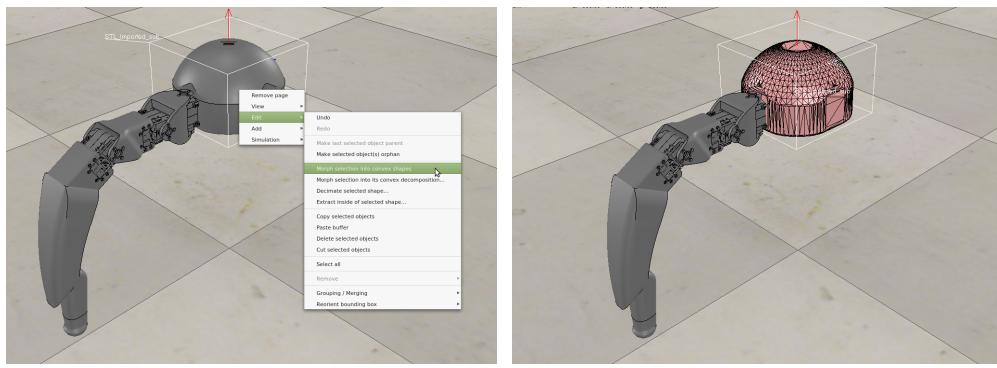


(c) Inserción de piezas útiles en una nueva escena

Fig. 6.3: Extracción de piezas útiles. Para extraer las piezas útiles pertenecientes a un modelo estas deben (a) seleccionarse, (b) cortarse y finalmente (c) pegarse en una nueva escena para facilitar su manipulación.

madas por un número menor de triángulos que la pieza original, facilitando las tareas de computo del procesador. La tarea de generar una pieza convexa a partir de otra pieza es posible de realizar a través del menú desplegado por medio del click derecho sobre la pieza a trabajar (*Edit→Morph selection into convex shapes*) como se muestra en la figura 6.4a, resultando en la pieza rosada formada

por triángulos mostrada en la figura 6.4b. Si una pieza posee demasiadas curvas cóncavas hacia su exterior provocará que al realizar el procedimiento descrito las paredes de la pieza se deformen, ya que el método para la obtención de una estructura convexa a partir de otra busca encontrar el sólido de menor volumen y de paredes de trazos rectos que encierre a la pieza en su totalidad. Es por esto que es recomendada realizar este paso cuando la pieza a simplificar no requiere demasiado detalle.



(a) Convirtiendo una pieza en convexa

(b) Pieza convexa obtenida

Fig. 6.4: Generación de una pieza en convexa. A través del procedimiento mostrado en (a) es posible obtener una simplificación de una pieza importada, resultando una figura convexa conformada por una cantidad de triángulos menor a la pieza original (b).

5. Simplificación de una pieza. Otro método de simplificación, que puede usarse además en conjunto con el paso 4, es el de reducción del número de triángulos que componen la estructura de una pieza, reduciendo los detalles de esta. Esto puede realizarse a través del menú desplegado por medio del click derecho sobre la pieza a simplificar (*Edit→Decimate selected shape...*) como se muestra en la figura 6.5a, desplegándose una ventana que solicita el porcentaje de reducción de triángulos deseada (20 % por defecto) como se muestra en la figura 6.5b. Luego de confirmado el porcentaje de reducción de triángulos se genera una nueva pieza con una estructura formada por el porcentaje de triángulos especificado (figura 6.5c). Siguiendo los pasos 4 y/o 5 con todas las piezas se obtiene el resultado mostrado en la figura 6.5d.

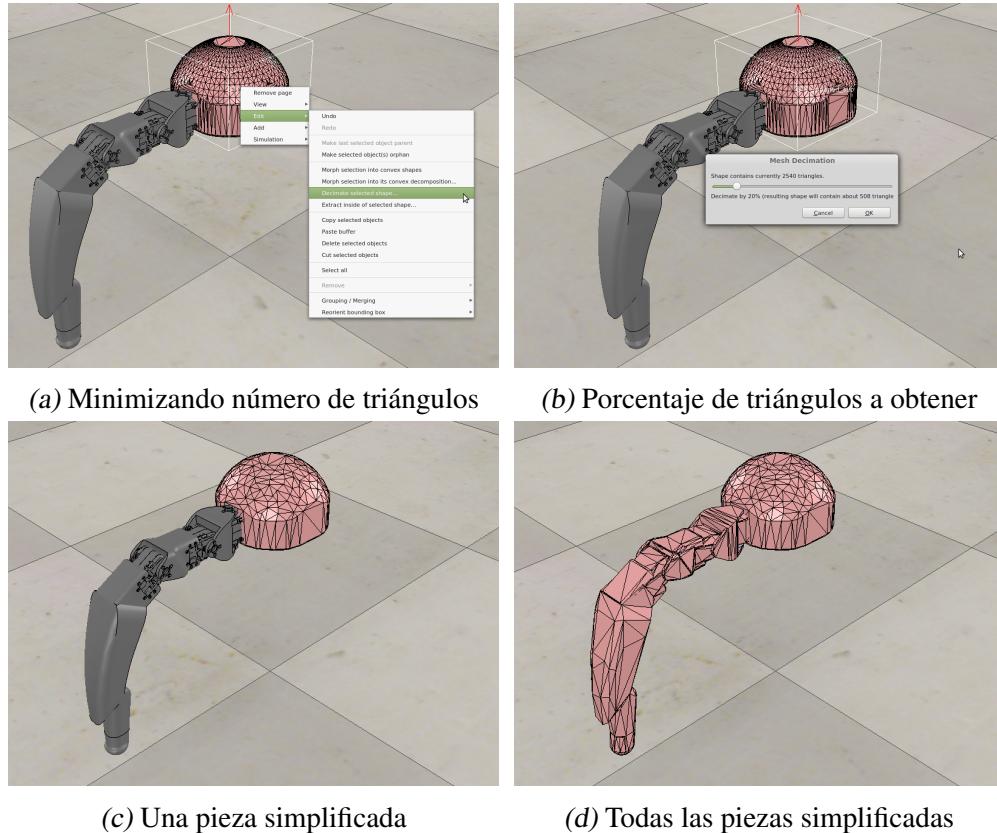
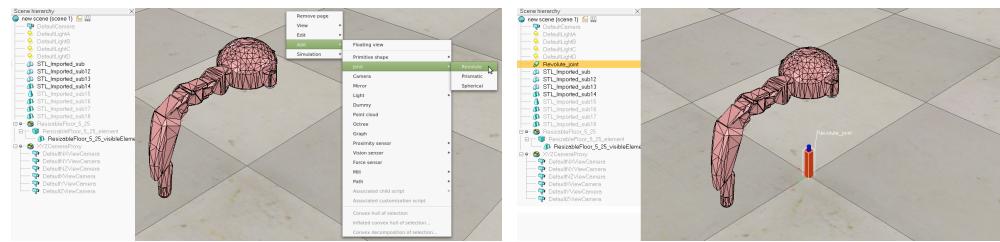


Fig. 6.5: Minimización del numero de triángulos de una pieza. Accediendo al menú mostrado en (a) es posible obtener una simplificación de cada una de las piezas de un modelo, pudiendo indicar el porcentaje de reducción del número de triángulos que componen. Una vez seleccionada la opción de reducción de triángulos, el programa consulta por el porcentaje de simplificación requerido (b) y genera una nueva pieza con un numero menor de triángulos (c). Siguiendo el mismo procedimiento con todas las piezas resulta en la figura mostrada en (d).

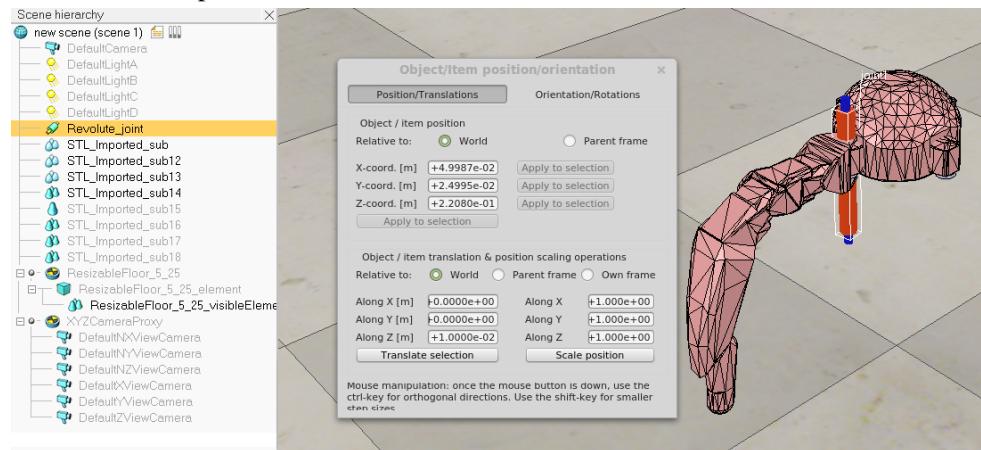
6. Crear articulaciones. Una vez que las piezas para una de las extremidades del robot y su tronco están listas es posible crear las articulaciones que permitirán dar movilidad al robot. Para crear una articulación es necesario crear un objeto de tipo *Joint de Revolución* dentro del simulador, a través del menú desplegado por medio del click derecho sobre cualquier parte del escenario (*Add→Joint→Revolute*) como se muestra en la figura 6.6a. Una vez realizado este procedimiento aparecerá un objeto Joint con la forma de un bastón azul alargado dentro de otro bastón anaranjado más corto justo en el centro del escenario de simulación (Fi-

gura 6.6b). Finalmente solo resta posicionar el objeto Joint entre las piezas del robot que se desean articular, haciendo uso del menú de posicionamiento y orientación de objetos ubicado en la barra superior. Así una vez seleccionado un objeto este puede ser posicionado y orientado entre las piezas del robot que se desean articular, como se muestra en la figura 6.6c. Es importante que la posición y orientación del Joint en la escena con respecto a las piezas a articular sea correcta, ya que de esto dependerá que el dinamismo presentado en la simulación sea lo más cercano posible al del modelo real.



(a) Menú para crear un Joint

(b) Joint creado



(c) Ubicación de un Joint en el modelo del robot

Fig. 6.6: Añadir un Joint para crear una pieza articulada. A través del menú mostrado en la figura (a) es posible añadir un objeto de tipo Joint a la escena con el fin de articular un par de piezas del modelo a diseñar. Una vez añadido el objeto este aparecerá en el origen del escenario (coordenada cartesiana (0, 0, 0)) como es posible apreciar en (b) y debe ser situada y orientada en el punto adecuado entre la piezas que se desean articular.

Luego de crear la primera articulación correspondiente al primer grado de libertad de una de las extremidades del robot es posible seguir el mismo procedi-

miento anterior para crear las articulaciones restantes. De esta forma se incorporan al modelo las dos articulaciones faltantes como se muestran en la figura 6.7a, en la cual además puede apreciarse una nueva asignación de nombres. Los Joints fueron nombrados como *joint1*, *joint2* y *joint3*, numerados en orden descendente desde el más cercano al tronco hasta el más lejano. De la misma manera se nombraron las piezas que componen la extremidad, agregándose además el sufijo *.dyn* a las piezas creadas a partir de las piezas importadas (siguiendo los pasos 4 y/o 5). Finalmente se nombra el tronco central del robot y su homólogo simplificado como *ArgoV2* y *ArgoV2.dyn* respectivamente.

Una vez realizados todos estos pasos es posible crear una estructura jerarquizada de cada una de las piezas del modelo con el fin de indicarle al simulador las piezas que se desean articular, como se muestra en la figura 6.7b. De esta forma se ubica al tronco central simplificado del robot como la pieza base de la jerarquía del modelo, y a su homólogo no simplificado y el *joint1* como hijos de este. Luego como hijo del *joint1* se ubicará el *Link1.dyn*, correspondiente a la primera pieza articulada de la extremidad, y como hijos de este a su homólogo no simplificado y el joint correspondiente a la articulación siguiente. Este procedimiento continua de la misma forma hasta jerarquizar todas las piezas del modelo.

7. **Replicación de extremidades en el modelo.** Una vez obtenido el diseño del tronco y una de las extremidades del robot ya articulado es posible añadir las demás extremidades a través de un proceso de replicación. Para obtener otra de las extremidades del robot basta con seleccionar todos los componentes de la extremidad existente desde el joint que une el tronco central del robot hacia la punta, copiarlos y luego pegarlos como se muestra en la figura 6.8a. Visualmente no será posible apreciar la existencia de una nueva extremidad en la escena ya que la existente con la nueva que se ha copiado y pegado usarán el mismo espacio físico, pero se podrá observar la existencia del segundo grupo de piezas

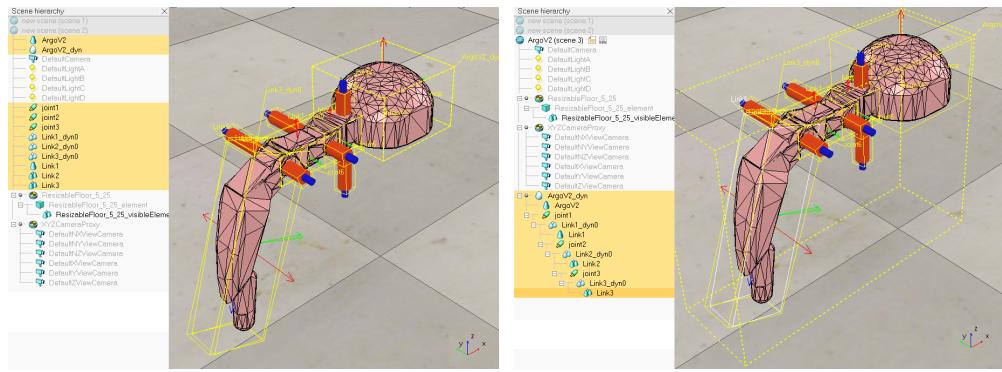


Fig. 6.7: Articulación completa una extremidad del modelo del robot. La figura muestra como es posible, luego de haber añadido los Joints necesarios y nombrar correctamente a cada una de las partes (a), conformar y articular un modelo de robot con extremidad robótica a través de una estructura jerárquica (b).

jerarquizadas que conforman la nueva extremidad en el display a la izquierda de la imagen de la figura 6.8a. Finalmente solo resta posicionar y orientar la nueva extremidad en su ubicación correspondiente, haciendo uso del menú de posicionamiento y orientación de objetos ubicado en la barra superior. Esto debe realizarse seleccionando cada unas de las piezas de la nueva extremidad y rotando en 90 grados las piezas sobre el eje z del escenario al rededor del objeto padre, en este caso, del tronco central del robot, como se muestra en la figura 6.8b. El mismo procedimiento anterior debe ser realizado, rotando las nuevas extremidades copiadas la cantidad necesaria para separar las cuatro extremidades en 90 grados cada una, como se aprecia en la figura 6.8c.

8. **Habilitar los objetos Joints como motores.** Para que los objetos Joints funcionen como motores y sea posible asignarles posiciones es necesario habilitarlos como tales. Así, una vez seleccionados todos los objetos Joint del modelo es posible acceder al menú de propiedades de objetos ubicado en la barra de tareas a la izquierda del simulador. Una vez desplegado el menú (figura 6.9a) es necesario asegurar que los Joints se encuentren en el modo de trabajo *Torque/Force mode* y la casilla *Position is cyclic* se encuentre inactiva. Además es posible acceder al

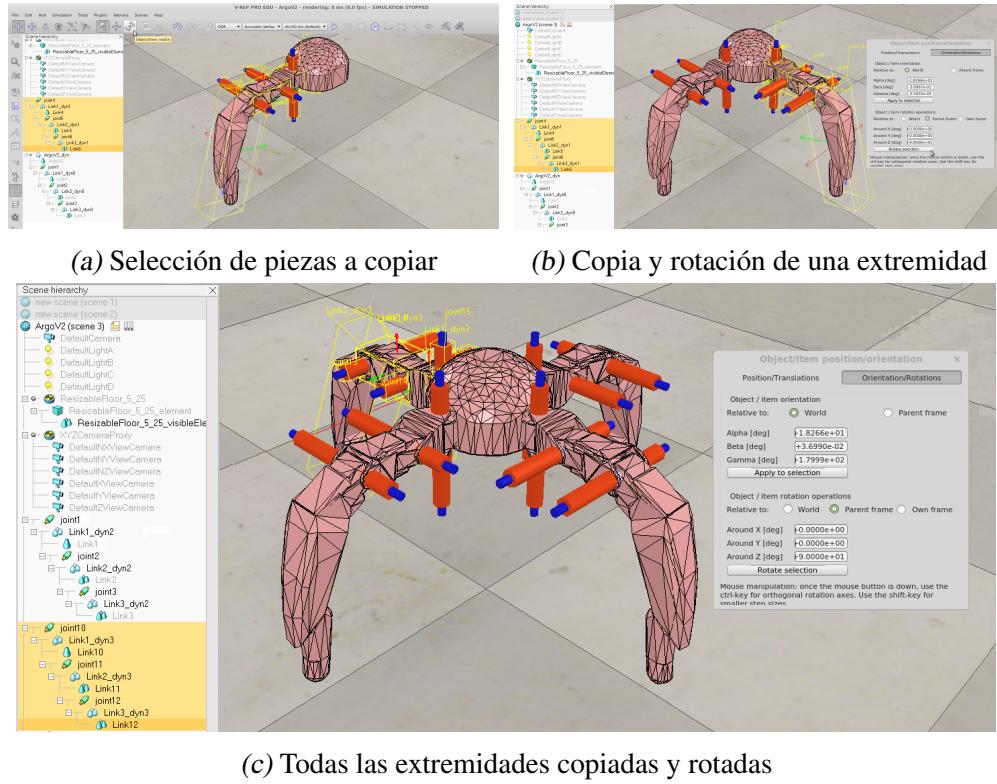
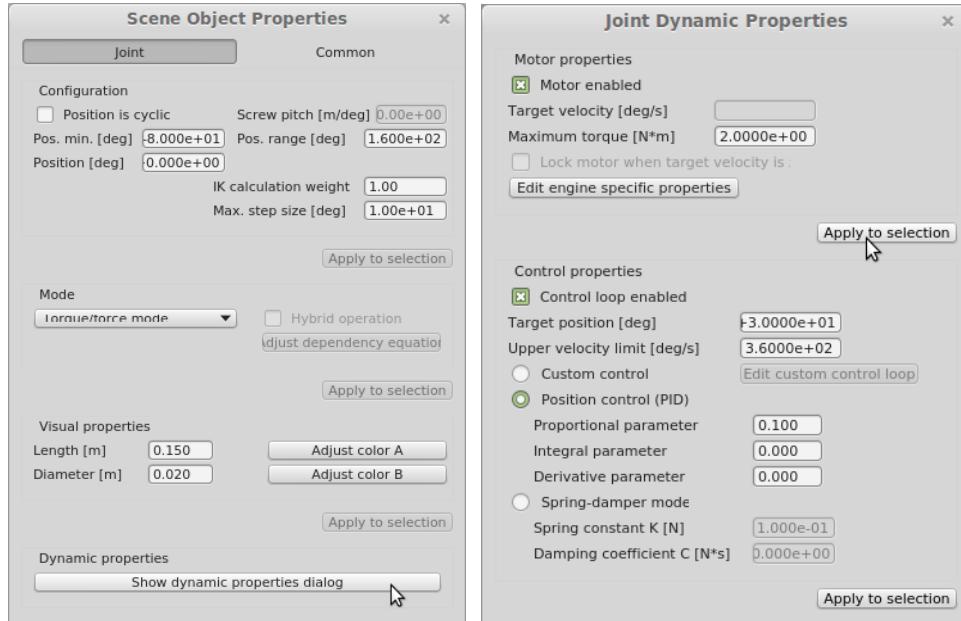


Fig. 6.8: Modelo armado con sus cuatro extremidades. Luego de copiar y pegar completamente una extremidad (a) es posible rotarla al rededor del tronco central del robot para situarla en su posición definitiva (b). Siguiendo el mismo procedimiento para las dos extremidades faltantes es posible obtener el modelo completo de un robot con un tronco central y 4 extremidades como el mostrado en (c).

menú de propiedades dinámicas de los objetos Joint por medio del botón *Show dynamic properties dialog* como se indica en la figura 6.9a. Ya desplegado el menú de propiedades dinámicas se debe asegurar que la casilla *Motor enable* se encuentre activa; asignar el *Maximum torque*, que en este caso es de 2 [Nm]; y que la casilla *Control loop enable* se encuentre activa. Para el control de posición del motor puede usarse el controlador deseado por el usuario, que en este caso corresponde a un controlador PID sólo con parámetro proporcional. Para que todas las configuraciones asignadas afecten a cada uno de los objetos Joint seleccionados previamente es necesario activar el botón *Apply to selection* en cada sección de configuración, como se indica en la figura 6.9b.



(a) Menú de propiedades de un Joint

(b) Propiedades dinámicas de un Joint

Fig. 6.9: Habilitando Joints como motores activos. La figura muestra el procedimiento para habilitar a los objetos Joints para funcionar como motores a través del menú de propiedades de los objetos (a). Además, por medio del menú anterior es posible acceder al menú de propiedades dinámicas de los objetos Joint mostrado en (b), pudiendo allí asignar el torque de los motores y configurar el tipo de control de posición de estos mismos.

9. Cambiar la visibilidad de los objetos en el escenario y sus características.

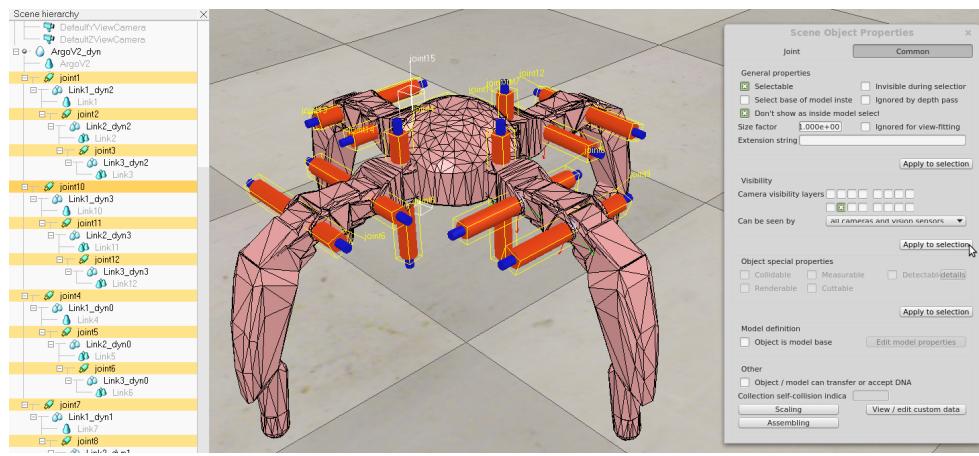
Previamente se indicó como realizar una simplificación de las piezas importadas, debido a que realizar cálculos de las físicas de los objetos que interactúan en el escenario de simulación directamente en las piezas importadas resultaba demasiado costoso. Estas piezas simplificadas son las que interactuarán dentro del escenario para cualquier efecto de cálculo, y las piezas importadas solo serán la parte visible del modelo para el usuario. Es por esto que se deben ocultar las piezas simplificadas y los demás objetos que no se desean observar durante las simulaciones, y se deben hacer visibles los demás objetos ornamentales del modelo que no influyen en los cálculos. Esto es posible de realizar a través del menú de propiedades de los objetos del simulador en la pestaña *Common*. En dicho menú aparecen dos secciones importantes, la de visibilidad (*Visibility*) y

la de propiedades especiales (*Object special properties*), como se muestra en las imágenes de la figura 6.10. En la sección de visibilidad aparecen dos filas de ocho casillas, en donde la primera fila corresponde a capas visibles en el escenario, y la segunda corresponde a capas ocultas. En la sección de propiedades especiales aparecen cinco casillas, la casilla *Collidable* referente a si un objeto esta habilitado para colisionar con otros objetos, la casilla *Mesurable* referente a si un objeto esta habilitado para soportar cálculos de mínimas distancias con otro objetos, la casilla *Detectable* que permite habilitar o deshabilitar la capacidad del objeto de ser detectado por sensores de proximidad, la casilla *Renderable* que permite habilitar al objeto para ser detectado por sensores de visión, y la casilla *Cuttable* que permite a un objeto ser modificado por herramientas de corte dentro del simulador.

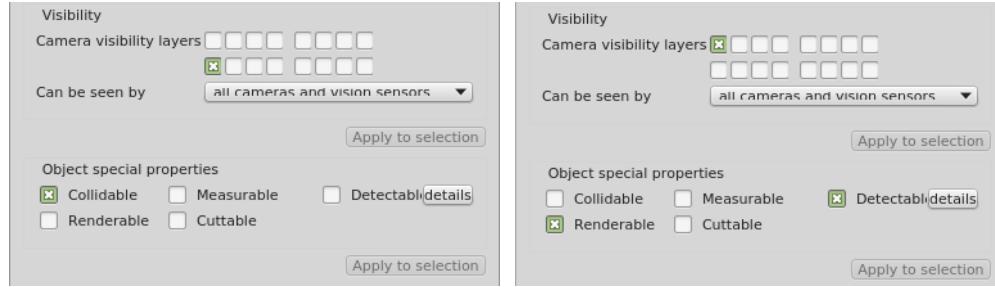
Entendido lo anterior se partirá configurando los objetos Joint, los cuales deben ser seleccionados y a través del menú configurados de tal forma que queden ocultos a la vista. Esta configuración implica deshabilitar la casilla activa de la primera fila de visibilidad y habilitar su casilla par ubicada inmediatamente debajo de la anterior, como se aprecia en la figura 6.10a. Finalmente se debe presionar el botón *Apply to selection* para asignarles a todos los Joints seleccionados la misma configuración. El siguiente paso es configurar las piezas dinámicas y ornamentales del modelo. Las piezas dinámicas serán configuradas con una visibilidad oculta (al igual que los objetos Joint) y propiedades de objeto para detección de colisiones (Figura 6.10b). Las piezas ornamentales correspondientes a las partes importadas se configuran con una visibilidad activa en la escena y con propiedades de objeto para ser detectados por sensores de proximidad y de visión (Figura 6.10c). El modelo resultante es mostrado en la figura 6.10d en el que se pueden observar los objetos Joints ocultos y el diseño importado con todos los acabados y detalles a la vista.

10. Cambiar el color de las piezas visibles.

Una vez obtenido el modelo mostrado

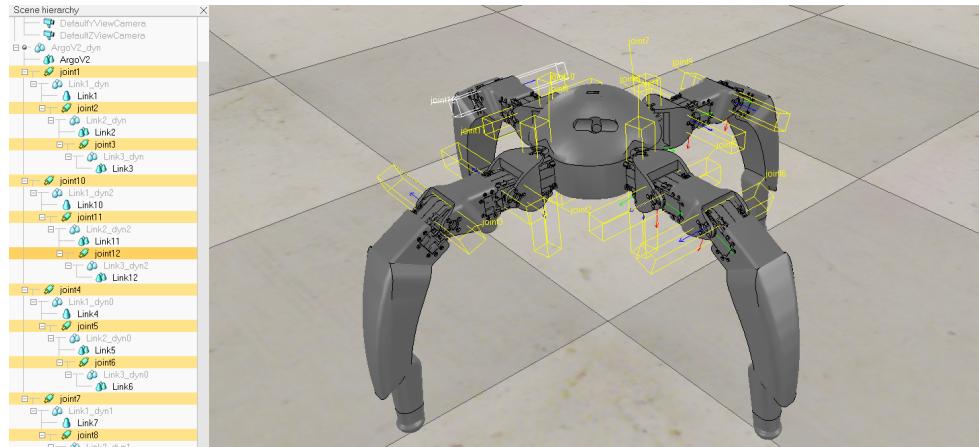


(a) Cambiando todos los objetos Joints a la capa oculta



(b) Configuración de objetos dinámicos

(c) Configuración de objetos ornamentales



(d) Resultado visto en el simulador

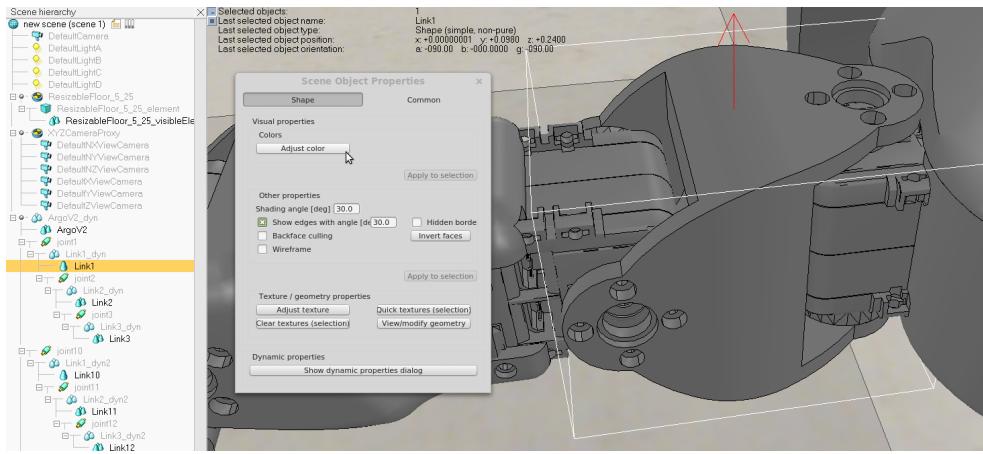
Fig. 6.10: Cambiar la visibilidad y otras características de los objetos. A través del menú de propiedades de objeto es posible configurar que los objetos Joint no sean visibles en la escena (a), y que además, los objetos dinámicos estén ocultos y tengan propiedades de Collidable (b), y los objetos ornamentales sean visibles en la escena y posean propiedades de Detectable y Renderable (c), terminando con el modelo mostrado en (d)

en la figura 6.10d solo con las piezas ornamentales visibles, es posible cambiar el color de las piezas, únicamente con un objetivo estético, para asemejar el modelo virtual del simulador lo más posible al real. Para acceder a cambiar el color de una pieza debe seleccionarse la pieza y abrir el menú de propiedades de objeto. Si la pieza es simple, osea esta compuesta por solo un elemento, el menú de propiedades será como el mostrado en la figura 6.11a y podrá accederse al menú de cambio de color haciendo click en el botón *Adjust color* indicado. Si la pieza es compuesta, ya sea por varias piezas simples o compuestas, el menú de propiedades será como el mostrado en la figura 6.11b y podrá accederse al menú de cambio de color haciendo click en el botón *Edit appearance of compound shape* indicado.

En la figura 6.12 se aprecia el menú desplegado producto del procedimiento anterior sobre una pieza compuesta de 3 componentes simples. Luego de seleccionar alguna de las piezas, haciendo click en el botón *Adjust color* (Figura 6.12a) se desplegará el menú de cambio de color (en el caso de una pieza simple esta acción no es necesaria y el menú de cambio de color es desplegado inmediatamente) mostrado en la figura 6.12b. Haciendo click en el botón *Ambient/diffuse component* aparecerá una ventana de ajuste de color (Figura 6.12c), en donde este puede ser ingresado tanto en formato RGB como HSL. Luego de modificar todas las piezas del conjunto solo basta cerrar el menú de edición de componentes para volver a la ventana principal del simulador para seguir cambiando el color de las demás piezas, para posteriormente obtener el modelo como el que se muestra en la figura 6.12c.

11. Asignación de posiciones iniciales y rangos de movimiento a los objetos

Joint. El paso final para comenzar a trabajar con el modelo del robot para la generación de caminatas es asignar la posición inicial y el rango de movimiento de cada uno de los objetos Joint, como se muestra en la figura 6.13. Una vez seleccionado un objeto Joint, a través del menú de propiedades de objeto es po-



(a) Menú de propiedades de una pieza simple



(b) Menú de propiedades de una pieza compuesta

Fig. 6.11: Acceso al menú de cambio de color de una pieza. La figura muestra las diferentes formas de acceder al menú de cambio de color de una pieza que puede ser simple (a) o estar compuesta por varias piezas simples y/o otras piezas compuestas (b).

sible modificar la posición inicial *Position* y los límites de movimiento del Joint *Pos. min.* y *Pos. range*. Observando las figuras 6.13a y 6.13b se puede apreciar el cambio de ángulo inicial del Joint indicado de 0 a 30 grados.

Luego de realizar todos los pasos anteriormente mencionados se obtendrá como resultado un modelo terminado y listo para trabajar, como el mostrado en la figura 6.14. Finalmente, terminado el modelamiento de las plataformas robóticas en el simulador es posible iniciar los entrenamientos para generar las caminatas en los robots, lo cual será relatado en el siguiente capítulo.

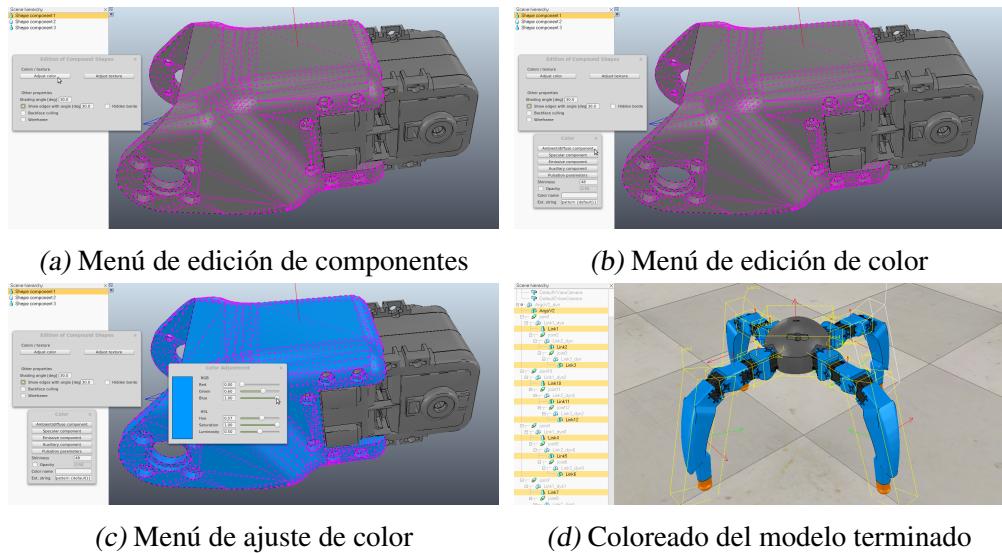
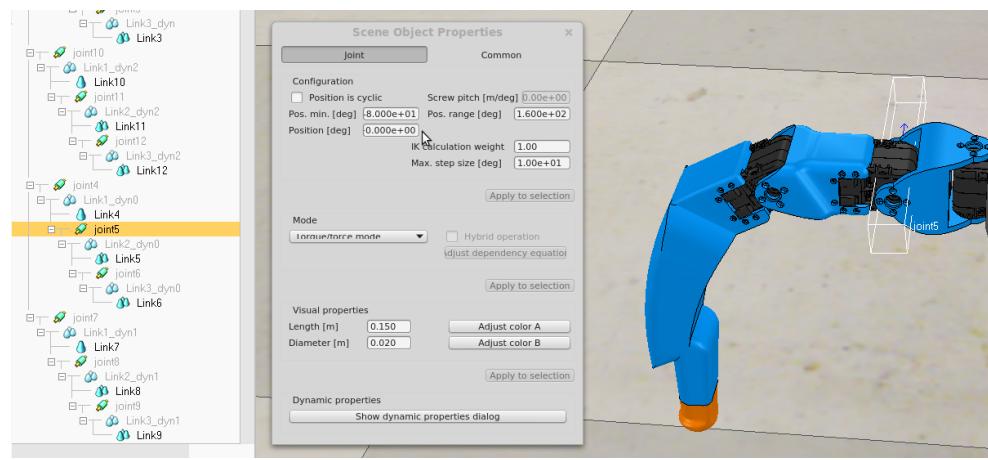
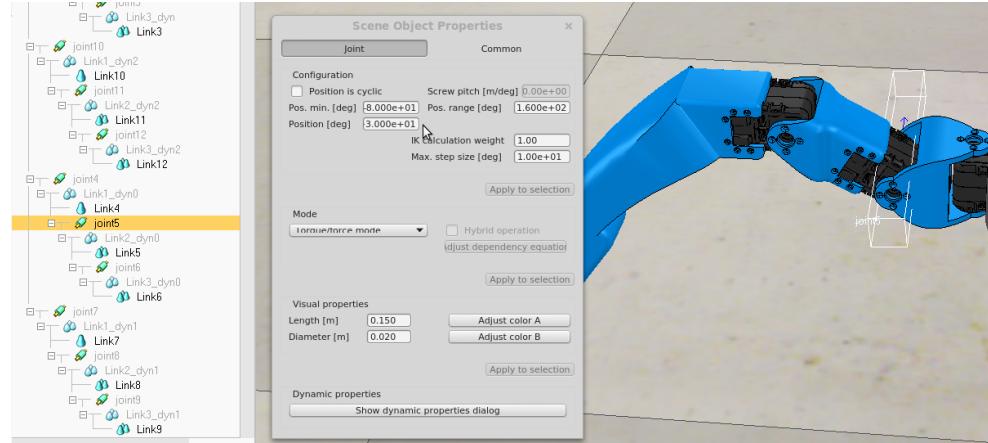


Fig. 6.12: Cambio de color de una pieza compuesta. La figura muestra el procedimiento efectuado sobre una pieza, compuesta de 3 piezas simples, para cambiar el color de sus componentes. Accediendo de la forma indicada en (a), (b) y (c) es posible cambiar el color de cada uno de los componentes de la pieza mostrada. Completando el procedimiento con las demás piezas del modelo es posible obtener el resultado mostrado en (d).



(a) Joint con posición inicial 0 grados



(b) Joint con posición inicial 30 grados

Fig. 6.13: Posición inicial y rango de movimiento de un Joint. La figura describe el procedimiento para ajustar la posición inicial de un Joint y sus límites de movimiento desde una posición mínima hasta una posición dada por el rango asignado. En (a) puede observarse que el Joint tiene asignado un valor de posición inicial por defecto cero, el cual es modificado en (b) para cambiar la pose inicial del robot.



Fig. 6.14: Modelo del robot ArgoV2 terminado. En la figura se muestra el resultado final del procedimiento llevado acabo para obtener el sistema robótico a utilizar para la tarea de generación de caminatas en robots con extremidades móviles dentro del software de simulación V-REP.

7. SIMULACIÓN DE CAMINATAS

Con el fin de lograr que las plataformas robóticas aprendan a mover sus extremidades de tal manera de generar una caminata que le permita desplazarse es necesario que estas entrenen. Un entrenamiento consiste en un gran número de simulaciones en donde un robot intentará realizar movimientos con el objetivo de desplazarse. Cada simulación tendrá una duración máxima de 6 segundos durante los cuales el robot podrá moverse, midiéndose su desempeño una vez finalizada la simulación. Durante el transcurso de los 6 segundos de simulación el programa de entrenamiento comprobará iteración tras iteración la existencia de colisiones entre las piezas de la estructura del robot y el suelo del escenario de simulación (exceptuando las puntas de las patas que deben hacer contacto con el suelo para lograr el desplazamiento del robot). Si una colisión es detectada, la simulación es terminada de forma abrupta, y los resultados de dicha simulación no son contemplados para el entrenamiento.

Para el caso de HyperNEAT y τ -HyperNEAT cada simulación de 6 segundo busca verificar el desempeño en la generación de caminatas de ambos métodos. Para ambos casos se ha establecido una configuración de substrato del tipo state-space sandwich, con una capa de entrada, una capa de salida y solo una capa oculta o intermedia. A la capa de entrada del substrato ingresarán los valores de las últimas posiciones de los motores de cada robot, junto a dos señales senoidales desfasadas en 90 grados; así si el robot posee 12 motores para mover sus extremidades, la capa de entrada poseerá 14 nodos. A la capa intermedia, conformada por n^2 nodos con n como el número de patas del robot, ingresarán las señales provenientes de la capa de entrada. Finalmente la capa de salida, conformada por tantos nodos como motores tenga el robot a utilizar, recibirá las señales provenientes de los nodos de la capa intermedia, y

sus salidas corresponderán a las nuevas posiciones que deben adoptar los motores del robot. Para el cálculo de una nueva iteración (nuevas posiciones para los motores), las posiciones de los motores obtenidas anteriormente a la salida del substrato serán las nuevas entradas de este. De esta forma el substrato en ambos métodos tendrá la labor de generar las señales de accionamiento de cada motor para lograr el desplazamiento del robot, a partir de las dos señales senoidales adicionales de entrada.

Para el caso de HyperNEAT, la red CPPN que establece las conexiones en el substrato posee sus cuatro entradas básicas de coordenadas x e y de los nodos de origen y destino, además de dos entradas adicionales de información referente al espaciamiento entre ellos a lo largo de los ejes cartesianos, y como salida los pesos de las conexiones entre nodos de la capa de entrada e intermedia y entre la capa intermedia y de salida. Para el caso de τ -HyperNEAT, la red CPPN que establece las conexiones en el substrato posee las mismas entradas que para el caso de HyperNEAT, pero como salida, además de entregar los pesos de las conexiones entre nodos de la capa de entrada e intermedia y entre la capa intermedia y de salida entrega los retardos correspondientes a dichas conexiones. En la figura 7.1 se puede apreciar un ejemplo de configuración de ambos métodos para el caso de un robot cuadrúpedo de tres grados de libertad por extremidad.

Una vez finalizada una simulación exitosa (no se detectaron colisiones), es necesario medir y evaluar el desempeño de la caminata generada por el robot. La determinación de que tan bueno o que tan malo fue el desempeño del robot durante una simulación es un aspecto crítico del entrenamiento, ya que esto determinará de qué manera evolucionarán las CPPNs encargadas de generar las conexiones en el substrato de HyperNEAT y τ -HyperNEAT. En la siguiente sección se ahondará respecto a la medición y evaluación del desempeño de las caminatas generadas por ambos métodos.

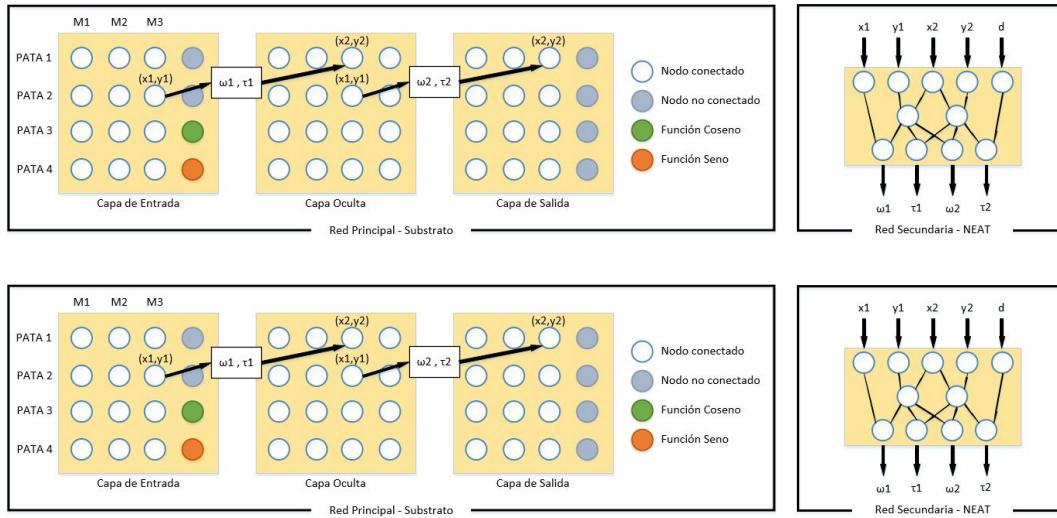


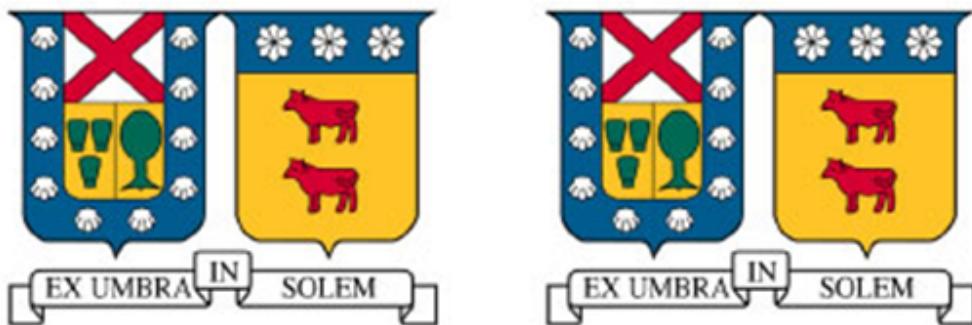
Fig. 7.1: Esquema estructural de HyperNEAT y τ -HyperNEAT. En la figura, los recuadros superiores e inferiores muestran el esquema estructural del método HyperNEAT y τ -HyperNEAT respectivamente, usado en los experimentos con un robot cuadrúpedo de 3 grados de libertad por extremidad. Ambos recuadros derechos corresponden a la estructura del substrato del método respectivo, mientras que los recuadros izquierdos corresponden a la estructura básica de la red CPPN, en las que es posible apreciar la diferencia en la cantidad de salidas entre ambas debido a que en τ -HyperNEAT la red CPPN debe calcular los retardos de las conexiones en el substrato, además de los pesos correspondientes a ellas.

7.1. FUNCIONES DE DESEMPEÑO

Con el objetivo de lograr medir correctamente el desempeño de las caminatas de cada robot es que se observarán dos variables importantes en cada simulación, la frecuencia de oscilación de cada uno de los motores y la máxima distancia alcanzada por el robot. Ambas variables son medidas los últimos cinco segundos de simulación, debido a que es posible que ocurran acciones no deseadas del robot inmediatamente al inicio de cada simulación, evitándose así lecturas erróneas de dichas variables.

La frecuencia de cada motor es calculada observando los cambios en el signo de la pendiente en el movimiento de este entre una iteración y otra, siendo un cambio de signo entre iteraciones consecutivas un cambio en la dirección en que se mueve el motor. Así dos cambios de dirección en el movimiento de un motor indicarán el suceso

de un periodo completo de señal de dicho motor. Una vez obtenida la frecuencia de cada uno de los motores se promedian las frecuencias correspondientes a motores de la misma extremidad y dicho promedio es ingresado a la función del cálculo del desempeño de la caminata según la frecuencia, mostrada en la figura 7.2a. Por último el desempeño final correspondiente a la frecuencia estará dado por el promedio de las salidas de la función de desempeño de frecuencias de cada extremidad.



(a) Función de desempeño de la frecuencia. (b) Función de desempeño de la distancia.

Fig. 7.2: Funciones de desempeño del entrenamiento. En esta figura se muestran las funciones de desempeño usadas para calificar que tan exitosa fue una caminata en el entrenamiento.

La distancia final alcanzada por el robot durante la simulación es calculada por la diferencia de distancia entre el punto donde se posiciona el robot en el segundo uno de simulación y el punto final que alcanza. Luego dicha distancia es ingresada a la función del cálculo del desempeño de la caminata según la distancia alcanzada, mostrada en la figura 7.2b.

El resultado final del desempeño de la caminata de un robot durante una simulación estará dado por una función multi-objetivo compuesta por las dos funciones mostradas anteriormente, en donde prevalecerá el menor resultado entre ellas con el fin de evolucionar las CPPNs forzando a mejorar siempre el resultado de la variable con más problemas. Así si el desempeño de la frecuencia y la distancia, por ejemplo, da como resultado 6 y 4 respectivamente, el desempeño final del robot será de 4. Fi-

nalmente, el resultado obtenido será asignado a la red CPPN usada en la respectiva simulación. Una vez comprendido el funcionamiento de cada simulación y el cálculo del desempeño en cada una de estas es posible entender la estructura básica de un entrenamiento, como se muestra en la siguiente sección.

7.2. ESTRUCTURA BÁSICA DE UN ENTRENAMIENTO

Un entrenamiento comienza con una población o grupo inicial de CPPNs, cuyo número es indicado por el usuario con anterioridad, conformando la primera generación de redes a entrenar. Cada una de estas redes creará un patrón de conectividad en el substrato de HyperNEAT y τ -HyperNEAT y se probará en una simulación para medir su desempeño. Una vez que se midieron y evaluaron todas las CPPNs de la primera generación deberán evolucionar en una nueva generación de redes CPPN, a través del método NEAT usado en ambos métodos, del mismo tamaño de la población inicial. Las CPPNs que conforman esta nueva generación deberán ser nuevamente probadas y evaluadas según su desempeño para dar origen a la generación siguiente. Esto debe realizarse tantas veces como generaciones haya establecido el usuario para el entrenamiento dado. De esta forma si el usuario, por ejemplo, definió al inicio del entrenamiento un número de 100 generaciones con una población de 100 CPPNs por generación, el entrenamiento finalizará una vez probadas las 10 mil CPPNs que conforman la totalidad de las generaciones. El proceso descrito anteriormente es la base de todo entrenamiento usando los métodos HyperNEAT y τ -HyperNEAT para cualquier experimento dado. En la sección siguiente se ahondará en el funcionamiento del programa de entrenamiento, escrito en lenguaje C/C++, que hace uso de las herramientas presentadas anteriormente para generar caminatas en robots con extremidades móviles.

7.3. PROGRAMA DE ENTRENAMIENTO

Una de las grandes problemáticas presentes en los entrenamientos que comprometen periodos de tiempo real para su ejecución es la extensa duración de estos. El entrenamiento para la generación de caminatas en robots con extremidades móviles es uno de estos, ya que cada una de las simulaciones del entrenamiento debe durar los 6 segundos definidos para esta. Así, si un entrenamiento esta estipulado para evolucionar una población de 100 CPPNs durante 100 generaciones, el tiempo estimado para su realización sobrepasa las 16 horas.

Debido a esta razón es que el programa de entrenamiento tiene la posibilidad de usar hebras (thread en inglés) para acelerar su ejecución. El uso de hebras permitirá disminuir el tiempo total usado para el entrenamiento, ya que de esta forma será posible utilizar simultáneamente tantos simuladores como hebras corra el programa¹. De esta forma si el usuario, por ejemplo, indica el uso de 2 hebras, la población de cada generación será dividida en 2, ejecutando cada grupo de CPPNs en un simulador distinto y dividiendo el tiempo de entrenamiento a la mitad.

Una vez iniciado el programa de entrenamiento, este creará todos los objetos necesarios para controlar al robot a usar dentro del simulador, usando la ya mencionada librería RobotLib. De usar más de un simulador, se deberán crear tantos objetos repetidos como simuladores se ejecuten. También se deben crear los objetos correspondientes al método a utilizar (HyperNEAT o τ -HyperNEAT), que al igual que en el caso de RobotLib, deben corresponder en número a la cantidad de simuladores o hebras estipuladas para el entrenamiento.

Ya creadas todas las entidades necesarias para el funcionamiento de los métodos de neuroevolución y de los simuladores el programa continuará con un bucle for que iterará tantas veces como generaciones se hayan estipulado para el entrenamiento, y en cada ciclo de este se iniciarán las hebras que se dividirán la población de CPPNs

¹ Si bien una computadora actual puede ejecutar una gran cantidad de hebras simultáneamente, no podrá ejecutar la misma cantidad de simuladores debido a que la demanda de recursos generada por estos es elevada.

para ejecutar las simulaciones correspondientes a cada una de ellas. Luego que cada una de las hebras termine de probar y evaluar cada unas de las CPPNs asignadas, estas terminarán su ejecución volviendo el programa al bucle for para posteriormente evolucionar la población de CPPNs en una nueva generación y completar nuevamente un nuevo ciclo del bucle. Una vez realizados todos los ciclos del bucle for se obtendrá la red CPPN capaz de crear el patrón de conectividad más adecuado sobre el substrato del método usado, logrando la caminata con el mejor desempeño del entrenamiento.

Para comprobar el funcionamiento de HyperNEAT y τ -HyperNEAT en la generación de caminatas en robots con extremidades móviles se entrenarán a 3 robots distintos: Quadratot, ArgoV2 y Rosita (ver Figura 1.1), siendo el primero un robot diseñado por el Investigador de la Universidad de Chile Juan Cristóbal Zagal, con el objeto de ser usado para comprobar el funcionamiento de distintos métodos para la generación de caminatas; y los últimos dos, robots diseñados por el estudiante de Ingeniería en Diseño de Productos Cristian Osorio Méndez para el Departamento de Electrónica de La Universidad Técnica Federico Santa María. En la sección siguiente se mostrarán los resultados de los entrenamientos realizados sobre cada uno de estos robots usando ambos métodos de neuroevolución. Todos los códigos utilizados en la siguiente sección pueden ser descargados desde el repositorio Github <https://github.com/osilvam/Memoria> y probados de manera fácil y sencilla.

7.4. RESULTADOS DE LOS ENTRENAMIENTOS

Tal como se mencionó anteriormente, los entrenamientos para la generación de caminatas usando los métodos de neuroevolución HyperNEAT y τ -HyperNEAT se aplicarán en 3 plataformas robóticas, Quadratot, ArgoV2 y Rosita. A continuación se mostrarán los resultados obtenidos para cada una de ellas.

7.4.1. QUADRATOT

HyperNEAT jkwnajkw

τ -**HyperNEAT** jkwnajkw

Bibliografía

- [1] STANLEY, K.O., D'AMBROSIO, D., and GAUSI, J. (2009). “A hypercube-based encoding for evolving large-scale neural networks”. *Artificial Life*, 15(2):185-212.
- [2] STANLEY, K.O., and MIIKKULAINEN, R. (2002). “Evolving neural networks through augmenting topologies”. *Evolutionary Computation*, 10(2):99-127.
- [3] ZYKOV, V., BONGARD, J., and LIPSON, H. (2004). “Evolving Dynamic Gaits on a Physical Robot”, *Proceedings of Genetic and Evolutionary Computation Conference, Late Breaking Paper, GECCO*.
- [4] HORNBYS, G.S., TAKAMURA, S., YAMAMOTO, T., and FUJITA M. (2005). “Autonomous Evolution of Dynamic Gaits with Two Quadruped Robots”.
- [5] CLUNE, J., BECKMAN, B.E., OFRIA, C., and PENNOCK, R.T. (2009). “Evolving coordinated quadruped gaits with the HyperNEAT generative encoding”, In *Proceedings of the IEEE Congress on Evolutionary Computing*.
- [6] YOSINSKI, J., CLUNE, J., HIDALGO, D., NGUYEN, S., ZAGAL, J.C., and LIPSON, H. (2011). “Evolving robot gaits in hardware: the HyperNEAT generative encoding vs. parameter optimization”, In *Proceedings of the 20th European Conference on Artificial Life*.
- [7] LEE, S., YOSINSKI, J., GLETTTE, K., and CLUNE, J. (2013). “Evolving Gaits for Physical Robots with the HyperNEAT Generative Encoding: The Benefits of Simulation”.

- [8] CAAMAÑO, P., BELLAS, F., and DURO, R. (2014). “ τ -NEAT: Initial experiments in precise temporal processing through neuroevolution”, International Joint Conference on Neural Networks.
- [9] Virtual Robot Experimentation Platform, Coppelia Robotics, Switzerland.
<http://www.coppeliarobotics.com/>
- [10] Deep learning, From Wikipedia, the free encyclopedia.
https://en.wikipedia.org/wiki/Deep_learning
- [11] RICHARD S SUTTON and ANDREW G BARTO. Reinforcement learning: An introduction, volume 1. Cambridge Univ Press, 1998.
- [12] CHURCHLAND, P.M. (1986). Some reductive strategies in cognitive neurobiology. *Mind*, 95:279-309.
- [13] Dynamixel Motors, Robotis, Korea.
http://www.robotis.com/xe/dynamixel_en