Fundamentos teóricos y Manual del Simulador IRSIM

Álvaro Gutiérrez & Félix Monasterio-Huelin

23 de enero de 2019

Índice general

Parte I

Teoria

Capítulo 1

Arquitectura de Procesos Distribuidos

1.1. Introducción.

El libro de Valentino Braitenberg titulado "Vehicles. Experiments in Synthetic Psychology" (1984) es una fuente de inspiración no sólo para la construcción de robots sino también para la comprensión de fenómenos relacionados con el comportamiento de las máquinas y de los animales.

En lo que sigue no vamos a ser suficientemente fieles con el autor, sino a extraer analíticamente unos principios generales básicos que por permita introducir la problemática del diseño de robots, adoptanto, eso sí, el enfoque sintético. Éste no sólo debe entenderse como "sintetizar" máquinas, sino como método sintético en oposición al método analítico. Si acaso sintetizaremos comportamientos de modo sintético, aunque los únicos experimentos que hagamos en esta asignatura sean de simulación.

Cualquier máquina se construye parte por parte, conectando unas con otras. Es esto lo que Braitenberg hace, aunque su finalidad sea mostrarnos que la idea de diseño va acompañada de la intención de dotar al robot de una potencialidad que se exprese en diferentes modos de comportamiento. Se trata de construir robots expresivos. Si un robot se acerca a la luz no debe interpretarse solamente como un comportamiento de aproximación sino también como la manifestación de una afectividad, su amor u odio hacia la luz, su deseo de luz, amoroso o agresivo. De esta manera amplía la misma noción de comportamiento tradicionalmente entendida como algo puramente externo al robot, puesto que no solo integra al robot en el medio, sino que busca en sus entrañas alguna clase de necesidad. Lo que está en juego para el robot no es algo trascendente sino inmanente, es lo que puede hacer el robot, su capacidad de afectar y de ser afectado. En este sentido nos parece spinozista.

En los primeros vehículos Braitenberg no se limita a la taxis, es decir a

los movimientos de orientación y desplazamiento en respuesta a estímulos externos (aproximación a la luz), sino a la quinesis o kinesis, es decir al movimiento que depende de la intensidad del estímulo y de sus variaciones, pero que no provoca un movimiento direccional. Arbib, en el prólogo del libro señala este hecho en relación con una publicación de Braitenberg de 1965, lo que es una observación interesante de las motivaciones teóricas de Braitenberg, de su concepción del mundo vivo.

Braitenberg propone catorce vehículos a lo largo de 80 páginas, pero escribe una segunda parte de más de 50 páginas, titulada "notas biológicas sobre los vehículos" que comienza con esta frase: "The preceding phantasy has roots in science". Si volvemos a la primera página del libro comienza así: "This is an exercise on fictional science, or science fiction, if you like that better". Braitenberg es neuroanatomista pero nos aporta a los ingenieros una mirada que creemos que debe ser muy estimada, y así se refleja en libros y artículos que intentan abrir un espacio a las investigaciones de una Nueva Inteligencia Artificial o como ahora se llama, un espacio a la Vida Artificial.

1.2. Vehículo I de Braitenberg.

El **Vehículo I** dispone de una conexión directa de un sensor de temperatura a un motor que mueve una única rueda, pero es una conexión que realiza, vista desde un observador externo, lo que llamaremos un **proceso**.

Braitenberg define el robot de la siguiente forma: **el Vehículo I se mueve más despacio en las regiones frías y más rápido en las regiones cálidas**. Se trata de una definición cinética (quinesis).

Enseguida nos advierte Braitenberg que la fricción con el suelo (rugosidad) puede provocar desplazamientos aleatorios que lo saque de la línea recta a pesar de que solo dispone de una rueda.

Matemáticamente puede representarse por una función que relaciona el vector velocidad con la temperatura (que Braitenberg llama cualidad), a la que se le añade una perturbación aleatoria vectorial debida a la fricción de la rueda con el suelo:

$$\begin{aligned} v_x &= f(T, n_x) \\ v_y &= f(T, n_y) \end{aligned} \tag{1.1}$$

donde T es la temperatura captada por el sensor, v_x y v_y son las velocidades del robot en la dirección x e y del plano del suelo respectivamente, y n_x y n_y las perturbaciones en estas dos direcciones. Es todavía una representación incompleta ya que habría que indicar la monotonía creciente de la función $f()$ en relación con la temperatura.

Braitenberg nos explica que la conexión directa del sensor al motor de la rueda supone una transformación de temperatura en fuerza. No nos habla de cómo ocurre esto, pero con la terminología de la física parece evidente

que el Vehículo I es una máquina que transforma energía térmica en energía mecánica. Pero también está implícito que esta energía mecánica es utilizada íntegramente en el desplazamiento, o en todo caso se traduce en una fuerza ejercida por el motor a la rueda que debe, a su vez, superar la fricción del suelo para poder desplazarse. La explicación de Braitenberg es dinámica.

Debido a que hay un único motor solo podremos establecer una relación escalar entre la magnitud del sensor de temperatura y el par o fuerza que el motor puede realizar:

$$\tau = g(T) \tag{1.2}$$

A esta función $g()$ la llamaremos **función de activación** o alternativamente **proceso interno** o intrínseco.

El proceso interno no queda completamente definido por esta función ya que hay muchos tipos de motores. Por ejemplo los motores eléctricos transforman energía eléctrica en energía mecánica (el par motor). También habría que fijarse en el sensor como transductor de energía térmica en energía eléctrica. La introducción en la descripción del proceso de la forma de energía que circula internamente no es importante para Braitenberg. No es importante la clase de movientos internos, rápidos y lentos que haya entre partículas (electrones, por ejemplo) (la cinética interna), sino de sus efectos dinámicos, de la relación de fuerzas que vincula al robot a su entorno.

Por otro lado sabemos de la Mecánica Clásica que el par motor es proporcional a la aceleración angular de la rueda, es decir que hay una relación entre los componentes dinámicos y cinemáticos. La integración de la aceleración angular permitiría obtener la velocidad angular de la rueda, lo que a su vez permitiría conocer la velocidad lineal del robot en ausencia de fricciones con el suelo. Por lo tanto existe una relación entre el proceso interno y el externo que sería determinístico si no fuese por la fricción.

Un robot concreto podría tener el siguiente comportamiento, visto desde un observador externo

$$\begin{aligned} v_x &= kT + n_x, \quad k > 0 \\ v_y &= k'T + n_y, \quad k' > 0 \end{aligned} \tag{1.3}$$

donde k, k' son constantes reales positivas.

Y otro robot concreto podría tener el siguiente comportamiento,

$$\begin{aligned} v_x &= k_1 T + k_2 T^2 + k_3 \tfrac{dT}{dt} + n_x, \quad k_i > 0 \\ v_y &= k'_1 T + k'_2 T^2 + k'_3 \tfrac{dT}{dt} + n_y, \quad k'_i > 0 \end{aligned} \tag{1.4}$$

pero internamente las relaciones podrían ser

$$\tau = \alpha T, \quad \alpha > 0; \tag{1.5}$$

o

$$\tau = \alpha_1 T + \alpha_2 T^2 + \alpha_3 \frac{dT}{dt}, \quad \alpha_i > 0; \tag{1.6}$$

Aunque Braitenberg no expone sus ideas con expresiones matemáticas, sino tan sólo descriptivamente, quedan abiertas muchas otras posibilidades, como por ejemplo representar los procesos mediante un sistema de ecuaciones diferenciales no lineales dependientes de la temperatura. Las funciones de activación entonces serían más complejas. Por ejemplo, $\tau = g(T, \frac{dT}{dt})$ siendo $g()$ alguna función no lineal.

De manera general un proceso es un sistema dinámico (de entrada/salida) con perturbaciones, lo que en la terminología matemática se denomina **proceso estocástico**.

El objetivo de Braitenberg no es hablarnos de los procesos estocásticos, sino de la potencialidad de los robots en términos de sus posibles comportamientos en un determinado medio. Si analizamos o miramos las entrañas del robot no encontraremos nada que nos informe detalladamente sobre los movimientos observados, tan sólo podremos descubrir la potencialidad del robot en relación con los entornos en los que pueda encontrarse: entornos térmicos (cualidad sensorial) y entornos con fricción (relación de fuerzas). Esta potencialidad expresaría un modo de comportamiento o más exactamente un modo de afección. En este sencillo Vehículo I podríamos deducir este modo de afección, pero si su complejidad aumentase el análisis se volvería más y más difícil. Clasificar a los robots por sus modos de afección es a lo que parece invitarnos Braitenberg.

1.3. Vehículos II a IV.

El **Vehículo II** de Braitenberg tiene dos ruedas actuadas por motores independientes, y dos sensores de luz, conectados cada uno de ellos directamente a cada una de las ruedas mediante funciones de activación similares a las del Vehículo I, es decir que a mayor intensidad de luz, mayor par aplicado a las ruedas, y en consecuencia se moverán a mayor velocidad angular. Puesto que podemos realizar dos conexiones posibles, según la posición de los sensores en el robot, podemos distinguir dos clases de robots que tendrán comportamientos distintos.

Desde el primer momento se presupone la existencia de un robot con un cuerpo morfológicamente constituido: una rueda derecha y una rueda izquierda, un sensor de luz en la parte frontal derecha y otro en la parte frontal izquierda. Se presupone una conectividad específica: una conexión lateral directa o una conexión contralateral (cruzada) directa entre sensores y motores (ver la Figura 1.1)[1]. Se presupone también la existencia de funciones de activación específicas, monótonas crecientes.

En resumen, un robot queda definido por:

1. Una morfología (M): una forma específica en la que se sitúan los sen-

[1] Todas las figuras del texto se han copiado del libro de Braitenberg.

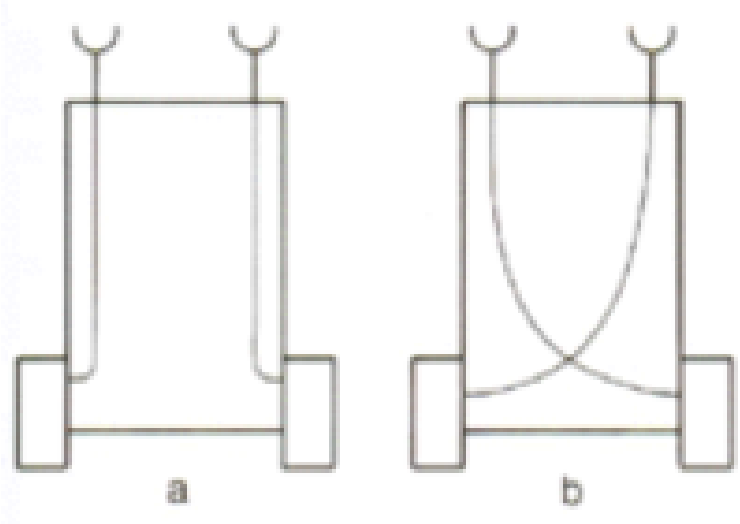


Figura 1.1: Conexiones directas lateral y contralateral.

sores y motores.

2. Una conectividad (C): un conjunto de sensores y motores con una conexión directa

3. Una relación funcional (F) o fisiológica entre sensores y motores.

Puede quedar claro que Braitenberg tiene una visión orgánica de los robots, pero a su vez estos organismos no son seres aislados del mundo sino que su constitución fisiológica está orientada a relacionarse con el medio. Es decir que los únicos órganos en los que se está interesado son órganos sensorio-motrices, y en este sentido su investigación consiste en plantear una metodología de diseño de **cerebros artificiales**. Hasta ahora hemos visto un cerebro muy limitado: un simple cable que propaga una señal desde un sensor hasta un motor.

Uno de los objetivos de Braitenberg es diseñar un cerebro más y más complejo, pero sobre todo un robot definido por la terna (M, C, F) cuyo aumento de complejidad se corresponda con una mayor riqueza de comportamientos. Braitenberg no tiene ningún escrúpulo a la hora de explicar esta riqueza con mecanismos generadores de lo que él llama trenes o series de pensamientos. En este escrito, no obstante no estudiaremos todas estas cuestiones ya que exceden el contenido de la asignatura.

Antes de entrar a describir el comportamiento de las variedades de Vehículos II, introduciremos los **Vehículos III**. Estos se diferencian de los anteriores en que las funciones de activación operan de manera inversa, es decir que a mayor intensidad de luz se produce una disminución en el par

que producen los motores. Las funciones de activación son ahora monótonas decrecientes.

Llamaremos **polaridad** a esta distinción. Por lo tanto los vehículos II tienen polaridad positiva o "+" y los vehículos III polaridad negativa o "−". No obstante, como esta diferencia, junto con los restantes elementos del robot permiten una generalización, los vehículos III de Braitenberg son máquinas con un conjunto variado de sensores (cualidades) conectados de diferentes formas, aunque directamente, a los motores, y con diferentes polaridades.

El Vehiculo II tiene polaridad positiva. Si la conexión es lateral tenderá a moverse alejándose de la fuente de luz, y si la conexión es contralateral, tenderá a moverse en dirección a la fuente de luz. Braitenberg llama **robot cobarde** al primero y **robot agresivo** al segundo, ya que el primero se alejará rápidamente de la fuente de luz y el segundo se aproximará violentamete hacia dicha fuente.

Algunos de los Vehículos III, equivalentes a los vehículos II tienen polaridades negativas, sin embargo son atraidos por la luz, les agrada la luz, aunque uno se acerque a ella deteniéndose suavemente sobre ella, y el otro se aleje deteniéndose suavemente en sus proximidades. Al robot que se aleja (el de conexión contralateral) lo llama Braitenberg **robot explorador**, "ya que mantiene un ojo abierto a otras fuentes de luz".

Podemos imaginarnos mentalmente los movimientos de estos cuatro robots en relación a una fuente de luz. Podemos utilizar un truco que consiste en asociar a cada rueda un vector, una flecha, de mayor o menor amplitud según que el robot esté a mayor o menor distancia de dicha fuente. Y por fin podemos aplicar una regla de tranformación de los dos vectores (uno de cada rueda) en un único vector de mayor o menor amplitud situado en el centro del robot que indique su orientación en relación a la fuente de luz. Sabemos que matemáticamente esto supone construir una representación geométrica, una curva en el plano. Sin embargo esta representación no nos informa adecuadamente de la amplitud o intensidad del vector de orientación ya que está relacionado con la aceleración. Para ello sería necesario construir otra curva que representase la diferencial de los vectores de orientación. También podría hacerse incorporando una nueva dimensión o eje ortogonal al plano, lo que supondría una representación tridimensional de una curva. O también puede hacerse dibujando una curva en el plano de colores o tonalidades de grises dependientes de la intensidad del vector de orientación. Lo que suele hacerse a veces es superponer a la curva de la trayectoria vectores tangentes de diferente amplitud, separados una pequeña distancia para que la gráfica no quede demasiado emborronada. El problema es cómo hacer una representación gráfica fácilemnte interpretable de un campo de fuerzas o de un campo de vectores ya que además hay que tener en cuenta las condiciones iniciales.

Puede observarse con estos dos tipos de robot, que la polaridad y la conectividad son variables de diseño independientes en cuanto a la forma o

modo de comportarse el robot: la aproximación a una fuente de luz se realiza con robots $(M, CL, +)$ y $(M, L, -)$, donde "CL" significa conexión contralateral y "L" conexión lateral. Pero la forma o modo de aproximación es distinta: el primero valora negativamente la luz (se aproxima violentamente) y el segundo la valora positivamente (se aproxima suavemente), según la interpretación de Braitenberg.

Los vehículos III generales (diversos tipos de sensores: de luz, de temperatura, de concentración de oxígeno, etc.), parecen tener un **sistema de valores**, lo que, para Baitenberg, podría interpretarse como el **conocimiento intrínseco del robot**, un conocimiento del que dispone el robot desde su nacimiento. No es un conocimiento concreto de algo sino un conocimiento expresado como una potencialidad que se actualizaría en la práctica, pero con la que nace el robot. En términos clásicos diríamos que es una capacidad para conocer cosas concretas. Pero a su vez, para Baitenberg, el conocimiento es de carácter evaluativo, es decir que el robot solo conoce algo en función del efecto de sus propios comportamientos. Podríamos decir que se trata de un **robot ético** aunque claramente egocéntrico. Es evidente que estas mismas ideas podrían suponer valoraciones que tomasen en consideración a otros robots, lo que permitiría hablar de **robots altruistas** o combinaciones entre egoismo y altruismo. Braitenberg no estudia estas posibilidades ya que se limita a un único robot (excepto con el Vehículo VI cuya finalidad es introducir un sencillo mecanismo se selección darwiniana, que puede aplicarse a poblaciones de robots), pero sus ideas pueden extenderse a **colectividades de robots**, en los que la evaluación de los comportamientos de otros robots pueda hacer variar los propios comportamientos.

La polaridad es una noción que matemáticamente se expresa mediante funciones monótonas crecientes o decrecientes. Sin embargo es posible imaginar robots cuyas funciones de activación presenten máximos o mínimos, o sean continuas o discontinuas. Esta generalización conduce a los **Vehículos IV**. Este tipo de vehículos se guían por **instintos**, entendiendo que un instinto se corrresponde con un sistema de valores de intensidades variables, es decir un **sistema de motivaciones y gustos**.

Un robot de este tipo dotado simplemente de sensores de luz puede realizar un movimiento circular alrededor de una fuente de luz externa si la función de activación tiene un máximo en algún punto, es decir si funciona con una polaridad positiva para valores de intensidad de la luz bajas y con polaridad negativa para intensidades elevadas. Si la conexión es contralateral, se acercará a la fuente de luz cuando esté lejos de ella, y se alejará de ella cuando esté más próximo.

Si el anterior robot dispusiese además de otros sensores con funciones de activación diversas, el conjunto de comportamientos que podrían realizar sería muy variado, hasta el punto de que resultaría difícil, si no imposible, para un observador externo deducir esta variedad de comportamientos a partir del análisis de los elementos del robot. La dificultad de **segmentar los**

comportamientos mediante técnicas analíticas es una de las características principales de los vehículos de Braitenberg. Para este investigador, estos vehículos IV no sólo están gobernados por instintos, sino que cualquier observador podría decir que pondera sus decisiones (si la función de activación fuese discontinua, con un valor de salida constante o casi nulo para un intervalo continuo de valores de entrada), y en consecuencia es como si tuviesen una voluntad para tomar decisiones. Debemos también no perder de vista que la segmentación en comportamientos del robot depende del observador ya que internamente no podemos encontrar estados o módulos específicos que los caractericen.

Hemos comentado cómo estos cuatro vehículos pueden generalizarse con funciones de activación representadas mediante ecuaciones diferenciales. El comportamiento final del robot se deberá a la participación de un conjunto de procesos que conectan sensores con efectores operando en paralelo. Esto sugiere el diseño de arquitecturas basadas en procesos.

1.4. El Vehículo V de Braitenberg

Un problema que no hemos planteado hasta ahora pero que resulta evidente es cómo integrar las diferentes vías sensoriales para producir una única salida. Este problema ya aparece en los Vehículos III, pero hay que tener en cuenta que Braitenberg plantea el problema desde la perspectiva de las fuerzas ejercidas por los motores, por lo que es de suponer, aunque no lo diga, que la fuerza total es la suma de las fuerzas.

No obstante matemáticamente se supone que existe una función adicional que relaciona un conjunto de sensores con una única salida, es decir

$$\tau_i = h(f_1, f_2, \ldots, f_n), \quad i = 1, 2 \tag{1.7}$$

donde el índice i-simo representa el motor de la rueda derecha ($i = 1$) e izquierda ($i = 2$) respectivamente, y f_j representan la función de activación del sensor j-simo, con $j = 1, 2, \ldots, n$.

Cuando hay más de un sensor la conexión no puede ser directa, sino que debe estar mediada por otra función que las combine de alguna forma. Una solución consiste en realizar una simple combinación lineal de las funciones de activación; otra solución es elegir alguna de ellas dependiendo de algún criterio de prioridades. Podemos imaginar otras combinaciones introduciendo la noción de flujos continuos o discretos, flujos sensoriales que se encuentran y se fusionan, convergentes o divergentes, con realimentaciones motrices o reflujos, que en última instancia activan los motores. Los procesos pueden ser altamente complejos, no solo por su estructura sino porque están funcionando en paralelo. La ecuación 1.7 no refleja realmente las posibilidades de las arquitecturas basadas en procesos.

Braitenberg propone un nuevo tipo de vehículos, el **Vehículo V**, al que incorpora **dispositivos de umbral** cuyas entradas pueden ser de **excitación** o de **inhibición**. Un dispositivo de umbral puede tener muchas entradas y una única salida, pero siempre realiza una operación sencilla. Los dispositivos de umbral se conectan unos a otros formando una red interconectada cuya salida final actúa sobre los motores de las ruedas. No obstante los dispositivos de umbral responden o producen una salida tras un breve **retardo**, es decir que su estado anterior se mantiene durante un breve periodo de tiempo, o dicho de otra forma las entradas y salidas son pulsos de una determinada duración, y no impulsos (deltas de Dirac). En términos matemátios pueden representarse como ecuaciones en diferencias temporales o más propiamente como ecuaciones diferenciales con retardos.

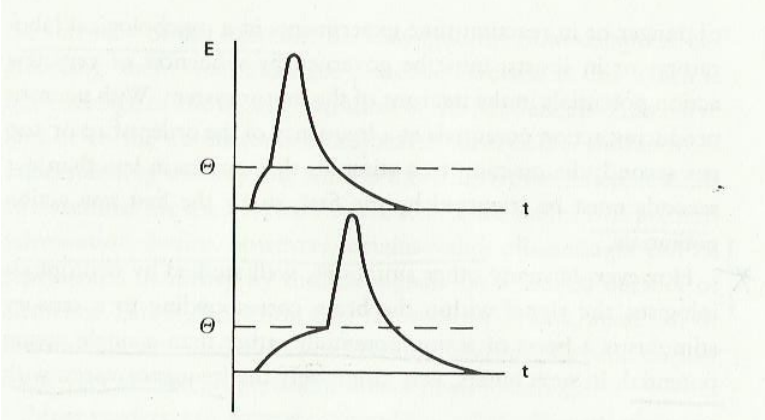


Figura 1.2: Evolución temporal del estado de dos dispositivos de umbral con diferentes retardos.

Braitenberg señala explícitamente su relación con las neuronas de McCulloch y Pitts, pero se distancia de ellas al introducir características de las neuronas reales: no se trata solo de cuándo llega a una neurona la señal excitadora, sino también de la variación en el tiempo de la intensidad de la excitación (ver Figura 1.2). El Vehículo V responde a esta última característica con la idea de retardos en la activación de los dispositivos. Un retardo largo se correspondería con una lentitud en el aumento de la señal de excitación hasta alcanzar un determinado umbral. La curva continua excitación-tiempo es la carcaterística que Braitenberg añade a la neurona formal de McCulloch y Pitts.

Sea un dispositivo de umbral D, con cuatro entradas (e_1, e_2, e_3, e_4) de las cuales las dos primeras son de excitación y las dos segundas de inhibición. La salida de este dispositivo será:

$$y = \begin{cases} 1, \texttt{si} & s \geq \theta \\ 0, \texttt{si} & s < \theta \end{cases} \tag{1.8}$$

donde

$$s = \omega_1 e_1 + \omega_2 e_2 - \omega_3 e_3 - \omega_4 e_4 \tag{1.9}$$

siendo ω_i pesos que toman valores en el intervalo $[0, 1]$, y θ el umbral del dipositivo que en general puede ser tanto positivo como negativo o nulo.

Braitengerg simplifica esta representación suponiendo que $\omega_i = 1$ para cualquier entrada. Además estos dispositivos dependen del tiempo por lo que realmente habría que escribir en las anteriores ecuaciones $e_i(t)$, $s(t + \Delta t)$ e $y(t + \Delta t)$, donde Δt representa el retardo.

Veamos un ejemplo. Supongamos una red de tres dispositivos conectados en serie excitadoramante y que reciben la misma entrada excitadora e, como se muestra en la Figura 1.3.

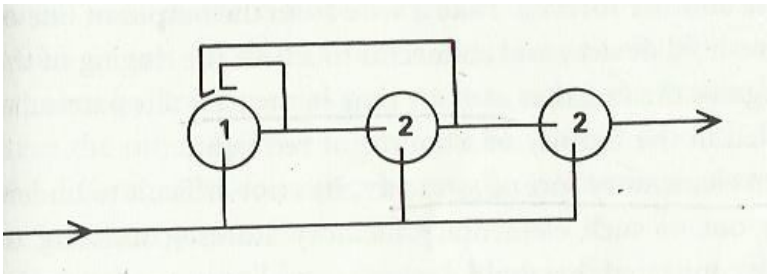


Figura 1.3: Ejemplo de una red de dispositivos de umbral que cuenta tres pulsos. (Ver detalles en el texto)

El primer dispositivo D_1, tiene un umbral $\theta_1 = 1$, y además recibe dos entradas inhibidoras adicionales: su propia salida y la salida del dispositivo D_2 (**realimentaciones**). Los dispositivos D_2 y D_3 tienen un umbral $\theta_2 = \theta_3 = 2$.

La salida de D_3 es la salida final de la red.

Supongamos también que el retardo en producir la salida es de una unidad de tiempo para todos los dispositivos (sea cual sea esta unidad de tiempo, que vendrá fijada en la práctica por un reloj común a todos los dispositivos).

Las ecuaciones en diferencias que definen esta red son:

$$\begin{array}{rl} s_1(k+1) = & e(k) - y_1(k) - y_2(k) \\ s_2(k+1) = & e(k) + y_1(k) \\ s_3(k+1) = & e(k) + y_2(k) \end{array} \tag{1.10}$$

y las salidas correspondientes serán 0 o 1 dependiendo del valor de umbral:

$$y_i(k+1) = \begin{cases} 1, \texttt{si} & s_i(k+1) \geq \theta_i \\ 0, \texttt{si} & s_i(k+1) < \theta_i \end{cases} \tag{1.11}$$

donde $i = 1, 2, 3$.

Supongamos que inicialmente $e = 0$ durante un tiempo suficientemente largo. Es fácil comprobar que las salidas y las entradas de cada dispositivo de la red serán nulas (dejamos esta demostración como ejercicio).

Supongamos que en el instante $k - simo$ se produce un cambio en la señal de entrada e, es decir $e(k) = 1$ durante un tiempo suficientemente largo. Vamos a demostrar que se produce una salida unidad por cada tres pulsos de entrada. Podemos decir que el robot cuenta tres pulsos.

Teniendo en cuenta las ecuaciones 1.10 las entradas en el instante $k - simo$ dependen de $e(k-1) = 0$, y en consecuencia serán nulas, por lo que

$$y_1(k) = y_2(k) = y_3(k) = 0$$

En la siguente iteración (al cabo de un breve periodo de tiempo que como se ha dicho es constante para todos los dispositivos), sin embargo

$$s_1(k+1) = s_2(k+1) = s_3(k+1) = 1$$

y de aquí que teniendo en cuenta los umbrales,

$$\begin{aligned} y_1(k+1) &= 1 \\ y_2(k+1) &= 0 \\ y_3(k+1) &= 0 \end{aligned}$$

De aquí que

$$\begin{aligned} s_1(k+2) &= 0 \\ s_2(k+2) &= 2 \\ s_3(k+2) &= 1 \end{aligned}$$

y en consecuencia

$$\begin{aligned} y_2(k+2) &= 0 \\ y_2(k+2) &= 1 \\ y_3(k+2) &= 0 \end{aligned}$$

En la siguiente iteración

$$\begin{aligned} s_1(k+3) &= 0 \\ s_2(k+3) &= 1 \\ s_3(k+3) &= 2 \end{aligned}$$

y de aquí

$$\begin{aligned} y_2(k+3) &= 0 \\ y_2(k+3) &= 0 \\ y_3(k+3) &= 1 \end{aligned}$$

En este momento se ha producido una salida unidad.

En el instante de tiempo siguiente

$$s_1(k+4) = s_2(k+4) = s_3(k+4) = 1$$

y de aquí que

$$y_1(k+4) = 1$$
$$y_2(k+4) = 0$$
$$y_3(k+4) = 0$$

salida que se corresponde con la que se ha producido en $(k+1)$. Puesto que las anteriores ecuaciones no dependen de y_3 la secuencia se repetirá mientras no cambie la señal de entrada.

En resumen. Cuando $e = 1$ esta red da un pulso unidad por cada tres pulsos consecutivos de la entrada, después de un breve periodo transitorio. Cuando $e = 0$ la salida será nula, por lo que podemos decir que la entrada e, que puede ser una entrada sensorial, **modula** el comportamiento del robot. Si el robot dispusiese de un sistema de valores, el comportamiento actual modularía, a través de la interacción con el entorno, el comportamieno futuro.

Una red más compleja de dispositivos de umbral permitiría realizar cualquier clase de operaciones con números enteros, y en consecuencia podríamos decir que es un computador. Un vehículo V podría, en consecuencia contar y realizar operaciones aritméticas y lógicas. Estaríamos ante un **robot computacional** pero entendiendo que las computaciones se realizan dependiendo de las entradas, por lo que de nuevo la noción de computabilidad debe redefinirse con respecto a la definición clásica, ya que no se trataría de hacer operaciones sobre símbolos sino que se trataría de procesos cuya dinámica parece realizar operaciones. Vemos que para Braitenberg lo que cuenta es la potencialidad del robot, si bien ésta se realiza o expresa o actualiza en un mundo concreto.

Braitenberg señala claramente este hecho. Por un lado observa que la cantidad de memoria esta limitada por el número de dispositivos de umbral: palabras de tantos dígitos como dispositivos de umbral. Por otro propone un truco para superar esta limitación que consiste en que el robot deje secuencialmente marcas en el suelo, marcas digitales que se correspondan con el exceso en la longitud de palabra (número de bits) que puede manejar computacionalmente el robot. Si el robot recorre hacia atrás y sobre sus propios pasos estas marcas la cantidad de memoria aumenta. El robot utiliza el entorno y depende de él para resolver problemas que exceden su capacidad computacional intrínseca. Obviamente esta ventaja solo es posible si el robot se desplaza y puede dejar marcas.

Se puede uno imaginar una generalización de esta idea con marcas dejadas por aquí y por allí que le sirvan al robot como **memoria externa**, como notas o balizas que él mismo ha puesto y que le facilitarían llegar a lugares que ya ha recorrido o recorrer lugares que todavía no ha explorado. En cierta forma podemos ver en esta idea una similitud con las feromonas de los insectos. Sin embargo esto no tiene nada que ver con el Vehículo V ya que la carcaterística de éste es su capacidad computacional y no la respuesta

reactiva sensorio-motriz a olores o sabores, que se correspondería más con los anteriores vehículos. Pero el truco de Braitenberg, como todo su libro, es fuente de inspiración, y no cabe duda de que todos los animales alteran su entorno de una manera o de otra. Introducir una memoria externa sería una forma de poder afectar y poder ser afectado. Esta memoria podría ser compartida por otros robots de la misma o de diferente especie.

Braitenberg pone un ejemplo interesante de Vehículo V. Supongamos que la entrada de los dispositivos de umbral es excitadora cuando pasa otro robot de color verde oliva que zumba a una determinada frecuencia y que nunca va más rápido de 5cm/s. La interpretación de Braitenberg es que el robot memoriza algo parecido a los **nombres propios**. Con dispositivos de umbral el robot reconoce a un amigo especial ya que podría estar horas sin moverse, pero ante la presencia de su amigo ("James, Calcuta o Júpiter") entrar en actividad. No hay que suponer que Braitenberg defiende un innatismo de nombres propios ya que más adelante muestra cómo pueden ser aprendidos variando los umbrales. No obstante sí parece haber un innatismo de dispositivos orientados a la formación de nombres propios, aunque siempre en una relación sensorio-motríz.

1.5. Arquitectura de procesos distribuidos

La función de activación $\tau_i = h()$ que definíamos antes puede tomar valores que sean la salida de una red de dispositivos de umbral cuyas entradas sean las salidas de los diferentes sensores, además de algunas realimentaciones internas. Estos son los vehículos V. Una modificación de este tipo de vehículos es el **Vehículo XII**, para los que existe un mecanismo de variación en tiempo real, y dependiente de los niveles de activación de la red, de los umbrales de cada dispositivo de umbral. No vamos a entrar aquí a analizar este nuevo tipo de vehículos, pero puede comprenderse que la simple modificación de los umbrales provocará comportamientos distintos. ¿Qué ocurre si en el ejemplo del apartado anterior se hace variar el umbral θ_3 de tal manera que cambie cada cinco pulsos (o cierto tiempo o cierto estado de la red) de $\theta_3 = 2$ a $\theta_3 = 1$? Si este criterio de variación del umbral dependiese de factores significativos para el comportamiento del robot, podría quedar claro que el comportamiento del robot dependerá de estos factores. Si los umbrales variasen en función de la actividad de las entradas sensoriales, podríamos decir que las entradas modulan los umbrales, y puesto que las entradas modulan a su vez, como hemos visto antes, las salidas, y las salidas dependen de los umbrales, las mismas entradas producidas en un instante de tiempo dado, podrían provocar comportamientos distintos en el robot, además de que estos comportamientos dependerían de la **secuencia temporal de las entradas**. En este sentido podemos decir que el robot tiene una **memoria** de patrones del entorno espacio-temporales que podrían ser muy útiles si el

robot tuviese capacidades exploratorias. El concepto de memoria ya no se entiende a la manera clásica de un almacén de datos, sino de relaciones entre entradas y salidas dependientes del tiempo. Es una memoria histórica, por lo que podríamos llamar a esta clase de robots **robot histórico**.

En primer lugar deberíamos preguntarnos qué podemos entender por "proceso". Braitenberg no utiliza esta palabra, pero tampoco parece que defina a los robots como máquinas o sistemas que procesan información. Un robot o incluso un organismo puede verse como una máquina que transforma diferentes tipos de energía (el Vehículo I es un ejemplo claro de transformación de energía térmica en mecánica), pero parece inevitable introducir alguna noción de complejidad. Braitenberg no especifica el tipo de energía que circula por el robot o mas bien parece que la forma de energía es irrelevante. Es más bien la relación funcional que se establece entre partes, en la que el tiempo y la intensidad juegan un papel importante, la que caracteriza a los vehículos. ¿Es posible traducir este lenguaje de procesos físicos a un lenguaje de procesos informacionales? No vamos a responder en este escrito. Solo señalaremos que el procesamiento de la información tiene por objetivo la utilización explícita de variables abstractas, es decir que el robot disponga de algún mecanismo de interpretación de su actividad interna. Por ejemplo una imagen en la retina tras pasar por un filtro paso alto o un mecanismo de inhibición lateral se puede transformar en otra cuyos pixeles activos representen los bordes de objetos. Cuando esta actividad es interpretada como "bordes de un objeto" a partir de los cuales, mediante una extrapolación se obtienen "contornos de objetos concretos" (circulares, triangulares, etc.), la actividad como tal ha pasado a un segundo plano y la interpretación o abstracción a primer plano. Lo mismo puede decirse cuando se utilizan variables como anchura de un objeto o distancia a un objeto aunque sean parámetros de una función de activación. En todos estos casos se está haciendo un uso simbólico de la actividad física. Braitenberg huye del procesamiento de la información en este sentido centrándose principalmente en la organzación y en su complejidad unida inseparablemente al funcionamiento. Desde otra perspectiva podríamos decir que el concepto de señal que Braitenberg utiliza no traspasa el nivel del significante de tal manera que el significado no está siendo utilizado ni implícita ni explícitamente en el funcionamiento de ninguno de sus vehículos. Además el significante no está en la naturaleza sino que depende de las capacidades de un cuerpo (sensorio-motriz) inmerso en un entorno. El ejemplo que Braitenberg pone sobre los nombres propios es una muestra de esto último.

Se han propuesto en la literatura muchas clases de aquitecturas basadas en procesos de las que no vamos a dar cuenta en este escrito. Uno de los grandes problemas que presenta la idea de procesos funcionando en paralelo es cómo articularlos de manera que los comportamientos de los robots presenten una **coherencia**, es decir que tengan algún sentido para el robot y para otros robots. Una de las ideas para abordar este problema consiste en

la separación de todo el sistema en subsistemas interconectados que reciben como entradas información sensorial y/o información de los estados internos de otros subsistemas. Se trata en este caso de realizar a priori una organización funcional del sistema introduciendo lazos de realimentación que regulen los diferentes procesos de cada subsistema.

Casi todas las soluciones dependen fuertemente del significado que se le de a la palabra "comportamiento". Si éste se define como la realización de tareas la arquitectura dependerá fuertemente del conjunto de tareas consideradas. Cada tarea puede definirse como una relación dinámica sensoriomotriz, y en consecuencia como un proceso, pero en general el conjunto de tareas pueden ser contradictorias entre sí, por lo que la activación simultánea de todos los procesos podría tener como consecuencia que no se resolviese ninguna de ellas.

Por otro lado la misma noción de tarea presupone una finalidad, un objetivo a alcanzar. La tarea puede consistir simplemente en desplazarse evitando obstáculos. La finalidad (supervivencia, exploración, por ejemplo) está implícita pero no tiene por qué estar implementada de manera explícita. Es decir que puede lograrse un fin sin que el robot se lo proponga, por lo que no parece oportuno llamar tarea a este modo de comportamiento. O en todo caso las tareas pueden corresponderse realmente con motivaciones, gustos o valoraciones (egocéntricas) cuya finalidad no debe, en general, entenderse como un objetivo a alcanzar sino que se alcanza un objetivo como efecto de las motivaciones o de un sistema de valores. La palabra "tarea" no hace justicia a esta diferencia teleológica. Para poder explicar estas cuestiones es necesario entender aspectos biológicos de la teoría de la evolución y del desarrollo que expliquen cómo es posible que en un mundo inmanente se hayan producido y reproducido organismos que parecen orientarse a fines.

La concepción de que los comportamientos se refieren a tareas también parece suponer que se está poniendo el énfasis sobre la acción: dadas unas condiciones sensoriales realizar una secuencia de acciones que logren culminar con éxito una tarea. Pero esta tarea también puede consistir en cambiar el punto de vista girando alrededor de un objeto. En este caso el énfasis estaría en la percepción: la secuencia de acciones tendría como finalidad percibirlas desde otro punto de vista, posiblemente para resolver un problema que puede presentar ambigüedades cuando se observa desde una única perspectiva. En ambos caso el problema lógico se centra en las condiciones de satisfacción, como diría Searle. La palabra "tarea" puede entenderse en términos lógicos, pero esta lógica debe ser entendida como el aspecto descriptivo de un comportamiento más que su aspecto explicativo.

Por último puede ocurrir que aunque la arquitectura dependa fuertemente de las tareas consideradas, el funcionamiento global del robot no se limite a estas tareas sino que emerjan otras como consecuencia de la conjunción de los procesos activos en el robot. En este caso sería mejor hablar de comportamientos que de tareas, pero al decir que son tareas queda más

claro que el comportamiento emergente tiene cierta utilidad adicional. Un ejemplo típico es la estigmergia, la resolución de tareas por una colectividad de insectos (robots) sin una planificaciń ni un poder centralizado.

Sin embargo no tiene por qué definirse el comportamiento como la realización de una tarea. Hemos visto que Braitenberg no lo plantea así, ya que como hemos señalado nos parece más acorde con una idea general de modos de afección, lo que también puede entenderse que no es el sutantivo comportamiento el concepto clave asociado a los procesos, sino el verbo comportarse en el doble sentido simultáneo de capacidad de afectar y ser afectado. Por ejemplo el verbo pasear no se reduce simplemente a desplazarse evitando obstáculos. Implica muchas veces cruzar calles por las que pasan coches, detenerse brevemente a charlar con algún conocido, etc. Pasear es algo indefinido en términos de tareas. Más bien parece que le acompaña alguna idea de acontecimiento. Ir a comprar el pan sin embargo sí parece estar definida como una tarea, aunque no escapa completamente a la idea de acontecimiento si se toma en consideración la historia o la génesis que ha llevado a "ir a comprar el pan", de la misma forma que uno puede decir que se asigna a sí mismo la tarea de conversar con un conocido encontrado casualmente durante un paseo por la calle. Desde el momento en que se entienda el comportamiento como un sustantivo asociado a una tarea, y aunque se nos presenten conjuntos de tareas contradictorias, no se estará tomando en consideración las potencialidades de un cuerpo. Si acaso tan solo sus posibilidades entendidas combinatoriamente, abstractamente.

Capítulo 2

Controladores de robots con Red Neuronal.

2.1. Introducción.

En este capítulo se introduce una clase de controladores de robots basados en redes neuronales artificiales. En primer lugar se explica en qué consiste una neurona matemática o formal, y posteriormente cómo pueden conectarse entre sí formando capas de neuronas.

2.2. La neurona y la red neuronal.

En general una neurona artificial es un sistema dinámico no lineal de múltiples entradas y una única salida.

Se representa mediante dos subsistemas conectados en serie. El primero recibe las entradas de la neurona, y el segundo genera la salida. El primero tiene una única salida, que se le denomina estado de la neurona, y que se obtiene como una combinación lineal de todas sus entradas[1]. A los parámetros de esta combinación lineal se les llama **pesos**. El segundo realiza una transformación no lineal, llamada **función de activación**, con la salida del primer subsistema. La función de activación depende de una función que se le suele llamar **sesgo** o **umbral**.

Una red neuronal consiste en un conjunto de neuronas interconectadas. A las conexiones se las llama sinaspis. Sus entradas pueden ser las salidas de otras neuronas (**conexión presináptica**) o las salidas de sensores. Sus salidas pueden conectarse a las entradas de otras neuronas (**conexión postsináptica**) o a las entradas de actuadores. De esta manera una red neuronal conecta sensores con actuadores a través de neuronas.

[1]En estos apuntes sólo consideraremos neuronas que se comporten dinámicamente como una simple ganancia, es decir cuya dinámica interna sea de orden cero. No se estudiarán, por lo tanto, las redes neuronales recurrentes.

Llamando $g()$ a la función de activación, la salida y_i de la neurona $i-sima$ será

$$y_i = g(x_i) \tag{2.1}$$

donde x_i es el **estado** de la neurona,

$$x_i = a_i\, c_i + \sum_{j=1}^{n} w_{ij} e_j \tag{2.2}$$

siendo $(n+1)$ el número de entradas, e_j la entrada $j-sima$ procedente de otras neuronas, c_i la entrada sensorial, a_i un factor de normalización (normalmente constante o nulo) y w_{ij} el peso de la neurona $i-sima$ asociado a la entrada $j-sima$.

Los pesos de las neuronas podrán tomar cualquier valor, aunque consideramos que siempre estarán en un intervalo continuo, es decir $w_{ij} \in [a_{ij}, b_{ij}]$.

En general las entradas a las neuronas pueden tomar cualquier valor, y así mismo pueden ser los valores de los pesos. Sin embargo es habitual considerar valores de entrada positivos. Cuando los pesos son positivos se considera que sus entradas corresponidentes son de carácter **excitador**, mientras que si son negativos son de carácter **inhibidor**. Por otro lado los límites en los valores de la entrada sensorial tendrán que ser coherentes con los límites de las restantes variables y parámetros de la neurona.

En este estudio se supondrá que una neurona no puede conectarse consigo misma.

No estudiaremos en estos apuntes el aspecto dinámico de la red neuronal. La secuencia temporal será la siguiente: primero se leen todos los sensores, segundo se procesa la red neuronal, y por último se actúa sobre los motores del robot. Puesto que el robot no se detiene, y la actuación supone tan sólo un cambio de velocidad, los retardos en el procesamiento de cada neurona de la red suponen un desajuste entre la posición real que ocupa el robot y la medida de los sensores. En la práctica, este retardo es pequeño y se considera despreciable. Si el periodo de muestreo es $T = 100$ ms, por ejemplo, y la velocidad del robot es $v = 5$ cm/s, el desplazamiento del robot entre medida y medida es de 0,5 cm. Lo que se está despreciando depende del submuestreo, por lo que la diferencia entre la posición real del robot y la medida del sensor cuando es utilizada por alguna neurona es menor que este desplazamiento.

Por esta razón entenderemos que la red neuronal funciona como un sistema de entrada/salida: todos los sensores (o entradas) se leen en el mismo instante de tiempo y cada vez que se produce una salida de la red neuronal. Por ello podemos escribir

$$y_i(k) = g(x_i(k)) = g(a_i\, c_i(k) + \sum_{j=1}^{n} w_{ij} e_j(k)),\ k = 0, 1, \ldots \tag{2.3}$$

donde $y_i(k)$ es la salida de la neurona i-sima de la red neuronal. Las neuronas tienen una respuesta instantánea que se propaga por toda la red hasta la salida.

Llamando o_j a la salida de la neurona j-sima que sea a su vez salida de la red neuronal, haremos $o_j(k+1) = y_j(k)$, entendiendo que el índice temporal k se refiere a los instantes de tiempo de entrada/salida de la red neuronal. En lo que sigue no se utilizará el índice temporal, salvo cuando sea imprescindible introducirlo.

En la literatura técnica se han definido una variedad de funciones de activación, y aunque sea habitual que las neuronas de una red neuronal lleven asociadas la misma función de activación, esto no tiene por qué ser así en general. En este estudio se admitirá una diversidad de funciones de activación. Esto exige que se introduzca en la notación funciones de activación indexadas, $g_i()$. Consideraremos solamente dos tipos generales de funciones de activación: las funciones de umbral y las funciones sigmoidales.

1. Función de umbral.

   En este caso la neurona solo toma dos valores de salida posibles, es decir $y_i \in \{m_{i1}, m_{i2}\}$,

   $$y_i = \begin{cases} m_{i2}, & x_i \geq \theta_i \\ m_{i1}, & x_i < \theta_i \end{cases} \tag{2.4}$$

   donde θ_i es el umbral de la neurona. Es habitual considerar el caso binario tal que $\{m_{i1}, m_{i2}\} = \{0, 1\}$ llamada función de Heaviside.

   También puede definirse la función de umbral trivaluada, $\{-1, 0, 1\}$ o función de signo:

   $$y_i = \begin{cases} 1, & x_i > \theta_i \\ 0, & x_i = \theta_i \\ -1 & x_i < \theta_i \end{cases} \tag{2.5}$$

2. Función (sigmoidal) logística.

   La función logística es una función continua que toma valores en $[0, 1]$,

   $$y_i = \frac{1}{1 + e^{-\mu(x_i - \theta_i)}} \tag{2.6}$$

   donde θ_i es el sesgo de la neurona, y μ un parámetro de pendiente constante.

   Se está admitiendo que el estado x_i puede tomar valores en $[-\infty, \infty]$.

   La salida puede, no obstante, renormalizarse para que tome valores en cualquier otro intervalo. Por ejemplo, en $[-1, 1]$ la función logística se transforma linealmente en otra función sigmoidal que es la función tangente hiperbólica,

   $$y_i = -1 + \frac{2}{1 + e^{-\mu(x_i - \theta_i)}} = \frac{1 - e^{-\mu(x_i - \theta_i)}}{1 + e^{-\mu(x_i - \theta_i)}} = tanh(-\mu(x_i - \theta_i)) \tag{2.7}$$

Puede observarse que dependiendo de los límites de los pesos de las neuronas, así como de los límites de las salidas de las neuronas a las que se conectan sus entradas, así serán los límites de sus estados. Las funciones de activación tienen un efecto de saturación no lineal del estado de las neuronas, además de que siempre dependen de un sesgo o umbral. Por esto a veces la escribiremos en la forma $g(x, \theta)$ en vez de $g(x)$ para que quede explícita la existencia del sesgo o umbral.

Conviene observar que todas estas funciones de activación pueden expresarse de otra manera, introduciendo el sesgo en el vector de estado en vez de hacerlo en la función de activación. Definimos el **potencial de acción** o también llamado **campo local inducido**, como

$$v_i = x_i - \theta_i \tag{2.8}$$

donde θ_i es el sesgo o umbral, por lo que la salida puede escribirse como

$$y_i = h(v_i) \tag{2.9}$$

Podemos seguir llamando a $h()$ funciones de activación, pero deberán definirse de otra forma:

1. Modelo de McCulloch-Pitts.

$$y_i = \begin{cases} 1, & v_i \geq 0 \\ 0, & v_i < 0 \end{cases} \tag{2.10}$$

2. Función logística.

$$y_i = \frac{1}{1 + e^{-\mu v_i}} \tag{2.11}$$

De esta manera podemos comprender que el sesgo o umbral constituye una entrada adicional a cada neurona, e_0, una entrada de valor unidad ($e_0 = 1$) cuyo peso correspondiente $w_{i0} = -\theta_i$. El potencial de acción representa una traslación del vector de estados. Con esta definición podemos escribir el potencial de acción en la forma:

$$v_i = a_i \, c_i + \sum_{j=0}^{n} w_{ij} y_j \tag{2.12}$$

en la que se ha escrito $y_0 = e_0$.

2.3. La capa neuronal

En estos apuntes consideraremos que una red neuronal se estructura en capas interconectadas. Cada capa consta de un conjunto de neuronas no conectadas entre sí que reciben como entradas las salidas de una o varias capas precedentes (neuronas presinápticas), y posiblemente disponga de una entrada sensorial adicional distinta para cada neurona. Las capas que tengan entradas sensoriales deberán tener conectados tantos sensores del mismo tipo como neuronas. Todas las neuronas de la misma capa llevan asociada la misma función de activación. Las salidas de cada capa o contituyen entradas de otras capas (neuronas postsinápticas) o se conectan a actuadores de motores, en cuyo caso será la capa de salida de la red neuronal.

El estado y la salida de una neurona $i-sima$ perteneciente a la capa $m-sima$, se escribirá como x_i^m e y_i^m respectivamente. En general pondremos el superíndice m en cualquier parámetro o variable para indicar la numeración de la capa. Con esta notación cada nuerona de una capa $m-sima$ puede representarse mediante la expresión,

$$x_i^m = a_i^m \, c_i^m + \sum_{j=1}^{N_m} w_{ij}^m y_j^{m'}, \quad i = 1, \ldots, n_m \tag{2.13}$$

$$y_i^m = g_i^m(x_i^m) \tag{2.14}$$

donde $m' \neq m$ representa el número asociado a la capa o capas precedentes, N_m es el número de entradas y n_m es el número de neuronas de la capa $m-sima$.

Estas ecuaciones pueden representarse matricialmente. El vector de estados x^m de la capa $m-sima$ toma la forma

$$x^m = A^m \, c^m + W^m y^{m'} \tag{2.15}$$

donde A^m es una matriz diagonal de dimensión $[n_m \times n_m]$, W^m es la matriz de pesos de la capa $m-sima$ de dimensión $[n_m \times N_m]$, x^m es el vector de estados de dimensión $[n_m \times 1]$ (matriz columna) y c^m es el vector de entradas sensoriales de dimensión $[n_m \times 1]$ (matriz columna).

La ecuación del vector de salida y^m de la capa $m-sima$ toma la forma

$$y^m = g^m(x^m, \theta^m) = h^m(v^m) \tag{2.16}$$

donde θ^m es el vector de sesgos o umbrales de las neuronas de la capa $m-sima$, y v^m es el potencial de acción, como se ha definido en el apartado anterior.

Distinguiremos cuatro tipos de capas neuronales:

1. Capa de entrada o capa sensorial.

Solo reciben entradas sensoriales, y sus salidas son sus mismas entradas, aunque posiblemente afectadas por una ganancia o una renormalización. En este caso las entradas e_j son nulas. El estado de cada neurona de la capa $m-sima$ será de la forma $x_i^m = a_i^m c_i^m$, o en forma matricial $x^m = A^m c^m$. Pero, en general, la salida será $y_i^m = g_i^m(x_i^m)$, para alguna función de activación g_i^m.

2. Capa asociativa.

   Es una capa cuyas entradas son las salidas de otras capas, y que a su vez tienen una entrada sensorial adicional y distinta, aunque del mismo tipo, conectada a cada una de sus neuronas. En este caso el parámetro a_i^m definido anteriormente debe ser distinto de cero. Representa la capa más general que estamos estudiando en estos apuntes.

3. Capa oculta.

   Es una capa cuyas entradas son las salidas de otras capas, pero a diferencia de las capas asociativas no tienen ninguna entrada sensorial adicional. En este caso el parámetro $a_i^m = 0$, por lo que el estado de las neuronas de cada capa se escribirá en la forma matricial $x^m = W^m y^{m'}$.

4. Capa de salida o capa motora.

   Es la capa cuya salida se conecta a los actuadores del robot. Tiene, por lo tanto, tantas salidas como actuadores, y como consecuencia tantas neuronas como motores. Si el robot tiene dos motores, entonces $n_m = 2$, siendo m la numeración de la capa motora. Lo normal es que no tenga vinculada ningún sensor como en las capas ocultas, es decir que $a_i^m = 0$.

2.4. Arquitectura del controlador neuronal.

La arquitectura del controlador neuronal que estudiaremos consiste en un conjunto de capas neuronales **interconectadas**. Debe disponer, como mínimo, de una capa sensorial y de una capa motora.

En la Figura 7.1 puede verse una arquitectura de 5 capas en la que se indica el número de salidas de cada capa (n_m) y el tipo de sensor de entrada indicado con una numeración distinta s_i.

Hay una conectividad completa entre capas conectadas, es decir, cada neurona de una capa recibe como entradas las salidas de todas las neuronas de la capa o capas presinápticas. Por ejemplo cada neurona de la capa 4 recibe 9 entradas, 8 de la capa 1 y 1 de la capa 3.

Las capas 0 y 2 son capas sensoriales. La capa 2 es la capa conectada a sensores de luz ($s_2 = 3$), y la capa 0 conectada a sensores de proximidad ($s_0 = 1$). Ambas tienen 8 neuronas por lo que habrá en el robot 8 sensores distintos del mismo tipo.

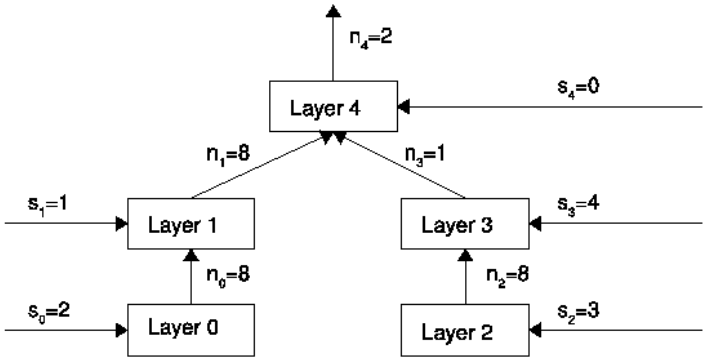


Figura 2.1: Arquitectura neuronal.

Las capas 1 y 3 son capas asociativas. La capa 3 está conectada a sensores de batería ($s_3 = 4$), y la capa 1 a sensores de contacto ($s_1 = 1$). Ambas tienen 8 neuronas. Dado que la capa 1 está conectada a la capa 0, cuyas neuronas están conectadas a sensores de proximidad, podemos entender que la capa 1 esté asociada a un conjunto de sensores de proximidad a través de la capa 0. De alguna forma, la activación de las neuronas de la capa 1 dependerá de ambos tipos de sensores, directa e indirectamente. Es por esto que la llamamos asociativa. Cualquier capa interna que esté conectado directamente a algún sensor, es decir cualquier capa asociativa, estará a su vez conectada indirectamente a una colección de sensores del mismo o de diferente tipo. Su salida se verá afectada por el estado de todos los sensores.

La capa 4 es la capa motora. En este ejemplo la capa 4 tiene dos salidas que serán las señales de actuación (velocidades) de cada una de las dos ruedas del robot. La capa 4 se conecta indirectamente a todos los sensores del robot de tal manera que su salida dependa del estado de todos los sensores. Si esto no fuese así se plantearía una duplicación o multiplicación de redes neuronales que no tienen ningún efecto en el comportamiento del robot: un absurdo funcional. No obstante debe entenderse que aunque haya una conexión indirecta entre actuadores y sensores puede haber sensores que no afecten a la salida o afecten de manera insignificante. Esto puede ocurrir sistemáticamente, lo que sería un síntoma de un diseño erróneo, o puede ocurrir para algunas situaciones en las que se encuentre el robot y no para otras. Este último caso es precisamente el deseable u óptimo, pudiendo entonces afirmar que el robot es una máquina que tiene un **comportamiento situacional**: el comportamiento dependerá del estado de sus sensores, que a su vez toman valores dependientes de su interrelación con el entorno lo-

cal del robot. Sin embargo una noción más interesante de comportamiento situacional es cuando el robot puede seleccionar su propia historia, es decir a qué atender y qué ignorar aunque este tema no va ser estudiado en estos apuntes.

Por último, vemos que en esta figura no hay ninguna capa oculta.

En estos apuntes no vamos abordar la difícil cuestión de diseñar arquitecturas para una aplicación específica. Desgraciadamente no se conoce ninguna metodología para hacerlo. No obstante sugerimos como idea general no hacerla demasiado compleja pensando en los objetivos de la aplicación. La arquitectura de la figura descompone el problema de asociar luz con el estado de la batería y contacto con proximidad a los obstáculos, en dos ramas de un árbol de conexión. La capa motora se ve afectada por todas las entradas sensoriales, pero la rama contacto/proximidad tiene cierta independencia con respecto a la rama luz/batería. En cierta forma se han considerado dos procesos asociados a dos posibles comportamientos distintos: **evitar obstáculos** y **recargar baterías**. La capa motora coordina, en cierta forma, estos dos comportamientos: sus salidas son una combinación de estos dos procesos. También subyace un comportamiento adicional, **navegar**, que podría darse si se tiene en cuenta en el diseño, en ausencia de actividad de los sensores. Puede comprobarse fácilmente que si el valor de todos los sensores de la arquitectura en un determinado instante de tiempo es nula, las salidas de sus capas corespondientes será una constante: la salida de las capas sensoriales 0 y 2 serán nulas, y la de las capas asociativas 1 y 3 dependerán de la función de activación que se haya escogido, y de sus sesgos o umbrales. Por lo tanto la entrada a la capa motora será una constante independiente de los pesos de las neuronas de las capas precedentes. Por lo tanto será en el diseño de la capa motora dónde se podrá implementar un comportamiento de navegación.

El comportamiento de evitar obstáculos exige que se establezca una asociación entre los sensores de contacto y de proximidad, de tal manera que no se activen los sensores de contacto en las proximidades de un obstáculo. Esta asociación puede lograrse de diversas maneras. En el capítulo 3 se estudia la sintonización de los parámetros de la arquitectura (pesos y sesgos de las neuronas) utilizando algoritmos genéticos.

El comportamiento de recargar baterías sigue una idea similar a la de evitar obstáculos. En este caso se supone que cuando la batería está por debajo de cierto nivel, sus sensores se activan. Se supone también que el cargador de batería se encuentra junto a fuentes de luz, por lo que es de esperar que el robot no tome en consideración la intensidad de la luz cuando la batería esté cargada.

Como vemos, el diseño de la arquitectura no es sencillo, pero puede seguirse una lógica dependiente de la clase de comportamientos deseados en relación a un entorno. Un entorno que es desconocido en cuanto a la ubicación espacial de los objetos, pero conocido en cuanto a su influencia

cualitativa sobre el robot. Sería más complicado diseñar una arquitectura si, por ejemplo, las fuentes de luz significasen para el robot dos cosas distintas: ser el lugar donde está el cargador de baterías, y ser una zona a la que no se debe acceder. Sería intersante considerar el caso en que ante situaciones indistinguibles el robot puediese tener comportamientos diferentes. Por ejemplo un comportamiento de acercamiento a la luz cuando está lejos de ella y de huída cuando está cerca si detecta, con otro sensor, una situación peligrosa, o de acercamiento cuando está cerca en caso contrario.

Puesto que el robot puede comportarse de manera idéntica ante obstáculos diferentes, podemos decir que hay una cierta sinécdoque o incluso homonimia, es decir que el robot toma la parte por el todo, o toma un conjunto de cosas distintas como siendo la misma cosa. No obstante puede entenderse este problema desde una perspectiva distinta a la de los tropismos lingüísticos, como suele ser además lo habitual. También puede entenderse que el robot realiza una categorización del mundo, separa en categorías o mejor dicho se comporta de manera distinta ante categorías distintas de objetos. No obstante no vamos a entrar en estos apuntes a estudiar estas capacidades. El hecho de que siempre haya una vinculación sensorio-motriz evita algunos problemas que aparecen en los diseños basados en el conocimiento, concretamente el llamado **problema del anclaje del símbolo**. En las aquitecturas neuronales no hay símbolos asociados a cosas del mundo sino procesos dependientes de la estructura sensorial, y por lo tanto siempre están anclados en el mundo, pero como se acaba de decir no resulta sencillo categorizar este mundo que puede llegar a ser muy ambiguo.

Capítulo 3

Robótica Evolutiva

3.1. Introducción

La robótica evolutiva es una técnica para la creación automática de robots autónomos. Se encuentra inspirada en principios darwinianos de reproducción selectiva de los mejores individuos. Es un enfoque que trata a los robots como organismos artificiales que desarrollan sus habilidades mediante una iteración con el entorno sin supervisión externa.

La idea básica de la robótica evolutiva se encuentra en los fundamentos de los algoritmos genéticos. Una población inicial de individuos, representados por unos cromosomas artificiales que codifican el sistema de control de un robot se generan aleatoriamente y se les permite que interactúen en el entorno. Cada individuo se le deja actuar y es evaluado con respecto a una función de coste (*fitness*). A los mejores individuos (de acuerdo con esa función de fitness) se les permite reproducirse (sexualmente o asexualmente) creando copias de si mismos añadiendo cambios en función de ciertos operadores genéticos (reproducción, mutación,...). Este proceso se repite durante un número de generaciones hasta que un individuo satisface la tarea(s) (fitness) impuesta(s) por el diseñador.

3.2. Algoritmos Genéticos

Los algoritmos genéticos son una herramienta fundamental en la computación evolutiva. Los algoritmos genéticos son algoritmos de búsqueda inspirados en propiedades de la selección natural y genética natural. Un algoritmo genético opera en una población de cromosomas artificiales reproduciendo, de una manera selectiva, los cromosomas que presentan mejores habilidades (en relación a ciertas tareas impuestas por el diseñador) y aplicando cambios aleatorios. Este procedimiento se repite durante varios ciclos (generaciones). Un cromosoma artificial (genotipo) es una cadena de caracteres que codifica las características de un individuo (fenotipo). En la práctica, el cromosoma

puede codificar diferentes parámetros a optimizar en el problema. En nuestra caso, nos centraremos en la codificación de los pesos, sesgos y parámetros libres de una red neuronal dentro del cromosoma. Por lo tanto, serán estos parámetros los que sintonizará el algoritmo genético en relación a una función de coste (*fitness*) definida por el diseñador.

La función de fitness es un criterio de optimización que evalúa el rendimiento de cada individuo. Cuanto mayor es el valor de la fitness mejor es el individuo.

3.2.1. Operadores genéticos

Los operadores genéticos son los encargados de la reproducción de una generación a otra. Existen una buena cantidad de operadores, desde los más básicos y utilizados a los más complejos y específicos. En estos apuntes nos centramos en los operadores más utilizados en la literatura, dejando al lector la posibilidad de profundizar en el resto en un futuro. Los 3 operadores más utilizados son: Reproducción (asexual), cruce (reproducción sexual) y mutación. A continuación detallamos cada uno de ellos.

Reproducción

La reproducción es un proceso en el que los individuos son copiados en función de sus valores de fitness. Esta reproducción quiere decir que individuos con un valor mayor de fitness tienen mayores probabilidades de estar presentes en la siguiente generación.

Individuo	Cromosoma	Fitness	% fitness
A	0011	126	50
B	1010	42	17
C	1101	84	33
Total		252	100

Tabla 3.1: Ejemplo de población con sus valores de fitness asociados.

Este operador puede ser implementado con varias técnicas. La más utilizada es la ruleta sesgada, donde cada cromosoma en la población tiene un ranura de tamaño proporcional a su fitness. Supongamos una población de 3 individuos (A, B y C) cuyas funciones de fitness son las que se muestran en la Tabla 3.1. Si sumamos los valores de las fitness obtenemos un total de 252. El porcentaje de fitness total al que corresponde cada uno se muestra también en la tabla y puede ser representado mediante la ruleta de la Figura 3.1. Observamos que el individuo A tiene una fitness de 126 que corresponde al 50 % de la fitness total, el individuo B 42 correspondiente al 17 % y el individuo C 84 correspondiendo con el otro 33 %.

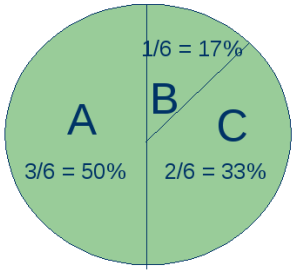


Figura 3.1: Ejemplo de ruleta sesgada para el ejemplo de la Tabla 3.1.

Una vez creada la ruleta podemos realizar la reproducción simplemente girando la misma tantas veces como individuos necesitemos. Podemos observar que la nueva generación será, en probabilidad, una representación de la distribución de fitness de la generación anterior.

Cruce

Al contrario que la reproducción (asexual), el cruce, crossover o reproducción sexual es un operador que actúa en dos fases: una fase de emparejamiento y una fase de cruce.

- **Fase de emparejamiento**: En esta fase los individuos de la nueva generación son emparejados aleatoriamente.
- **Fase de cruce**: Para cada pareja se selecciona un punto de cruce aleatorio (k).

Por ejemplo, consideremos los individuos A y C de la población del ejemplo de la Tabla 3.1.

```
A = 0 | 0 1 1
C = 1 | 1 0 1
```

Supongamos que al elegir un número aleatorio entre 1 y 4 el resultado ha sido $k = 1$ como se indica por el separador "|". El resultado del cruce crea dos nuevos individuos:

```
A' = 0 1 0 1
C' = 1 0 1 1
```

Mutación

Al contrario que en los otros dos operadores descritos anteriormente, la mutación es un operador que no busca entre los mejores (reproducción)

o pretende obtener un individuo con las mejores capacidades de los más aptos (crossover), sino que pretende encontrar nuevas soluciones no incluidas en las generaciones anteriores. Eventualmente la mutación también puede hacer que individuos con altas capacidades pierdan su potencial debido a mutaciones excesivas.

En los algoritmos genéticos, el operador de mutación supone una alteración aleatoria (con baja probabilidad) de un gen de un cromosoma. En una codificación binaria, una mutación se corresponderá con la modificación de un 0 a 1 o viceversa. El operador de mutación juega un papel secundario en los algoritmos genéticos sencillos. La probabilidad de mutación suele ser suficientemente baja para que no exista una pérdida de información con el paso de las generaciones.

Élite

En los algoritmos genéticos utilizados para la robótica evolutiva es común el uso de los individuos llamados élite. Estos individuos, los mejores de su generación, no sufren alteraciones de cruzamiento ni mutación, y se reproducen directamente a la nueva generación. Gracias a la élite, la nueva generación mantiene a los mejores individuos de la generación anterior sin perder a los más cualificados. Esto permite también que la probabilidad de mutación pueda aumentarse, evitando estancarse en mínimos locales a cambio de un mayor tiempo de convergencia.

3.2.2. ¿Cómo funcionan?

Para comprobar el funcionamiento de los algoritmos genéticos vamos a plantear un problema de maximizar una función. Supongamos la función $f(x) = x^2$ donde $x \in [0, 31]$. Para la codificación de los individuos vamos a trabajar con cromosomas de 5 bits binarios, desde el número 0 (00000) hasta el número 31 (11111). Para comenzar seleccionamos una población inicial aleatoria de 4 individuos (ver Tabla 3.2). La tabla muestra los 4 individuos de la población, junto con el valor (entero) que representan y el valor de fitness de la función a maximizar. La probabilidad de que cada individuo sea seleccionado para una futura generación se muestra en la última columna. Téngase en cuenta que la probabilidad de ser seleccionado se calcula como la fitness de cada individuo dividida por la fitness total ($\frac{f_i}{\sum fi}$). La Figura 3.2 representa gráficamente la ruleta sesgada correspondiente a esta población.

Una vez evaluada la primera generación aplicamos los operadores genéticos para crear la siguiente. Comenzamos con la reproducción. Para ello giramos la ruleta creada 4 veces para obtener 4 nuevos individuos. En nuestro ejemplo esto ha resultado en 1 copia de los individuos 1 y 4 y 2 copias del individuo 2 (ver columna 2 (cromosoma) en Tabla 3.3). De acuerdo con la probabilidad de ser seleccionados, observamos que hemos obtenido lo espe-

Individuo	Cromosoma	Valor (x)	$f(x)$	Probabilidad($\frac{f_i}{\sum f}$)
1	01101	13	169	0.14
2	11000	24	576	0.49
3	01000	8	64	0.06
4	10011	19	361	0.31
Total			1170.00	1.00
Media			292.50	0.25
Max			576.00	0.49

Tabla 3.2: Ejemplo de algoritmo genético. Inicialización aleatoria de la población

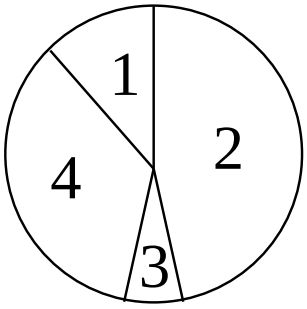


Figura 3.2: Ejemplo de algoritmo genético. Representación gráfica de la fitness de la población inicial.

rado: el mejor (2) obtiene más copias, los promedios (1 y 4) se mantienen y los peores (3) desaparecen.

Una vez finalizada la reproducción comenzamos con el cruce. Necesitamos (i) emparejar los individuos aleatoriamente y (ii) seleccionar un punto de cruce entre ellos. Si nos fijamos en la Tabla 3.3, observamos que el primer y segundo individuo han sido emparejados con un punto de cruce en el bit 4, mientras que el tercer y cuarto individuo han sido emparejados con un punto de cruce en el bit 2. La tabla muestra el resultado de la nueva población.

Individuo	Cromosoma	Parejas	Cruce	Nueva Población
1	0110|1	2	4	01100
2	1100|0	1	4	11001
3	11|000	4	2	11011
4	10|011	3	2	10000

Tabla 3.3: Ejemplo de algoritmo genético. Reproducción y cruce.

Finalmente necesitamos aplicar la mutación. Supongamos que tenemos una probabilidad de mutación de 0.05. Puesto que tenemos 5 bits x 4 individuos, deberíamos esperar que 20 x 0.05 = 1 bit recibiese mutación. La Tabla 3.4 muestra que nuestro experimento se comporta como es esperado, en el que el bit 3 del individuo 2 ha sido modificado de 0 a 1. Puesto que la nueva población está construida, ya podemos evaluarla. Al igual que en la población inicial, decodificamos el cromosoma con respecto a su valor entero, calculamos la fitness y su probabilidad de selección.

Individuo	Cromosoma	Valor (x)	$f(x)$	Probabilidad($\frac{f_i}{\sum f}$)
1	01100	12	144	0.07
2	11**1**01	29	841	0.43
3	11011	27	729	0.37
4	10000	16	256	0.13
Total			1970.00	1.00
Media			492.50	0.25
Max			841.00	0.43

Tabla 3.4: Ejemplo de algoritmo genético. Mutación y evaluación de la segunda generación.

Continuando a la siguiente generación repetimos el mismo proceso (ver Tabla 3.5. El resultado de la reproducción en este caso a obtenido 2 copias del individuo 2 y 2 copias del individuo 3. El cruce a impuesto un primer emparejamiento del individuo 2 con el individuo 3 en el bit 1 y un segundo emparejamiento de los otros individuos con un punto de cruce bit 3. La mutación ha modificado el bit 1 del individuo 3 de 1 a 0.

Individuo	Reproducción	Cruce	Mutación	Valor (x)	$f(x)$	$\frac{f_i}{\sum f}$
1	1|1101	11101	11101	29	841	0.32
2	1|1011	11011	11011	27	729	0.28
3	110|11	11001	**0**1001	9	81	0.03
4	111|01	11111	11111	31	961	0.37
Total					2612	1.00
Media					653	0.25
Max					961	0.37

Tabla 3.5: Ejemplo de algoritmo genético. Evaluación de la tercera generación.

Se puede observar que el proceso de búsqueda puede continuar indefinidamente, aunque ya hemos encontrado la mejor solución al problema (11111). Si continuase la evolución, la población tendería a tener más repre-

sentaciones del individuo 11111.

Si intentamos sacar algunas conclusiones de este pequeño ejemplo podemos observar que:

- Según vamos avanzando en la generación la media ha aumentado desde 292.5 hasta 653.
- La fitness máxima a pasado de 576 a 961.

Aunque el proceso aleatorio ayuda a crear esta situación favorable, se puede ver que la mejora no es casual. En la primera generación el individuo 11000 se reproduce 2 veces debido a su valor alto de fitness comparado con la media. Sin embargo únicamente la reproducción no ayuda a buscar nuevos puntos en el espacio. Cuando este individuo se ha cruzado con el 10011, obteniendo el nuevo individuo 11011, el proceso ha dado como resultado un mejor individuo de acuerdo con la fitness. Esto ha permitido que la combinación de 2 sub-cromosomas 11XXX y XXX11 mejorase al individuo final. Aunque en un principio esto parece un heurístico, podemos intuir un efecto interesante de búsqueda robusta en el algoritmo genético.

Por otro lado, la mutación ha permitido obtener un mejor individuo (11101) en la generación 2, mientras que ha permitido desechar al nuevo individuo (01001) en la generación 3. Por lo tanto, tal y como comentábamos en la sección anterior, la mutación es capaz de eliminar individuos que han perdido potencial a la hora de realizar la reproducción sexual a la vez que permite que nuevos individuos más capaces aparezcan en la generación.

Existen fundamentos matemáticos que prueban estas reflexiones anteriores basados en el concepto de sub-cromosomas (*schemas*), y cómo se propagan a lo largo de las generaciones. Sin embargo, consideramos estos análisis analíticos materia de un curso superior a *Introducción a la Robótica Inteligente* y por lo tanto no se presentan en estos apuntes.

3.3. Evolucionando robots

La evolución artificial de robots puede implicar un tiempo considerable debido a la evaluación de varios individuos en una población durante un número elevado de generaciones. Además si la evolución se realiza online, sobre un robot físico, el tiempo de de evaluación puede llegar a ser excesivo. Esto hace que normalmente se creen simuladores, lo más acordes posible con la realidad (morfología del robot, dimensiones, tamaño de la arena, situación de los objetos, muestreo de sensores, etc.), para llevar a cabo una evolución en simulación y posteriormente cargar el controlador en el robot real.

La robótica evolutiva, como se ha comentado anteriormente, se centra en hacer uso de algoritmos genéticos que codifican los parámetros libres de una red neuronal (pesos, sesgos, ganancias,etc.) y evolucionan dicho cromosoma

hasta maximizar una fitness suministrada por el usuario. En esta sección presentamos unos ejemplos de robótica evolutiva para su comprensión.

Téngase en cuenta que cuando trabajamos con robots la fitness no representa una simple función, sino que normalmente se evalúan las trayectorias y acciones que realiza el robot en un periodo de tiempo dado. De esta forma no se van buscan los máximos de una función, como en el ejemplo propuesto en la Sección 3.2, sino que se pretenden localizar máximos de funcionales [1]. De esta manera, se intenta obtener la curva que más se aproxime a la óptima de la funcional.

En los ejemplos que presentamos a continuación se clarifican estos conceptos y se observan diferentes posibilidades de evaluación de individuos dependiendo de las tareas y acciones que se quiera que realicen. Téngase en cuenta que durante los ejemplos denominamos "fitness" indistintamente a la fitness de una función o de una funcional. Observaremos en los ejemplos que el robot es evaluado de acuerdo a una fitness instantánea de la función que normalmente representaremos por ϕ. Esta función representa el valor de fitness en un instante de tiempo. Sin embargo, al finalizar el experimento evaluamos al robot con respecto a una funcional que dependerá de esa fitness instantánea. Esa fitness, representada por F, será el valor con el que trabajará el algoritmo genético.

3.3.1. Navegación

Para conseguir una navegación efectiva de un robot es necesario disponer de una relación entre las entradas sensoriales y las acciones motoras. Las acciones realizadas por el robot dependen de la información sensorial, que igualmente depende de las acciones realizadas por el robot en el instante anterior. Esta realimentación hace que la definición de una navegación efectiva sea compleja. Desde el punto de vista ingenieril, uno puede pretender atender todas las situaciones sensoriales posibles y asociarlas con unas acciones de los motores. En el caso de robots inteligentes y autónomos que operan en entornos no estructurados, impredecibles y desconocidos esta solución no es siempre viable debido al numero de situaciones posibles. Incluso en los sencillos vehículos I-V de braitenberg (ver Capítulo 1) el diseño de las conexiones excitadoras o supresoras requiere un análisis de los motores y los sensores.

Mediante la robótica evolutiva se deja a discreción de la evolución el diseño de dichas conexiones con respecto a la relación entre el robot y el entorno en el que opera y a la función de fitness definida. En este ejemplo presentamos un robot con 8 entradas sensoriales y 2 motores en configuración diferencial (ver Figura 3.3a). La arquitectura del robot es una red neuronal (perceptrón). Consiste en una capa sensorial de entrada donde se encuentran

[1] Téngase en cuenta que una funcional es una función que depende de una función.

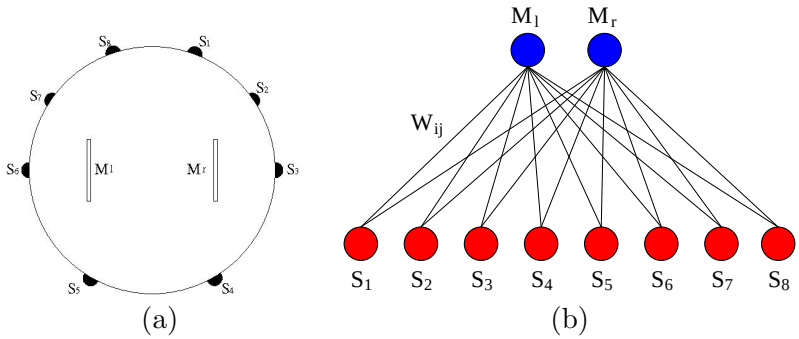


Figura 3.3: Arquitectura neuronal del experimento de navegación.

los 8 sensores de proximidad y una capa de salida donde se conectan los 2 motores (ver Figura 3.3b). Cada neurona motora implementa una función sigmoide cuya salida en el intervalo [0,1] representa el movimiento del motor. 0 velocidad máxima hacia atrás, 0.5 parado y 1 velocidad máxima hacia adelante. Todas las neuronas de la capa sensorial se encuentran conectadas a todas las neuronas de la capa motora.

La función de fitness utilizada para evaluar la navegación del robot se basa únicamente en variables accesibles por el robot mismo. De esta manera, esta evolución se puede llevar a cabo desde el mismo robot, sin necesidad de ningún control externo. La velocidad de cada rueda del robot es medible gracias a unos encoders, que nos ofrecen un valor del intervalo [0,1] tal y como se ha definido anteriormente. La función implementada es la siguiente:

$$\Phi = V(1 - \sqrt{\Delta v}) * (M_r * M_l) * (1 - i) \tag{3.1}$$

donde V es la suma de los valores absolutos de cada una de las velocidades de cada motor menos 0.5, Δv es el valor absoluto de la diferencia entre las velocidades, M_r y M_l son las velocidades del motor derecho e izquierdo respectivamente e i es la normalización entre 0 y 1 del máximo valor de los sensores infrarrojos. Por lo tanto, tenemos que:

$$V = |M_r - 0{,}5| + |M_l - 0{,}5| \tag{3.2}$$

$$\Delta v = |M_r - M_l| \tag{3.3}$$

$$i = max(S_i) \tag{3.4}$$

donde S_i es el valor de cada uno de los sensores de proximidad.

Si observamos la función de fitness en detalle podemos ver que:

- La componente V se maximiza cuando la rotación de ambas ruedas, independientemente de su sentido, es máxima. Por lo tanto se pretende favorecer el hecho de que el robot se encuentre en movimiento.

- La componente $1 - \sqrt{\Delta v}$ favorece el que ambas ruedas giren en el mismo sentido. Cuanto mayor es la diferencia entre el sentido y valor de giro de las ruedas, más se aproximará Δv a 1. La raíz cuadrada es utilizada para aumentar el peso de pequeñas diferencias. Puesto que el valor es restado de 1, esta componente se maximiza por robot cuyo movimiento de las ruedas sea en el mismo sentido, sin tener en cuenta la velocidad y el sentido de giro. De tal manera que un robot parado y un robot a máxima velocidad hacia adelante generarán el mismo valor. Si combinamos esta componente y la anterior obtenemos un función de fitness máxima para un robot que se mueva en línea recta a la máxima velocidad (hacia adelante o hacia atrás).

- La tercera componente $(M_r * M_l)$ maximiza la función de fitness para un robot moviéndose hacia adelante ($M_r = 1$ y $M_l = 1$).

- Finalmente la última componente $(1 - i)$ es la que se encarga de la evitación de obstáculos. Cada uno de los sensores de proximidad (S_i) emite un rayo infrarrojo que se refleja en obstáculos a una distancia menor de 28 cm. Cuanto más cerca está el obstáculo mayor es el valor de lectura que pertenece al intervalo [0,1]. De tal manera, el valor i representa como de cerca se encuentra un obstáculo independientemente de su posición. Puesto que este valor es restado a 1, esta componente maximiza la fitness cuando el robot se encuentra alejado de los objetos.

Para comenzar a evolucionar nuestro robot, primero tenemos que seleccionar ciertos parámetros y operadores genéticos. Es necesario definir el número de individuos por generación (100) , así como el número de generaciones máximas a ejecutar (100). Igualmente, necesitamos definir si se van a utilizar los diferentes operadores genéticos, que en el ejemplo propuesto son los 3 explicados en secciones anteriores: reproducción, cruce y mutación con una probabilidad del 0.05. Igualmente, es necesario definir cual va a ser el tiempo de evaluación del individuo (100 s) y cómo va a evaluarse la fitness. En nuestro ejemplo, el individuo es evaluado acorde con la fitness en todo el tiempo de ejecución (100 s). Durante todo este tiempo calculamos $fit_i = \sum_{t=0}^{100} \Phi$, y una vez finalizado el experimento calculamos la fitness del individuo como $F_i = fit_i/100$.

Una vez definido el problema y los operadores nos ponemos a evolucionar nuestro robot navegador en una arena cerrada libre de obstáculos. Para ello se ha hecho uso del simulador Irsim (ver Parte II) y se ha codificado el controlador tal y como se comenta en la Sección 8.4.1.

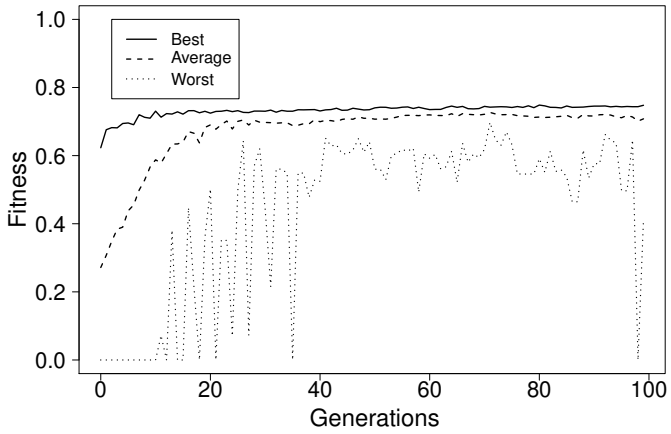


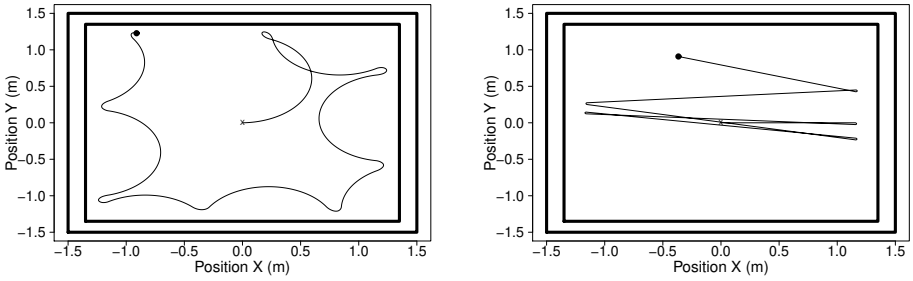
Figura 3.4: Experimento navegación. Media de las fitness de la población y mejor y peor individuo de la población para cada una de las generaciones.

Para cada generación se grabaron los datos de la media de las fitness de la población así como del peor y mejor individuo (ver Figura 3.4). Aunque la fitness ha ido aumentando paulatinamente hasta la generación 100, observamos que sobre la generación 20 se ha llegado a un estado permanente en el que el mejor individuo muestra una navegación muy eficiente. Téngase en cuenta que una fitness de 1.0 sólo podría haberse conseguido en el caso de un robot moviéndose a máxima velocidad en un arena sin obstáculos ni bordes. Bajo la arena utilizada, una arena de 3x3 m^2. con bordes en los laterales observamos que el mejor individuo ha sido capaz de obtener una fitness de 0.75.

La Figura 3.5 muestra la trayectoria de a) el mejor individuo de las primera generación y b) el mejor individuo de las 100 generaciones. El robot ha comenzado en la posición marcada por una "X" y ha finalizado su movimiento al cabo de los 100 segundos en la posición del círculo. Podemos observar que el individuo de las primera generaciones realiza movimientos curvilíneos, mientras que el individuo de las generación 100 es capaz de realizar movimientos rectilíneos tal y como se lo hemos exigido en la fitness.

Una vez comprobado el funcionamiento de nuestro robot, debemos preguntarnos si funcionará correctamente en otras situaciones o entornos. En nuestro caso hemos modificado el entorno para incluir una serie de obstáculos en el mismo y observar el comportamiento del robot. La Figura 3.6 muestra las trayectorias de los robots.

Finalmente queremos mostrar el estado de los sensores y motores cuando el robot se encuentra navegando. Para el estudio se ha hecho uso del mejor



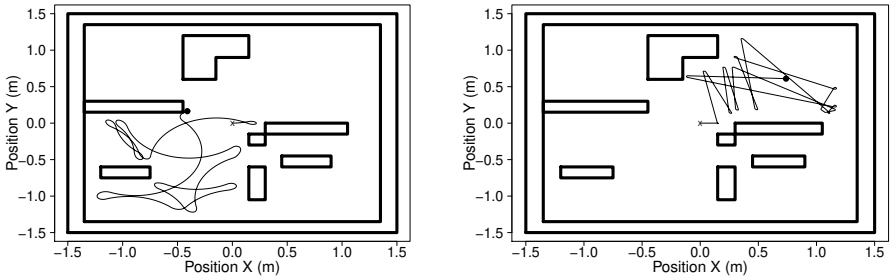
(a) (b)

Figura 3.5: Experimento navegación. Trayectorias realizadas por los mejores individuos de la (a) generación 0 y (b) generación 100, para una duración de 100 segundos de experimento en un entorno sin obstáculos.

individuo de la generación 100 en la arena sin obstáculos. En la Figura 3.7 mostramos el estado de los 8 sensores de infrarrojos y los 2 motores para el intervalo [8,11] segundos en el cual el robot se encuentra con un obstáculo. Podemos ver que cuando el robot se encuentra lejos del obstáculo (a una distancia mayor de 28 cm.) todos los sensores está inactivos ($S_i = 0\,, \forall i$). En ese estado la salida de los motores es prácticamente 1 (0.993) y el robot se mueve en línea recta. En cuanto el robot se acerca al obstáculo los sensores lo detectan y comienzan a aumentar su valor. En ese momento el robot mantiene el motor izquierdo a máxima velocidad (0.993) mientras comienza a reducir la velocidad del motor derecho. Observamos que la velocidad del motor derecho es menor cuanto mayor es el valor de los sensores S_1 y S_8, los dos sensores delanteros. Conforme el valor de dichos sensores se ve reducido la velocidad del motor derecho comienza a aumentar de nuevo para finalmente, una vez que no se detecta el obstáculo, volver al estado inicial de máxima velocidad en ambos motores.

3.3.2. Navegación y carga

Ahora vamos a plantear un experimento en el que el robot debe navegar por el entorno a la par que mantiene la batería lo más cargada posible. El robot dispone de una célula fotoeléctrica que empieza a cargar la batería del mismo cuando se encuentra a menos de 50 cm. de una fuente de luz. Para llevar a cabo el experimento hacemos uso de una red neuronal de 4 capas, 3 capas sensoriales conectadas con la capa motora (ver Figura 3.8). Las capas sensoriales incorporan 8 sensores de infrarrojos (I_{Si}), 8 sensores de luz (L_{Si}) y un sensor de batería (B_{Si}). La capa motora está conectada a los 2 motores (M_l y M_r) del robot.



(a) (b)

Figura 3.6: Experimento navegación. Trayectorias realizadas por los mejores individuos de la (a) generación 0 y (b) generación 100, para una duración de 100 segundos de experimento en un entorno con obstáculos.

Posicionamos el robot en una arena de 3x3 m^2 sin obstáculos con una fuente de luz en la parte inferior izquierda. Cada generación consta de 100 individuos y la evaluación durará un máximo de 100 generaciones. Evaluaremos cada individuo durante 300 segundos mediante la siguiente función de fitness:

$$\Phi = \alpha * V(1 - \sqrt{\Delta v}) * (M_r * M_l) * (1 - i) + \beta * B + \gamma * l \tag{3.5}$$

donde B es el valor real del estado de la batería del robot (0 descargada, 1 cargada), l es la normalización entre 0 y 1 del máximo valor de los sensores de luz, α, β y γ son 3 parámetros de ajuste de la ecuación de fitness (0.5, 0.2 y 0.3 respectivamente en nuestro ejemplo), y el resto de parámetros son los utilizados en el experimento anterior. Además se han tenido en cuenta las colisiones del robot con los obstáculos (paredes de los bordes de la arena). La fitness total permite un máximo de 10 colisiones con los obstáculos, a partir de los cuales el robot obtiene una fitness = 0:

$$F_i = \frac{\sum_{t=0}^{300} \Phi}{300} * (1 - \frac{min(col, 10)}{10}) \tag{3.6}$$

donde *col* es el número de colisiones del robot en los 300 segundos de evaluación.

Observamos que esta función de fitness es máxima para movimiento rectilíneos del robot con máxima carga, máximo valor de potencia lumínica y mínimo número de colisiones. Por lo tanto, un robot que intente maximizar dicha función intentará situarse próximo a una fuente de luz que le permita mantener la carga al máximo pero lo suficientemente alejado para permitir su movimiento.

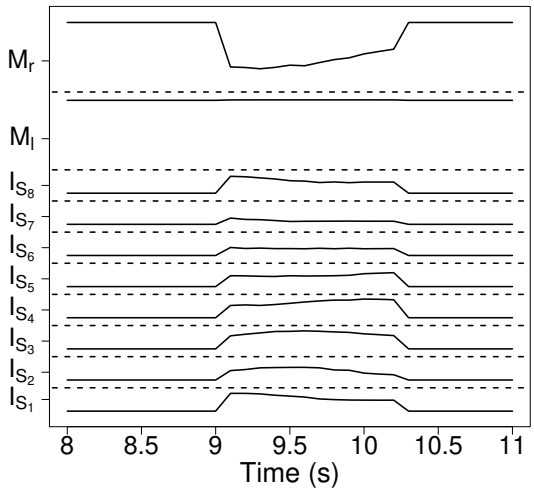


Figura 3.7: Experimento navegación. Estado de los sensores y actuadores durante el periodo de tiempo [8,11] segundos en el que el robot se encuentra con un obstáculo en su trayectoria.

Para cada generación se grabaron los datos de la media de las fitness de la población así como del peor y mejor individuo (ver Figura 3.9). Observamos que el mejor individuo comienza con un valor de fitness próximo a 0.6 y al cabo de 100 generaciones consigue superar los 0.7. El valor de la media crece desde 0.1 en la generación 0 hasta 0.6 en la generación 100. Por último podemos observar que el peor individuo suele tener un valor 0 en la mayoría de las generaciones debido a la restricción impuesta del número de colisiones.

En la Figura 3.10 se muestran las trayectorias del mejor individuo de la generación 0 y generación 100 para una arena sin obstáculos, la misma con la que se ha realizado la evolución, mientras que la Figura 3.11 muestra a los mismos individuos en una arena con obstáculos. Obsérvese que en este último caso el mejor individuo de la generación 0 es incapaz de sortear el obstáculo que se encuentra delante de la luz (circulo rojo), mientras que el individuo de la generación 100 lo sortea y realiza un comportamiento similar al de la arena sin obstáculos.

3.3.3. Recogida de objetos

Este último experimento se centra en una tarea de recogida de objetos. En la arena disponemos de diferentes objetos (representados por baldosas de color gris) a ser recogidos por el robot y que deben ser transportados a

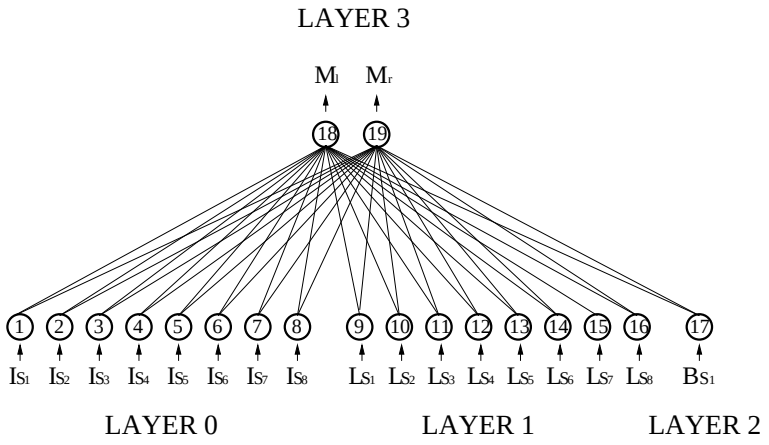


Figura 3.8: Arquitectura neuronal del experimento de navegación y carga.

un centro de recogida (representados por baldosas de color gris). El robot debe navegar por el entorno evitando obstáculos y localizando objetos. Una vez que los ha localizado debe transportarlos (virtualmente) a la zona de recogida. Los robots recogen y dejan objetos de una manera virtual. Se ha creado un sensor virtual a modo de pinza que es activado (1) cuando un robot pasa por una baldosa de color gris y se desactiva (0) cuando llega a una baldosa de color negro, simulando la recolección y depósito de los objetos. La arena, de 3x3 m^2 sin obstáculos, consta de 4 objetos y un centro de recogida. Además se ha situado una fuente de luz en el centro del centro de recogida para guiar al robot hasta el mismo.

La arquitectura neuronal del robot implementada se presenta en Figura 3.12. Está formada por 5 capas neuronales, 2 sensoriales, 2 asociativas y 1 motora. La capas sensoriales tienen como entrada los sensores de proximidad (I_{Si}) y de luz (L_{Si}), mientras que las asociativas los sensores de contacto (C_{Si}) y la pinza (G_{Si}). Observamos que la rama de la izquierda es la encargada de la evitación de obstáculos gracias a los sensores de proximidad y de contacto. Por otro lado, la rama de la derecha es la encargada de la recolección y depósito gracias a los sensores de luz y la pinza.

La evolución se ha llevado a cabo durante 180 generaciones, con 100 individuos por generación, donde cada individuo ha sido evaluado durante 300 segundos. La evaluación de cada individuo depende del estado en el que se encuentre, búsqueda o depósito. Mientras un robot se encuentra buscando objetos en la arena es evaluado en función de sus trayectorias rectilíneas tal y como se realiza en el experimento de navegación. Si consigue localizar un objeto, su función de fitness parcial es incrementada en 10 y cambia su estado a depositar. Una vez que el robot ha encontrado un objeto es necesario que lo deposite en el centro de recogida. En este estado el robot es evaluado en

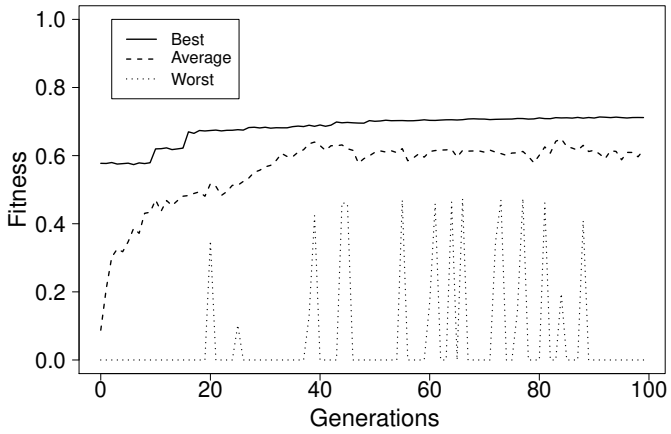


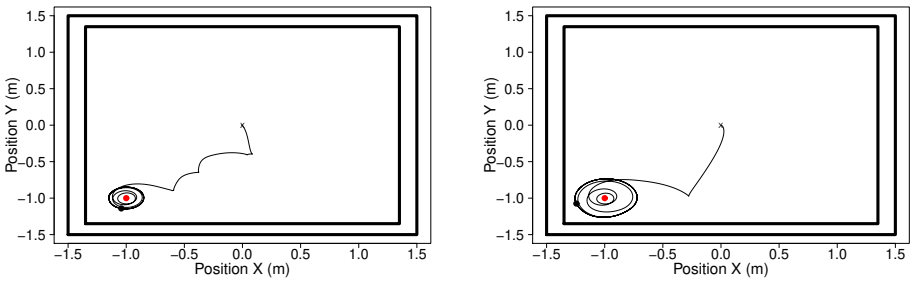
Figura 3.9: Experimento navegación y carga. Media de las fitness de la población y mejor y peor individuo de la población para cada una de las generaciones.

función del valor máximo de los sensores de luz, de tal manera que cuanto más cerca se encuentre de la luz mejor fitness obtendrá. Si consigue llegar al centro de recogida su función de fitness parcial es incrementada en 10 y vuelve a su estado de búsqueda:

$$\Phi = \begin{cases} V(1-\sqrt{\Delta v}) * (M_r * M_l) * (1-i) + G * 10; & si\ S = 0 \\ l + (1-G) * 10; & si\ S = 1 \end{cases} \tag{3.7}$$

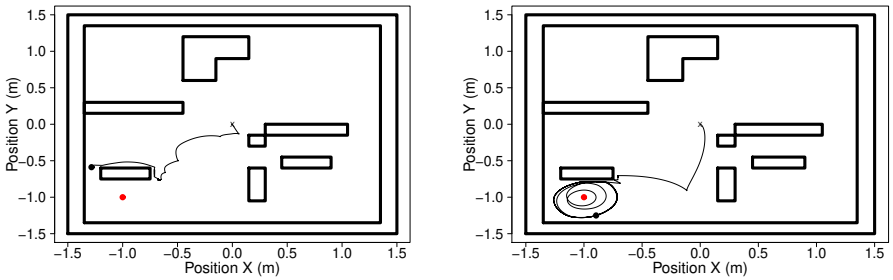
donde G es el estado de la pinza virtual (0 si no hay objeto o 1 si hay objeto) y S es el estado de búsqueda (0) o depósito (1) en el que se encuentra el robot.

La Figura 3.13 muestra la evolución durante las 180 generaciones, mientras que la Figura 3.14 muestra las trayectorias del mejor robot de la generación 0 y la del robot de la generación 100. Podemos observar como el mejor robot de la generación 0 es incapaz de localizar ningún objeto en el tiempo del experimento al encontrarse girando en una espiral en el centro de la arena. Por contra, el mejor individuo de la generación 180 localiza una de las fuentes de comida y recorre la arena rebotando (sin chocar) con las paredes hasta que localiza el centro de recogida y vuelve a realizar trayectorias rectilíneas para localizar de nuevo objetos en la arena.



(a) (b)

Figura 3.10: Experimento navegación y carga. Trayectorias realizadas por los mejores individuos de la (a) generación 0 y (b) generación 100, para una duración de 100 segundos de experimento en un entorno sin obstáculos.



(a) (b)

Figura 3.11: Experimento navegación y carga. Trayectorias realizadas por los mejores individuos de la (a) generación 0 y (b) generación 100, para una duración de 100 segundos de experimento en un entorno con obstáculos.

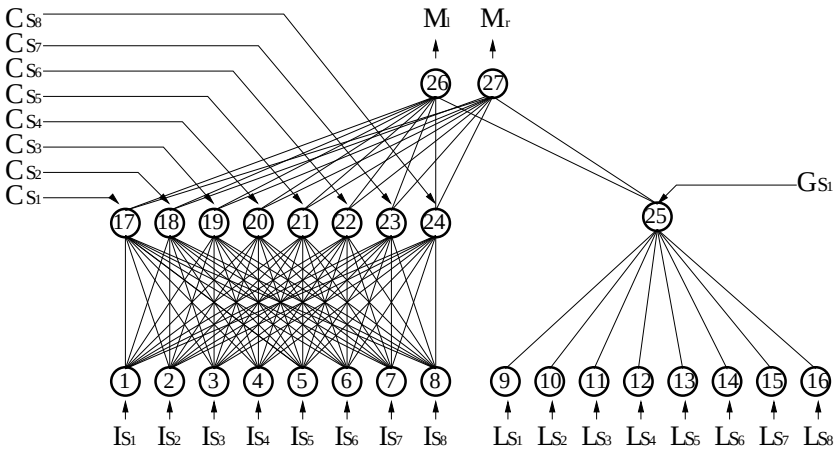


Figura 3.12: Arquitectura neuronal del experimento de recogida de objetos.

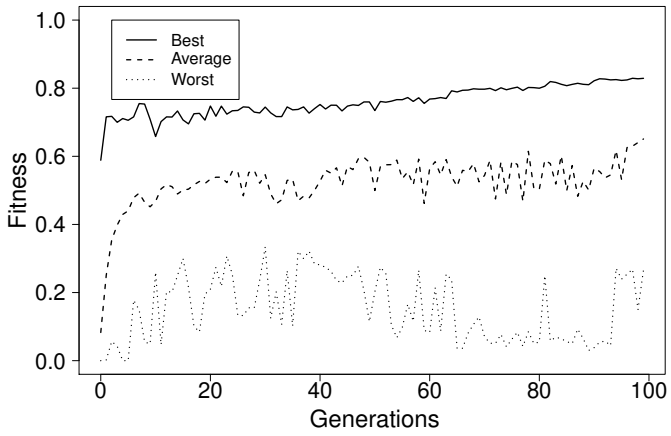
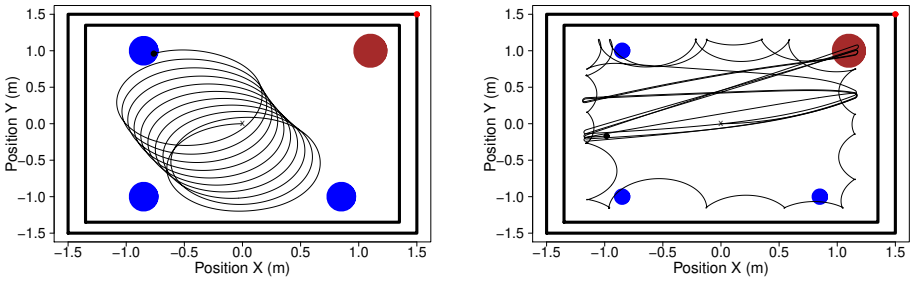


Figura 3.13: Experimento de recogida de objetos. Media de las fitness de la población y mejor y peor individuo de la población para cada una de las generaciones.



(a) (b)

Figura 3.14: Experimento de recogida de objetos. Trayectorias realizadas por los mejores individuos de la (a) generación 0 y (b) generación 100, para una duración de 100 segundos de experimento en un entorno sin obstáculos.

Capítulo 4

Arquitectura Subsunción

4.1. Introducción

La Arquitectura Subsunción (*Subsumption Architecture*) nace gracias a Rodney A. Brooks en 1986 como un enfoque ingenieril al diseño de comportamientos de las máquinas. La intención de Brooks era crear una metodología que facilitara el diseño de algoritmos en los que robots tuvieran múltiples objetivos respondiendo a múltiples sensores de una manera rápida y prácticamente reactiva. Brooks pretendía obtener arquitecturas **robustas** frente a fallos, en las que los robots tuvieran como prioridad su supervivencia, contemplada en las capas bajas del algoritmo. Pero a su vez que fuese una arquitectura **modular y ampliable**, para que los robots fueran capaces de realizar cada vez tareas más complejas. Es esta última característica la que dio popularidad a la Arquitectura Subsunción, ya que se comienza construyendo robots con comportamientos simples y luego se construyen comportamientos más complejos encima de ellos sin necesidad de realizar modificaciones en las capas inferiores. Si añadimos que las ideas de poder reutilizar los comportamientos inferiores reflejan aspectos importantes de la teoría de evolución y que los conceptos de percepción-acción se encuentran presentes como en arquitecturas puramente reactivas (ver Capítulo 1), la Arquitectura Subsunción es una arquitectura ampliamente usada después de 20 años de su creación.

4.2. Arquitecturas basadas en el comportamiento

La Arquitectura Subsunción fue la primera arquitectura dentro de lo actualmente conocido como Arquitecturas Basadas en el Comportamiento. Anteriormente los trabajos estaban centrados en las Arquitecturas Basadas en el Conocimiento que se basan en una arquitectura secuencial de percepción, deliberación y acción (ver Figura 4.1). En estas arquitecturas existe una primera fase de sensado, donde la información sensorial del mundo es

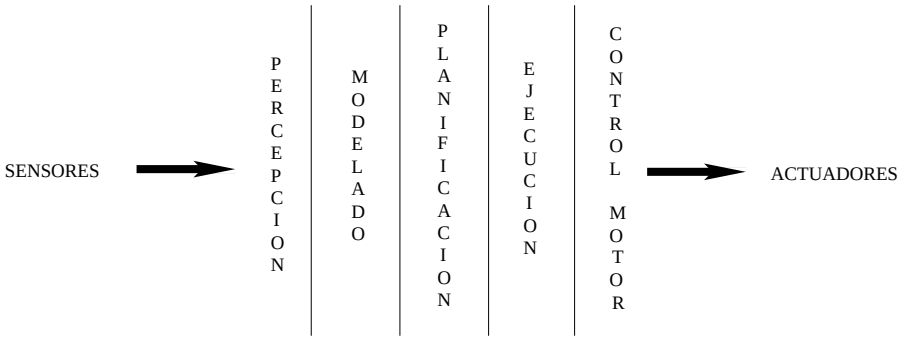


Figura 4.1: Arquitectura Basada en el Conocimiento

integrada dentro de un modelo del mundo o representación interna. Una vez modificado el modelo del mundo, se evalúan las posibilidades de las siguientes acciones y se eligen cuales se van a llevar a cabo. Finalmente se ejecutan unas acciones en base a los planes establecidos. Uno de los mayores problemas de estas arquitecturas, desde el punto de vista funcional, es la velocidad de respuesta a cambios rápidos del entorno. Esta característica ha sido caricaturizada por Fitz Patrick en la Figura 4.2.

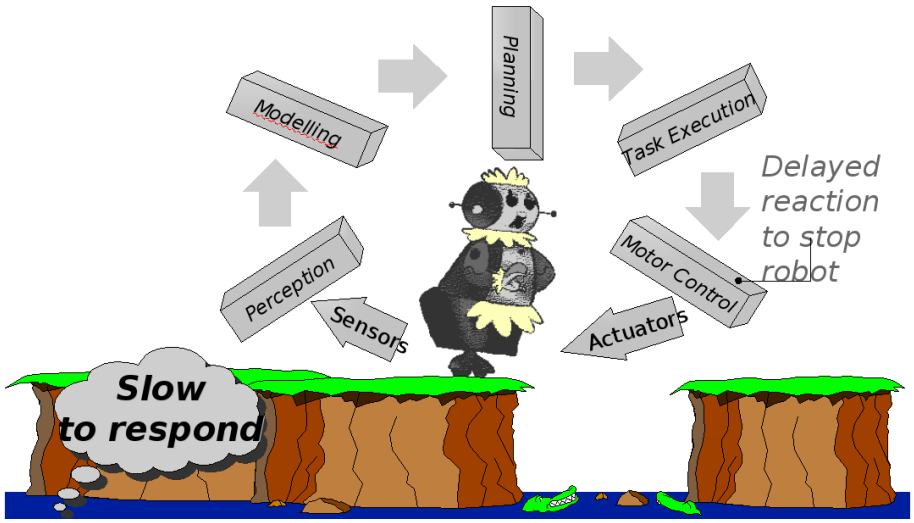


Figura 4.2: Caricatura de una Arquitectura Basada en el Conocimiento (Dibujo obtenido de Fitz Patrick)

En contraste en las Arquitecturas Basadas en el Comportamiento se intenta evitar el modelo del mundo, entendiendo que el mejor modelo del

mundo es el mundo mismo. Por lo tanto, la fase de deliberación se ve reducida o eliminada haciendo que el comportamiento del robot sea más rápido y por lo tanto responda a mejores estímulos y modificaciones del mundo. En consecuencia, la mayoría de estas arquitecturas son prácticamente reactivas. De hecho, no existe una programación en el robot que indique que es una pared, o un suelo sobre el que el robot se está moviendo, sino que toda la información es extraída de los sensores del robot. Es esa información instantánea la que el robot extrae para reaccionar a cambios del entorno. Por este motivo las Arquitecturas Basadas en el Comportamiento se encuentran dentro de las ciencias cognitivas corporeizadas y situadas. Además, las Arquitecturas Basadas en el Comportamiento muestran comportamientos parecidos a los encontrados en la biología, en contraposición de los bien planeados y estructurados de las basadas en el conocimiento. Cierto es que este comportamiento en ocasiones es debido a errores y repeticiones, pero estos mismos defectos suelen ser ventajas a la hora de explorar nuevas soluciones en entornos no estructurados. Por todo ello, la respuesta de estas arquitecturas es mucho más rápida y acorde con un mundo no estructurado, como caricaturiza de nuevo Fitz en la Figura 4.3

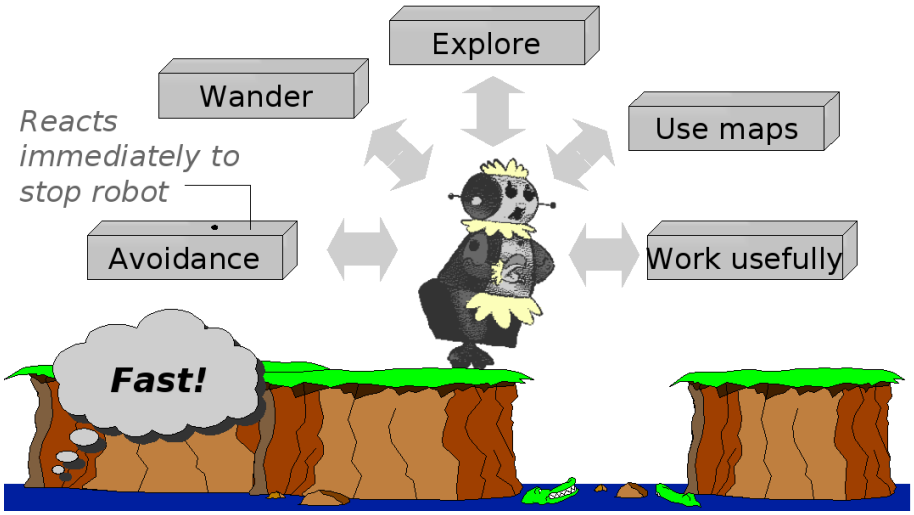


Figura 4.3: Caricatura de una Arquitectura Basada en el Comportamiento (Dibujo obtenido de Fitz Patrick)

4.3. Arquitectura Subsunción

Como se ha comentado anteriormente, la Arquitectura Subsunción pertenece al grupo de las Arquitecturas Basadas en el Comportamiento. Esta

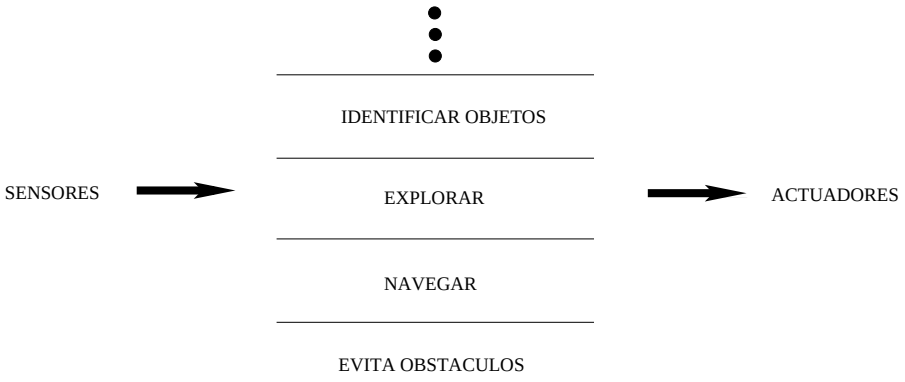


Figura 4.4: Arquitectura Subsunción

arquitectura se construye añadiendo capas de comportamientos unas sobre otras (ver Figura 4.4). La implementación de estos comportamientos se denominan niveles de competencia o capas (ver Sección 4.3.1). En vez de tener una secuencia de acciones (percepción, deliberación y acción) basadas en un modelo del mundo, existen diferentes caminos, los niveles de competencia, que se activan en paralelo. Cada uno de estos niveles de competencia se encarga de una pequeña subtarea del comportamiento del robot, de tal manera que puedan funcionar independientemente. Por lo tanto, cada capa no espera una instrucción proveniente de otra o un resultado que produce otro nivel de competencia, sino que, al ejecutarse en paralelo, el control es completamente descentralizado y por consiguiente no jerárquico. En resumen, cada nivel de competencia establece una relación directa entre los sensores y los actuadores con un pequeño procesamiento interno.

4.3.1. Niveles de competencia

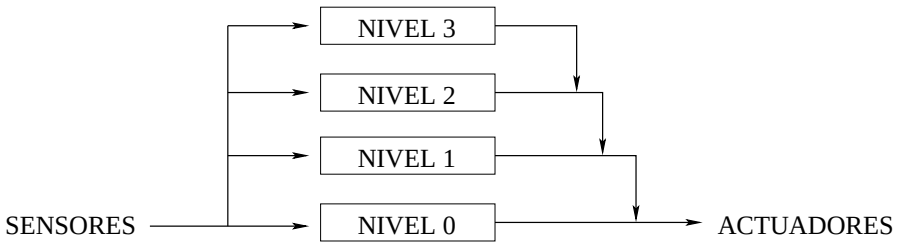


Figura 4.5: Ejemplo de una Arquitectura Subsunción: Niveles de competencia

Para crear una Arquitectura Subsunción es necesario comenzar definiendo los niveles de competencia. Un nivel de competencia es una definición

interna de un comportamiento externo deseado (por ejemplo navegar, evitar obstáculos, ...) que el robot debería ser capaz de realizar en el entorno en el que vaya a actuar. Los niveles superiores se apoyan en los niveles inferiores, en los que residen los comportamientos de más bajo nivel del robot (ver Figura 4.5).

Por ejemplo, un robot que deba moverse por el entorno sin chocar con los obstáculos podría tener una arquitectura de dos niveles de competencia. El primero y más bajo sería el de evitar obstáculos, donde el robot, en base a sus sensores, debe tomar el control de los motores para evitar chocarse cuando se encuentre cerca de una pared, silla, mesa, o cualquier objeto dentro de su entorno. El segundo a diseñar podría ser un nivel tan simple como andar en línea recta. En este caso el robot siempre estará andando en línea recta, pero cuando se encuentre con un obstáculo en su trayectoria debe girar para evitarlo.

Como se puede observar, uno de los puntos importantes en una Arquitectura Subsunción, es que gracias a que unos niveles de competencia pueden ser diseñados sin influir y, lo que es más importante, aprovechando los anteriormente diseñados. El concepto de escalabilidad surge instantáneamente en los que nuevos comportamientos pueden ser añadidos a controladores ya existentes y depurados.

Además, puesto que cada nivel de competencia, ya sean los más altos o bajos, pueden tener entradas sensoriales, ningún nivel requiere información del anterior, facilitando también el diseño en paralelo de las arquitecturas. También es posible que al diseñar nuevos comportamientos sea necesario modificar las salidas de ciertos niveles de competencia inferiores en alguna situación, para ellos los niveles superiores pueden inhibir o suprimir a los inferiores en momentos necesarios, gracias a los elementos de control de inhibición y supresión definidos en la arquitectura (ver Sección 4.3.3). Esta capacidad de "subsumir" los niveles inferiores es la que dio el nombre a la Arquitectura Subsunción. Sin embargo, aunque esta interacción pueda hacer que el comportamiento del robot sea rico en cuanto a sus posibilidad, la idea es minimizar estas interacciones para facilitar el proceso de diseño.

4.3.2. Módulos: Máquina de estados finitos

Una vez definidos los comportamiento, por lo tanto los niveles de competencia, es necesario implementar físicamente la arquitectura. Para ello, cada nivel de competencia consiste en un conjunto de módulos asíncronos conectados entre si (ver Figura 4.6). Cada módulo es una Máquina de Estados Finitos (MEF) Aumentada. Una MEF es un dispositivo computacional que cambia de estado dependiendo del estado en el que se encuentre así como de su entrada. Una MEF Aumentada (MEFA) es una MEF a la que se le han añadido ciertas habilidades extras. Típicamente estas habilidades suelen ser registros o temporizadores que permiten el paso entre estados después de un

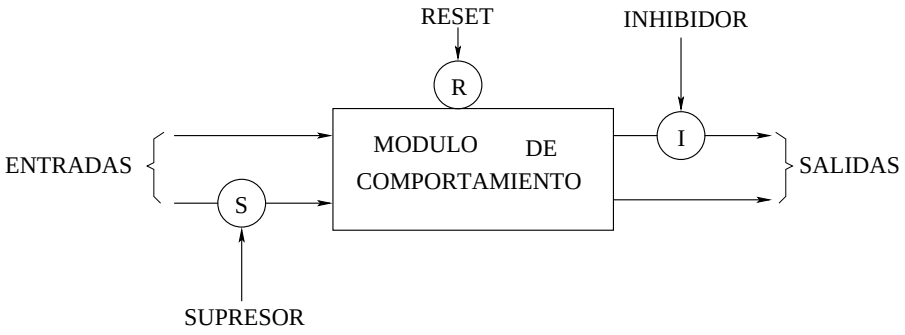


Figura 4.6: Módulo de comportamiento de una Arquitectura Subsunción

cierto periodo de tiempo o después de recibir varias veces un mismo estímulo (ver Figura 4.7).

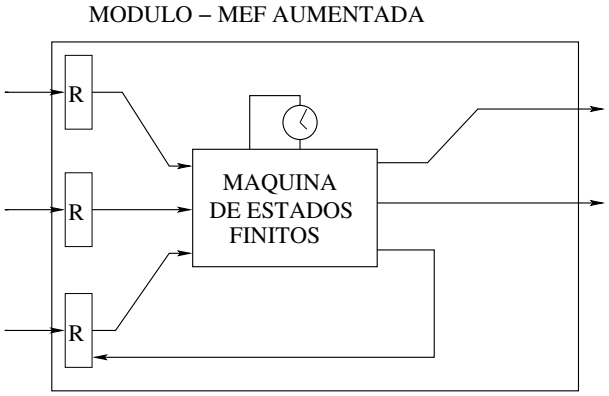


Figura 4.7: Máquina de Estados Finitos Aumentada

4.3.3. Elementos de control

Como se ha comentado anteriormente, es posible que niveles de competencia superiores necesiten modificar entras o salidas de niveles de competencia inferiores para adecuar los diferentes módulos a los comportamientos requeridos. Para ello se han definido dos elementos de control dentro de la Arquitectura Subsunción:

- **Inhibición**: Inhibe una señal, evitando que continúe su transmisión y por lo tanto introduciendo una alta impedancia en la misma línea. En la Figura 4.8 podemos observar 2 niveles de competencia conectados a través de un elemento inhibidor. En el caso en el que no exista inhibición, ya que el módulo "buscar comida" no activa su salida, la

salida del módulo “evitar obstáculos” es conectada directamente a los motores. Por contra, si la salida del módulo “buscar comida” es activada, ésta inhibe a la salida del módulo “evitar obstáculos” y bloquea la señal a los motores.

- **Supresión**: Suprime una señal, modificando su valor por la señal supresora. En la Figura 4.9 podemos observar 2 niveles de competencia conectados a través de un elemento supresor. En el caso en el que no exista supresión, ya que el módulo “buscar comida” no activa su salida, la salida del módulo “evitar obstáculos” es conectada directamente a los motores. Por contra, si la salida del módulo “buscar comida” es activada, ésta suprime la salida del módulo “evitar obstáculos” y hace que la salida del módulo “buscar comida” sea la que se conecte directamente a los motores.

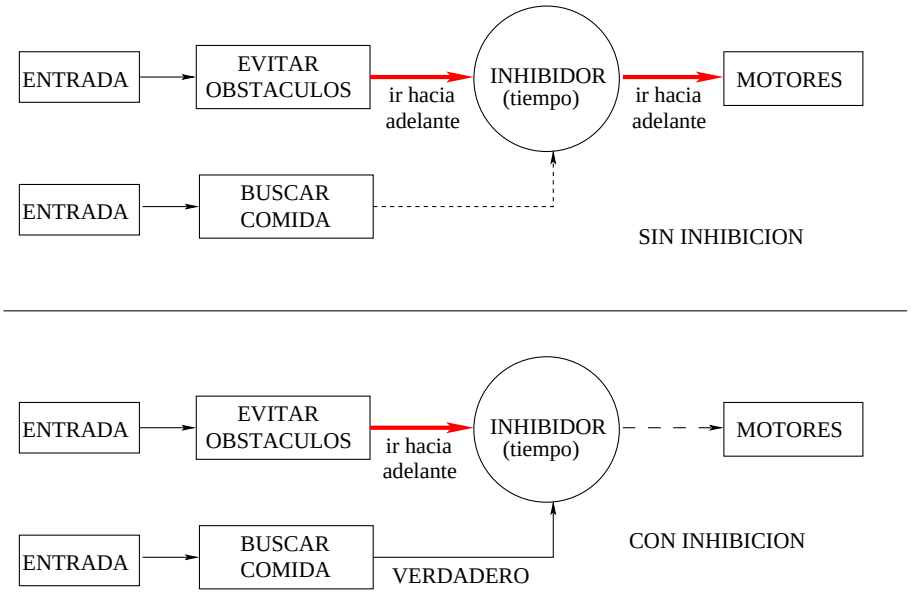


Figura 4.8: Elemento de control inhibidor.

4.4. Ejemplos

En esta sección presentamos 3 ejemplos basados en la Arquitectura Subsunción. Estos ejemplos pretenden ser una implementación de los mismos presentados en el Capítulo 3 pero en base a esta nueva arquitectura en vez de una red neuronal sintonizada con algoritmos genéticos.

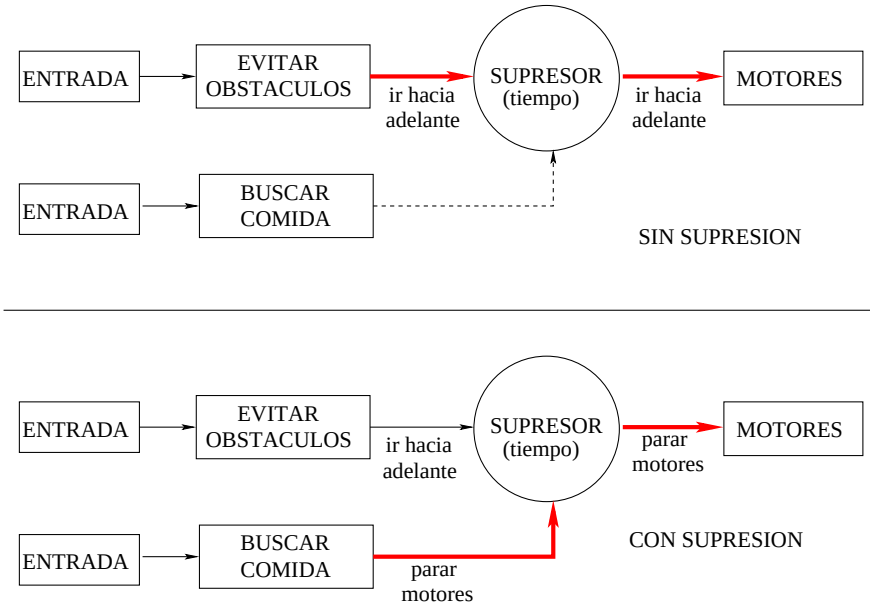


Figura 4.9: Elemento de control supresor.

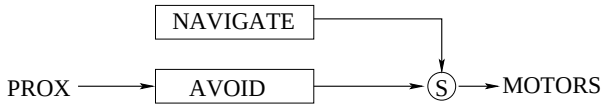


Figura 4.10: Arquitectura subsunción del experimento de navegación.

4.4.1. Navegación

Como se ha comentado anteriormente es posible implementar un robot que navegue sin chocar por un entorno no estructurado mediante una arquitectura de dos niveles de competencia (ver Figura 4.10):

- **Avoid**: nivel de competencia encargado de evitar obstáculos.
- **Navigate**: nivel de competencia encargado de navegar en línea recta.

Para este ejemplo supondremos un robot circular con 8 sensores de proximidad en su perímetro y dos motores en configuración diferencial. Tal y como se observa en la Figura 4.10, los sensores de proximidad son entradas del nivel de competencia *avoid*, mientras que el nivel de competencia *navigate* no tiene ninguna entrada sensorial. A continuación se describen unas posibles implementaciones de ambos niveles de competencia.

Navigate

Navigate se encargará exclusivamente del movimiento en línea recta del robot, por lo que si definimos M_l y M_r como las velocidades del motor izquierdo y derecho de nuestro robot respectivamente, el módulo implementará la siguiente función:

$$M_l = M_r = V \tag{4.1}$$

donde V es la velocidad máxima de movimiento del robot. La salida del módulo será el vector $\vec{M} = (M_l, M_r)$

Avoid

El nivel de competencia *avoid* es algo más complejo que el módulo anterior. Suponemos que nuestro robot tiene 8 sensores de infrarrojos ($I_{Si} \in [0,1]$, $i \in [1,8]$) con una posición angular θ_i para cada uno de ellos con respecto al sentido de movimiento del robot. El módulo se encarga de calcular la orientación a la que se encuentra el obstáculo más cercano (ϕ) mediante el valor de cada uno de los sensores y sus posiciones angulares:

$$\phi = atan\left(\frac{\sum_{i=1}^{8} I_{Si} * cos(\theta_i)}{\sum_{i=1}^{8} I_{Si} * sin(\theta_i)}\right) \tag{4.2}$$

Además calcula el máximo de los valores medidos por los sensores de proximidad. En caso de que dicho valor se encuentre por encima de un umbral ($max(I_{Si} \geq T_r)$, activará su salida suprimiendo la señal del módulo navigate, mediante el vector:

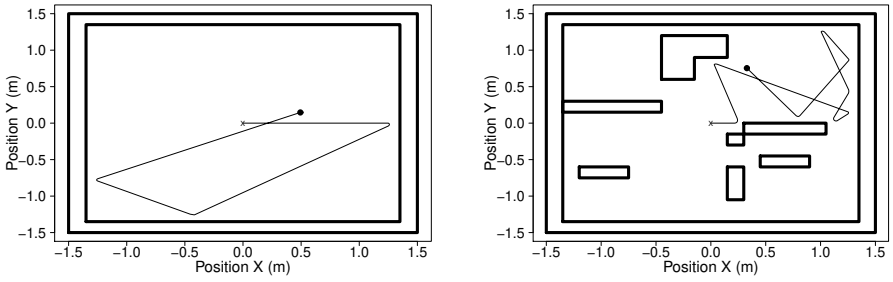
$$\vec{M} = (V * cos(\frac{\gamma}{2}) - \frac{V}{\pi} * \gamma, V * cos(\frac{\gamma}{2}) - \frac{V}{\pi} * \gamma) \tag{4.3}$$

donde γ representa el ángulo opuesto a la dirección del obstáculo ($\gamma = -\phi$) normalizado entre [$-\pi$,π].

Si comparamos los resultados de esta arquitectura con los de la del experimento navegar del Capítulo 3, observamos que el comportamiento del robot es similar al obtenido por el algoritmo genético. La Figura 4.11 muestra las trayectorias del robot en una arena idéntica (3x3 m^2) con y sin obstáculos, donde el robot es situado con las mismas condiciones iniciales.

4.4.2. Navegación y carga

Para implementar un experimento en el que el robot navegue por el entorno y sea capaz de cargar su batería (gracias a una célula fotovoltaica) por medio de una fuente de luz, sólo es necesario crear un tercer nivel de



(a) (b)

Figura 4.11: Experimento navegación. Trayectorias realizadas por el robot con una arquitectura subsunción de 2 niveles de competencia en (a) una arena sin obstáculos y (b) una arena con obstáculos.

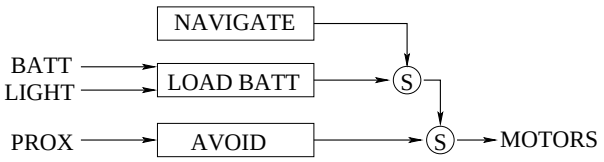


Figura 4.12: Arquitectura subsunción del experimento de navegación y carga.

competencia que se añadirá a los ya creados anteriormente (ver Figura 4.12). Este nivel de competencia, "cargar batería" se encargará de comprobar el estado de carga del robot y orientarse hacia la fuente de luz.

Load battery

Suponemos que nuestro robot tiene 8 sensores de luz ($L_{Si} \in [0,1]$, $i \in [1,8]$) con una posición angular θ_i para cada uno de ellos con respecto al sentido de movimiento del robot. Únicamente 2 sensores de luz, del vector de 8, son activados simultáneamente por la fuente de luz. Estos 2 sensores son los más cercanos a la fuente de luz. Además el robot dispone de un sensor de batería ($B_{S1} \in [0,1]$) donde 0 representa que la batería está completamente descargada y 1 completamente cargada.

El módulo se encarga de calcular el estado de carga con respecto a un umbral (T_r) determinado por el diseñador. Si la batería se encuentra cargada ($B_{S1} \geq T_r$) el módulo no realiza ninguna acción. En el caso en el que la batería se encuentre descargada ($B_{S1} < T_r$) el módulo se encarga de orientar el robot en el sentido de la luz. Si el robot ya se encuentra orientado ($L_{S1} \neq 0$ y $L_{S7} \neq 0$) el módulo no realiza ninguna acción. En el caso contrario, el robot

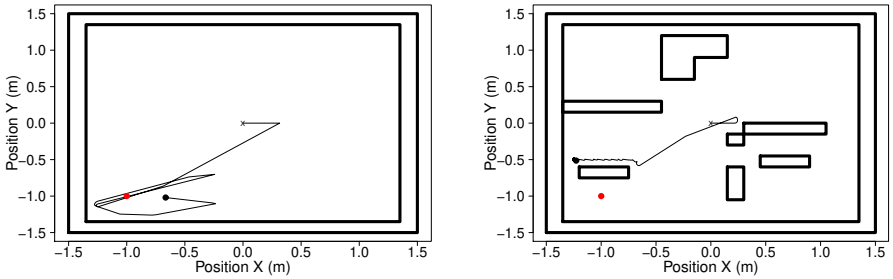
trata de orientarse con respecto a la misma:

$$\vec{M} = \begin{cases} (-V, V); & si\, L_{Si} > L_{Sj} \\ (V, -V); & si\, L_{Si} <= L_{Sj} \end{cases} \quad \text{donde}\, i \in [1,4], j \in [5,8] \tag{4.4}$$

Observamos que en el caso en el que la batería se encuentre por encima del umbral definido, el robot se comporta igual que en el experimento anterior. Cuando la batería se descarga por debajo del umbral definido se pueden observar diferentes comportamientos:

- Si el robot se encuentra orientado en dirección a la fuente de la luz, el nivel de competencia *navigate* tomará el control del robot, llevando al mismo hacia la zona de carga.
- Si el robot no se encuentra orientado hacia la luz, el nivel de competencia *load battery* se encarga de girar el robot hasta orientarlo para que posteriormente *navigate* se encargue de acercarlo a la misma.
- En el caso de que exista un obstáculo en la trayectoria del robot, *avoid* tomará el control de los motores hasta que se haya esquivado, y devolverá el control a los otros niveles de competencia. Esto implica que en el caso de encontrar obstáculos en dirección de la fuente de luz cuando el robot se encuentra sin batería, el comportamiento será que el robot se acercará a la misma, sorteando los obstáculos que se encuentre en el camino.

Si comparamos los resultados de esta arquitectura con los de la del experimento navegar y carga del Capítulo 3, observamos que el comportamiento del robot es similar al obtenido por el algoritmo genético. La Figura 4.13 muestra las trayectorias del robot en una arena idéntica (3x3 m^2) con y sin obstáculos, donde el robot es situado con las mismas condiciones iniciales. Podemos observar que en el caso de la arena sin obstáculos el robot realiza trayectorias rectilíneas en las que se aproxima y se aleja de la zona de carga. Desde el punto de vista del movimiento del robot, se ha mejorado con respecto a lo obtenido por el algoritmo genético dado que su movimiento es más rectilíneo tal y como se había definido el objetivo del robot. Sin embargo, si nos centramos en el experimento en la arena con obstáculos observamos que el robot no es capaz de sortear uno de los obstáculos que se encuentran delante de la fuente de luz, debido al tamaño del pasillo. En este caso, el robot no consigue acercarse a la zona de carga, y por lo tanto el diseño debería ser modificado para satisfacer los requisitos impuestos.



(a) (b)

Figura 4.13: Experimento navegación y carga. Trayectorias realizadas por el robot con una arquitectura subsunción de 3 niveles de competencia en (a) una arena sin obstáculos y (b) una arena con obstáculos.

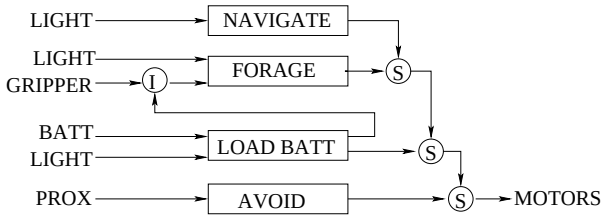


Figura 4.14: Arquitectura subsunción del experimento de recogida de objetos.

4.4.3. Recogida de objetos

Finalmente, queremos plantear el problema de recogida de objetos del entorno. Suponemos que en el entorno existen objetos que deben ser recogidos y trasladados a un centro de recogida por el robot. Por simplicidad de diseño hemos situado el centro de recogida de objetos en el extremo opuesto de donde se encuentra la fuente de luz, zona de carga. De tal manera que si el robot pretende ir a cargar la batería, se acercará a la fuente de luz, pero si debe depositar un objeto recogido, se desplazará en sentido contrario a la fuente de luz. Supongamos que el robot dispone de un sensor pinza ($G_{S1} \in 0, 1$) que se activa (1) cuando el robot ha cogido un objeto y (0) cuando no tiene ningún objeto. Este será el sensor que nos permita saber si tenemos que buscar objetos o depositarlos. Para llevar a cabo este ejemplo mantendremos la arquitectura desarrollada hasta ahora de 3 niveles de competencia y añadiremos 1 nivel de competencia más (*forage*). Además, será necesario incorporar un elemento inhibidor, para así mantener la funcionalidad completa de nuestro robot (ver Figura 4.14).

Es necesario modificar ligeramente el módulo *load battery* para disponer de una señal de inhibición dependiendo del estado de carga de la batería. Si la batería se encuentra cargada, la señal de inhibición no se activará, en caso contrario si. De esta manera, el módulo *load battery* inhibe la señal de entrada del sensor de la pinza, evitando que el módulo *forage* tenga en cuenta si el robot ha recogido un objeto o no.

Forage

El nivel de competencia *forage* comprueba el estado del sensor de la pinza. En el caso en el que no se encuentre activo, ya sea por no haber recogido ningún objeto o porque la señal está siendo inhibida, el módulo no realiza ningún acción. En el caso contrario, se comporta de una manera opuesta al módulo *load battery*. Comprueba si el robot se encuentra orientado en el sentido opuesto de la luz ($L_{S3} \neq 0$ y Ł$_{S4} \neq 0$). En este caso no realiza ninguna acción, y permite que el nivel de competencia *navigate* tome el control de los motores. En caso contrario, el robot trata de orientarse para alejarse de la luz:

$$\vec{M} = \begin{cases} (V, -V); & si\, L_{Si} > L_{Sj} \\ (-V, V); & si\, L_{Si} <= L_{Sj} \end{cases} \qquad (4.5)$$
$$\text{donde}\, i \in [1,4], j \in [5,8]$$

Si comparamos los resultados de esta arquitectura con los de la del experimento recogida de objetos del Capítulo 3, observamos que el comportamiento del robot es similar al obtenido por el algoritmo genético. La Figura 4.15 muestra las trayectorias del robot en una arena idéntica (3x3 m^2) sin obstáculos, donde el robot es situado con las mismas condiciones iniciales. Podemos observar que el comportamiento del robot es "mejor" que el obtenido con la arquitectura neuronal. En este caso observamos como el robot mantiene trayectorias rectilíneas, y ha conseguido encontrar y depositar 2 objetos en un tiempo de simulación de 300 segundos, mientras que el robot con la arquitectura neuronal, localizó sólo uno.

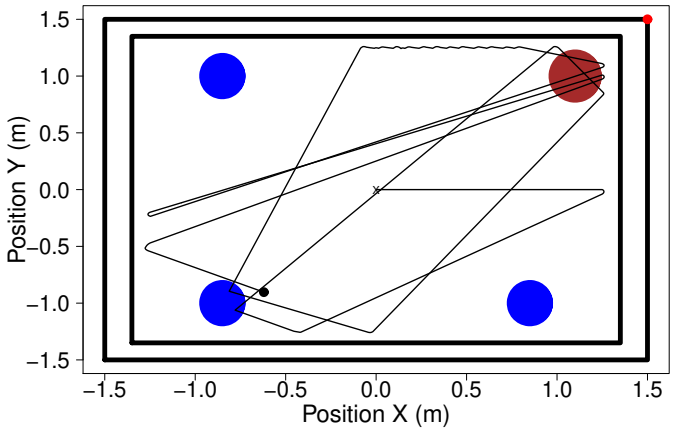


Figura 4.15: Experimento recogida de objetos. Trayectoria realizada por el robot con una arquitectura subsunción de 4 niveles de competencia en una arena sin obstáculos.

Parte II

Simulador

Capítulo 5

Introducción a IRSIM

5.1. Introducción

Irsim es un simulador robótico para el estudio de la asignatura *Introducción a la Robótica Inteligente*. Irsim es una versión reducida del simulador *twodeepuck* desarrollado en la *Universidad Libre de Bruselas*. Está desarrollado en C++, para aprovechar las ventajas de las herencias.

5.2. Instalación

Esta Sección presenta los pasos necesarios para la instalación, compilación y ejecución del simulador.

5.2.1. General

El simulador corre en Sistemas Operativos Linux de 32 bits. Se ha conseguido ejecutar correctamente el simulador en Linux de 64 bits y MACs, sin embargo para hacerlo funcionar en dichos sistemas es necesario tener un buen dominio de los mismos para localizar las librerías adecuadas y modificar las directivas de las autotools. Dichas modificaciones no se encuentran en este manual y no son materia de la asignatura, por lo que en el caso de que el alumno decida aventurarse a trabajar con dichos sistemas, será el mismo quien deba resolver dichos problemas por su cuenta.

5.2.2. Librerías

Dependiendo de la distribución que tengamos las versiones de las librerías varían. En esta Sección presentamos las librerías necesarias para la ejecución del simulador, mientras que en la Sección 5.2.3 se plantean unos ejemplos de su instalación.

Las librerías necesarias para la instalación son:

- automake $\geq$ 1.10
- libreadline-dev $\geq$ 5.0
- libgsl-dev $\geq$ 0.0
- libgl-mesa-dev $\geq$ 1.0
- libglu-mesa-dev $\geq$ 1.0
- libode-dev $\geq$ 0.0
- make $\geq$ 3.81
- g++ $\geq$ 4.1

Las versiones mínimas que se muestran anteriormente han sido comprobadas para el correcto funcionamiento del simulador, por lo que no existe garantía de funcionamiento para versiones anteriores de las librerías. Dependiendo de la distribución, existen diferentes formas de obtener cada una de las versiones de las librerías disponibles en los repositorios. Para distribuciones Debian y derivados (Ubuntu, por ejemplo), podemos ver las librerías que tenemos disponibles mediante el siguiente comando:

```
apt-cache search <NOMBRE_DE_LA_LIBRERIA>
```

De tal manera, si por ejemplo queremos saber cuales son las versiones disponibles de la librería `libgsl-dev`, ejecutaremos:

```
apt-cache search libgsl-dev
```

Esto nos dará una lista de librerías existentes en el repositorio. Habrá que escoger una de las librerías que cumplan los requisitos mencionados anteriormente. En la Sección 5.2.3 mostramos unos ejemplos.

5.2.3. Instalación de dependencias

Aquí vamos a presentar los requisitos necesarios para unas distribuciones Ubuntu y Debian. Para versiones diferentes de la misma distribución, o el resto de distribuciones, es necesario localizar las versiones específicas de cada paquete.

Ubuntu 13.10:

En línea de comando ejecutar:

```
sudo apt-get install automake libreadline6-dev libgsl0-dev \
libgl1-mesa-dev libglu1-mesa-dev libode-dev make g++
```

Debian 6 - Wheezy - (i386):

En línea de comando, como superusuario ejecutar:

```
apt-get install automake libreadline5-dev libgsl0-dev \
libgl1-mesa-dev libglu1-mesa-dev libode-dev make g++
```

Debian 7 - Jessie - (amd64):
En línea de comando, como superusuario ejecutar:

```
apt-get install automake libreadline6-dev libgsl0-dev \
libgl1-mesa-dev libglu1-mesa-dev libode-dev make g++
```

En el caso en el que nuestra versión de la distribución no posea alguna de las versiones de las librerías especificadas, el comando `apt-get` mostrará un error por pantalla. En este caso será necesario localizar cual es la librería errónea y substituirla por la versión correcta. Para ello hágase uso del comando `apt-cache search` tal y como se ha descrito en la Sección 5.2.2.

5.2.4. Descarga y compilación

Una vez instaladas todas las librerías es necesario descargarse el simulador desde http://www.robolabo.etsit.upm.es/iri/apuntes/.

El simulador hace uso de las autotools previamente instaladas (ver secciones 5.2.2 y 5.2.3). Para compilar el simulador es necesario ejecutar la primera vez las siguientes instrucciones:

- Descomprimir el paquete:

  ```
  tar -xvzf irsim-v<version>.tgz
  ```

- Ir al directorio donde se ha descomprimido:

  ```
  cd irsim-v<version>
  ```

- Ejecutar:

  ```
  ./bootstrap.sh
  ```

- Ejecutar:

  ```
  ./configure
  ```

- Ejecutar:

  ```
  make
  ```

La próxima vez que modifiquemos el código sólo será necesario hacer `make` para compilarlo.

Si al finalizar el make aparece un error, hay que presetar atención a qué es debido dicho error.

- **Si el error dice: /usr/bin/ld: no se puede encontrar -lGL"**:
  Debe resolverse de la siguiente manera:

  ```
  sudo unlink /usr/lib/x86_64-linux-gnu/libGL.so
  sudo ln -s /usr/lib/x86_64-linux-gnu/libGL.so.1 /usr/lib/x86_64-linux-gnu/libGL.so
  ```

5.2.5. Prueba de instalación

Para comprobar que el simulador se ha compilado correctamente ejecutar:

```
./irsim -E
```

Deberá aparecer por pantalla lo siguiente:

```
EXPERIMENTS IDENTIFIERS

1 - TEST WHEELS EXPERIMENT
2 - TEST CONTACT SENSOR EXPERIMENT
3 - TEST PROXIMITY SENSOR EXPERIMENT
4 - TEST LIGHT SENSOR EXPERIMENT
5 - TEST BLUE LIGHT SENSOR EXPERIMENT
6 - TEST RED LIGHT SENSOR EXPERIMENT
7 - TEST SWITCH LIGHT EXPERIMENT
8 - TEST GROUND SENSOR EXPERIMENT
9 - TEST BATTERY SENSOR EXPERIMENT
10 - TEST ENCODER SENSOR EXPERIMENT
11 - TEST COMPASS SENSOR EXPERIMENT

20 - TEST BRAITENBERG VEHICLE EXPERIMENT
21 - TEST NEURON EXPERIMENT
22 - TEST SUBSUMPTION LIGHT EXPERIMENT
23 - TEST SUBSUMPTION GARBAGE EXPERIMENT

30 - TEST IRI1 EXPERIMENT
31 - TEST IRI2 EXPERIMENT
32 - TEST IRI3 EXPERIMENT
```

Si esto es así, la compilación del simulador habrá sido correcta.

5.3. Estructura

En esta sección se muestran las partes fundamentales y de necesaria comprensión para poder trabajar con el simulador. Estas partes son las siguientes:

- **Experimentos**: Donde se define el experimento a llevar a cabo.
- **Actuadores**: Donde se definen los actuadores del robot.
- **Sensores**: Donde se definen los sensores del robot.
- **Controladores**: Donde se define el controlador a utilizar.

Antes de comenzar detallando cada una de las partes, vamos a detenernos en la explicación del **Fichero de Configuración**.

5.3.1. Fichero de configuración

El fichero de configuración es el encargado de suministrar unos parámetros al simulador. Tiene una estructura de grupos que deben aparecer siempre en el mismo orden. Sin embargo, no todos los grupos serán utilizados en todos los experimentos. Por simplicidad de uso en la asignatura, se han creado 2 tipos de ficheros de configuración. El primero es el que incorpora todos los grupos comunes, que será utilizado por los vehículos de Braitenberg y la arquitectura Subsunción. El segundo incorpora todos los elementos del anterior, más las partes correspondientes a redes neuronales y arquitectura evolutivas. En esta Sección nos centramos en el primer tipo de fichero, que incorpora los grupos comunes a todos las arquitecturas. Los grupos particulares se detallarán en los capítulos específicos a la par que se explican con ejemplos.

La existencia del fichero de configuración nos permite una flexibilidad en el desarrollo de aplicaciones, ya que cualquier modificación de valores del mismo no requiere recompilar el programa.

El fichero de configuración es interpretado línea a línea. Cada línea es leída hasta que encuentra un signo "=", obteniendo el o los números que aparecen a continuación. Téngase en cuenta que el programa no "parsea" los caracteres anteriores al signo igual, sino que dichos caracteres son únicamente utilizados para ayudarnos a la comprensión del fichero. Por lo tanto, el orden de los parámetros en el fichero de configuración es estricto, y no permite modificación.

El fichero de configuración está dividido en grupos que a su vez se encuentran divididos en bloques y estos en sub-bloques. Encontraremos varios ejemplos en el directorio `paramFiles`.

Lo primero que nos encontraremos en el fichero es el grupo `EXTRA`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% EXTRA %%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NUMBER OF ROBOTS                = 1
ROBOT 1: X POSITION (meters)    = 0.0
ROBOT 1: Y POSITION (meters)    = 0.0
ROBOT 1: ORIENTATION (radians)  = 0.0
WRITE TO FILE (0 No, 1 YES )    = 0
RUN TIME                        = 10000
```

Este grupo define primero el número de robots en nuestra simulación (`NUMBER OF ROBOTS`). Para cada robot, las 3 líneas siguientes representan la posición X, posición Y y la orientación del robot. Debemos escribir tantos conjuntos de 3 líneas como número de robots haya en el sistema. En el ejemplo propuesto sólo existe un robot, por lo que sólo hay 3 líneas indicando

sus coordenadas, en el caso que tuviéramos 2 robots deberíamos tener 6 líneas y así sucesivamente.

Se ha añadido una línea para la escritura de datos en ficheros. Esta línea se codifica dentro del código en la variable `m_nWriteToFile`. Esta variable permite que controlemos si queremos escribir datos a los ficheros o no. Como se verá en el Capítulo 7 cuando utilizamos un controlador neuronal, éste se encarga de escribir datos a los ficheros automáticamente. En el caso de los controladores de subsunción y braitenberg, es el usuario quien se encarga de utilizar dicha variable para escribir o no en ficheros (ver ejemplos en los Capítulos 6 y 9). Finalmente, `RUN TIME` es el tiempo de ejecución del simulador en segundos.

El siguiente grupo es el encargado de definir las opciones del entorno, `ENVIRONMENT`.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% ENVIRONMENT %%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NUMBER OF LIGHT OBJECTS                    = 1
LIGHT 1: X POSITION                        = -1
LIGHT 1: Y POSITION                        = 1
NUMBER OF BLUE LIGHT OBJECTS               = 1
BLUE LIGHT 1: X POSITION                   = 1
BLUE LIGHT 1: Y POSITION                   = 0
NUMBER OF RED LIGHT OBJECTS                = 1
BLUE LIGHT 1: X POSITION                   = 0
BLUE LIGHT 1: Y POSITION                   = 1
NUMBER OF GROUND AREA                      = 1
GROUND AREA 1 X POSITION                   = 0.0
GROUND AREA 1 Y POSITION                   = 0.0
GROUND AREA 1 RADIUS                       = 0.1
GROUND AREA 1 COLOR (0.0 Black, 0.5 Grey) = 0.5
```

Este grupo define las fuentes de luz y las baldosas de suelo que hay en el entorno (ver Sección 5.3.2). Primero es necesario especificar cual es número fuentes de luz (`NUMBER OF LIGHT OBJECTS`). En nuestro simulador disponemos de tres fuentes de luz, luz amarilla, azul y roja. Primero es necesario especificar el número de luces amarillas. Por cada fuente de luz es necesario especificar cual es su posición `X` e `Y` en metros. Por lo tanto si disponemos de una fuente de luz, como en el ejemplo presentado, tendremos 2 líneas definiendo su posición. Si tuviéramos 2 fuentes de luz, tendríamos 4 líneas, 2 para cada una de las coordenadas de cada fuente. Por último, si no tuviéramos fuentes de luz, por lo tanto `NUMBER OF LIGHT OBJECTS = 0`, no habría ninguna línea definiendo su posición X e Y. El mismo procedimiento se aplica a las fuentes de luz azul y roja.

Posteriormente definimos el número de baldosas de suelo. Por cada baldosa es necesario especificar 4 parámetros. 2 correspondientes a la posición,

uno para el radio y otro para el color. Las posiciones se especifican también en metros, así como el radio de la baldosa circular. Por último necesitamos especificar cual es el color de la baldosa, 0.0 si es Negra y 0.5 si es Gris. Nótese que el suelo de la arena es blanco, por lo que el robot distinguirá únicamente entre 3 tonos de grises (Blanco, Gris y Negro, ver Sección 5.3.4).

El siguiente grupo (`SENSORS`) se encarga de definir algunos parámetros de los sensores.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%% SENSORS %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

LIGHT SENSOR RANGE (meters)           = 2.0
BLUE LIGHT SENSOR RANGE (meters)      = 1.0
RED LIGHT SENSOR RANGE (meters)       = 0.5
BATTERY SENSOR RANGE (meters)         = 0.5
BATTERY CHARGE COEF                   = 0.01
BATTERY DISCHARGE COEF                = 0.0001
BLUE BATTERY SENSOR RANGE (meters)    = 0.5
BLUE BATTERY CHARGE COEF              = 0.01
BLUE BATTERY DISCHARGE COEF           = 0.0001
RED BATTERY SENSOR RANGE (meters)     = 0.5
RED BATTERY CHARGE COEF               = 0.01
RED BATTERY DISCHARGE COEF            = 0.0001
ENCODER ERROR                         = 0.0
```

El primer parámetro (`LIGHT SENSOR RANGE`) hace referencia al sensor de luz amarilla, y especifica cual es su rango. Esto permite que tengamos un sensor de luz "ad-hoc" para nuestros experimentos. El valor que introduzcamos, en metros, será la distancia máxima a la que un sensor de luz es capaz de detectar una fuente de luz. El mismo procedimiento se aplica a las fuentes de luz azul y roja.

Los otros 3 grupos de 3 parámetros hacen referencia a la batería y su sensor. Estamos asumiendo que el robot incorpora una célula fotoeléctrica que hace que la batería se cargue cuando se encuentra más cerca que un umbral de una fuente de luz amarilla, azul o roja. Este umbral, se define en nuestro fichero de configuración como `BATTERY SENSOR RANGE` con su valor en metros. En el ejemplo propuesto, la batería empieza a cargarse cuando el robot se encuentra a una distancia menor de 0.5 m de una fuente de luz amarilla. Los otros dos parámetros son parámetros de la ecuación de carga y descarga de la batería (ver Sección 5.3.4). El mismo procedimiento se aplica a las baterías asociadas a las fuentes de luz azul y roja.

Por último, el parámetro `ENCODER ERROR` hace referencia al error en el encoder. Dicho error se presenta como un número aleatorio uniforme de $\pm$ `ENCODER ERROR`.

5.3.2. Experimentos

Una vez introducido el fichero de comunicación vamos a describir las diferentes partes del simulador. En el directorio `experiments` encontramos todos los experimentos actualmente disponibles en el simulador. Un experimento define *i*) el tamaño de la arena en el que se va a trabajar, *ii*) el número de robots junto con *iii*) sus sensores/actuadores y *iv*) los objetos disponibles. Un experimento consta de los siguientes métodos:

- **Constructor**: Construye el experimento e inicia los valores por defecto.
- **Destructor**: Destruye el experimento.
- **CreateArena**: Crea el entorno del experimento
- **AddActuators**: Crea y añade actuadores al robot.
- **AddSensors**: Crea y añade sensores al robot
- **SetController**: Crea y añade el controlador al robot
- **CreateAndAddEpucks**: Crea y añade los robots
- **Reset**: Resetea el experimento

En el simulador ya se encuentran unos experimentos (`iri1exp`, `iri2exp` e `iri3exp`) creados para que el alumno implemente sus algoritmos dentro de la asignatura. También se han desarrollado unos experimentos a modo de ejemplo para el manejo del simulador en la Sección 5.6. Además se han creado unos experimentos para cada una de las diferentes arquitecturas que se describen en los capítulos dedicados a cada una de ellas. En los siguientes apartados detallamos cada uno de los métodos para el experimento `iri1exp`, similar al resto de los experimentos.

Constructor

La definición del constructor es la siguiente:

```
CIri1Exp::CIri1Exp(const char* pch_name, const char* paramsFile);
```

En dicho constructor recogemos los valores suministrados en el fichero de configuración.

Destructor

La definición del destructor es la siguiente:

```
CIri1::~CIri1Exp ( void );
```

En el destructor liberamos la memoria de las variables utilizadas.

CreateArena

Sintaxis:

```
CArena* CIri1Exp::CreateArena ( void );
```

En este método definimos la Arena. La arena es el entorno virtual en el que el robot ejecuta su controlador. Para la definición de la arena es necesario crear una cadena de caracteres que define los muros u obstáculos (%) y el espacio abierto (#). En los experimentos creados se ha definido una arena diáfana de 3x3 m2 con una rejilla de 20x20. La arena viene definida por:

```
static char* pchHeightMap =

"%%%%%%%%%%%%%%%%%%%%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%####################%"
"%%%%%%%%%%%%%%%%%%%%";
```

Dentro del método creamos la arena de la siguiente manera:

```
CArena* pcArena = NULL;
pcArena = new CProgrammedArena("CProgrammedArena", 20, 20, 3.0, 3.0);
((CProgrammedArena*)pcArena)->SetHeightPixelsFromChars(pchHeightMap,
```

Esto hace que el entorno creado sea igual al presentado en la Figura 5.1: En el caso de querer modificar el tamaño de la arena o la rejilla del simulador, es necesario modificar los cuatro últimos parámetros de la función:

```
pcArena=new CProgrammedArena("CProgrammedArena",GRIDx,GRIDy,LENGTHx,LENGTHy);
```

donde `GRIDx` y `GRIDy` son número enteros que definen la discretización del espacio en los ejes X e Y respectivamente, y `LENGTHx` y `LENGTHy` definen el tamaño de la arena en metros en el eje X e Y respectivamente. Téngase

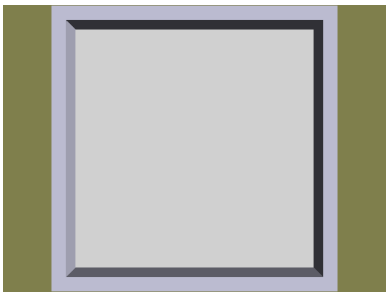


Figura 5.1: Arena creada en el experimento `Iri1exp`.

en cuenta de que en caso de modificar los parámetros `GRID` será necesario suministrar un `pchHeightMap` acorde a dichos valores; tantas filas como `GRIDx` y tantas columnas como `GRIDy`.

Por último, en este método también se añaden los objetos de la arena. Actualmente sólo disponemos de "fuentes de luz" y "baldosas de suelo" como objetos.

Fuentes de Luz: Una fuente de luz es una bombilla que se encuentra en el entorno del simulador. La intensidad de luz de estas fuentes serán detectadas por los sensores de luz. Para crearlas es necesario darles una posición (x,y) en el fichero de configuración.

Todas las fuentes de luz definidas en el fichero de configuración (amarillas, azules y rojas) se añaden al simulador de la siguiente manera:

```
/* Create and add Light Object */
char pchTemp[128];
CLightObject* pcLightObject = NULL;
for( int i = 0 ; i < m_nLightObjectNumber ; i++){
        sprintf(pchTemp, "LightObject%d", i);
        CLightObject* pcLightObject = new CLightObject (pchTemp);
        pcLightObject->SetCenter(m_pcvLightObjects[i]);
        pcArena->AddLightObject(pcLightObject);
}

[...]
```

Se ha creado un método (`GetTiming`) dentro de todos los objetos luces (amarillas, azules y rojas) que permite que el diseñador establezca cualquier tipo de secuencia de encendido/apagado.

Por simplicidad y permisibilidad se ha decidido no incorporar dicha secuencia dentro del fichero de configuración sino que la programación de la secuencia se hará directamente sobre el método `GetTiming`.

En la versión del simulador colgada en la página web se puede observar cómo en el fichero `objects/redlightobject.cpp` el método `GetTiming`

crea una secuencia periódica de encendido/apagado cada 110 `Simulation Step`. Por lo tanto, el diseñador puede crear cualquier secuencia de encendido/apagado en función del tiempo de simulación.

Baldosas de suelo: Una baldosa es una pieza circular que se introduce en el entorno para modificar el color del suelo. El color de dichas baldosas será detectado por los sensores de suelo (ver Sección 5.3.4). Para crearlas es necesario darles una posición (x,y), un radio y un color como se ha visto en la Sección 5.3.1. El color puede ser gris (0.5) o negro (0.0). Esto permitirá tener 3 tipos de colores en el simulador, ya que la arena se considera blanca (1.0).

Todas las baldosas de suelo definidas ene el fichero de configuración se añaden al simulador de la siguiente manera:

```
/* Create GroundArea */
char sGroundAreaName[100]="epuck";

for ( int i = 0 ; i < m_nNumberOfGroundArea ; i++)
{
        //Create GroundArea
        sprintf(sGroundAreaName,"groundArea%d",i);
        CGroundArea* groundArea = new CGroundArea(sGroundAreaName);
        groundArea->SetCenter(m_vGroundAreaCenter[i]);
        groundArea->SetExtRadius(m_fGroundAreaExternalRadius[i]);
        groundArea->SetIntRadius(m_fGroundAreaInternalRadius[i]);
        groundArea->SetColor(m_fGroundAreaColor[i]);
        groundArea->SetHeight(0.20);
        pcArena->AddGroundArea(groundArea);
}
```

AddActuators

Sintaxis:

```
CArena* CIri1Exp::AddActuators ( CEpuck* pc_epuck );
```

Creamos y añadimos los actuadores a cada uno de los robots. Actualmente, como se verá en la Sección 5.3.3, sólo disponemos de dos actuadores que son los motores. En los experimentos creados añadimos los actuadores como sigue:

```
/* Create and Add Wheels */
char pchTemp[128];
sprintf(pchTemp, "actuators_%s", pc_epuck->GetName());
CActuator* pcActuator = NULL;
pcActuator = new CWheelsActuator(pchTemp, pc_epuck);
pc_epuck->AddActuator(pcActuator);
```

AddSensors

Sintaxis:

```
void CIri1Exp::AddSensors(CEpuck* pc_epuck);
```

Creamos y añadimos los sensores a cada uno de los robots. Actualmente, disponemos de 10 tipos de sensores: proximidad, luz amarilla, luz azul, luz roja, contacto, suelo, suelo con memoria y 3 baterías asociadas a los sensores de luz. En este apartado sólo nos encargamos de mostrar como se introducen en el simulador y no las particularidades de cada uno que se detallan en la Sección 5.3.4.

```
/* Create and add Proximity Sensor */
CSensor* pcProxSensor = NULL;
pcProxSensor = new CEpuckProximitySensor(252);
pc_epuck->AddSensor(pcProxSensor);

/* Light Sensor */
CSensor* pcLightSensor = NULL;
pcLightSensor = new CLightSensor("Light Sensor", m_fLightSensorRange);
pc_epuck->AddSensor(pcLightSensor);

/* Blue Light Sensor */
CSensor* pcBlueLightSensor = NULL;
pcBlueLightSensor = new CBlueLightSensor("Blue Light Sensor",
                                         m_fBlueLightSensorRange);
pc_epuck->AddSensor(pcBlueLightSensor);

[...]
```

SetController

Sintaxis:

```
void CIri1Exp::SetController(CEpuck* pc_epuck);
```

Definimos el controlador a utilizar el cual ha tenido que ser creado con anterioridad (ver Sección 5.3.5) y se encuentra en el directorio `controllers`.

```
char pchTemp[128];
sprintf(pchTemp, "Iri1");
CController* pcController =
  new CIri1Controller(pchTemp, pc_epuck, m_nWriteToFile);
pc_epuck->SetControllerType( CONTROLLER_IRI1 );
pc_epuck->SetController(pcController);
```

CreateAndAddEpucks

Sintaxis:

```
void CIri1Exp::CreateAndAddEpucks(CSimulator* pc_simulator);
```

El simulador permite la creación de más de un robot (sin límite) por cada experimento. Sin embargo, para la asignatura de *Introducción a la Robótica Inteligente* trabajaremos en un principio con un único robot. En este método se crea cada robot (e-puck) y se añade al entorno de simulación junto con un nombre y su posición y orientación inicial

```
/* Create and add epucks */
char label[100] = "epuck";
for (int i = 0; i < m_nRobotsNumber; i++)
{
        sprintf(label, "epuck%0.4d", i);
        CEpuck* pcEpuck = CreateEpuck(label,
                                m_pcvRobotPositions[i].x,
                                m_pcvRobotPositions[i].y,
                                m_fRobotOrientations[i]);
        pc_simulator->AddEpuck(pcEpuck);
}
```

Mediante la instrucción `CreateEpuck(label, m_pcvRobotPositions[i].x, m_pcvRobotPositions[i].y, m_fRobotOrientations[i])` se modifica la posición inicial y orientación del robot en nuestro experimento en función de los valores introducidos en el fichero de configuración.

Reset

Sintaxis:

```
void CIri1Exp::Reset ( void );
```

Establece las condiciones iniciales del robot.

5.3.3. Actuadores

Como se ha comentado anteriormente disponemos únicamente de un tipo de actuador, los motores del robot.

Motores

Un robot está definido por 2 motores independientes situados en la parte izquierda y derecha del robot. Cada motor se controla independientemente. La velocidad de los motores se puede definir de dos maneras

- **Normalizada**: Valor entre 0 y 1 donde, 0 es máxima velocidad hacia atrás, 1 es máxima velocidad hacia adelante y 0.5 es parada.
- **No normalizada**: Valor entre -1000 y 1000, donde -1000 es máxima velocidad hacia atrás, 1000 es máxima velocidad hacia adelante y 0 es parada.

Se han definido estas dos interfaces para simplicidad a la hora de generar los controladores. Ya sea para experimentos de arquitecturas basadas en comportamiento, donde utilizaremos la interfaz de velocidad no normalizada, o experimentos con redes neuronales, donde utilizaremos la interfaz de velocidad normalizada.

Normalizada Para introducir la velocidad a los motores de la forma normalizada es necesario dar una instrucción por cada uno de los motores.

```
SetOutput(output, value)
```

Donde `output` es 0 para la rueda izquierda y 1 para la rueda derecha, y `value` es un `float` entre 0 y 1.

No normalizada Para introducir la velocidad a los motores de la forma no normalizada se utiliza una sola instrucción.

```
SetSpeed(left_speed,right_speed);
```

Donde `left_speed` y `right_speed` es un entero de valor entre -1000 y 1000 para la rueda izquierda y derecha respectivamente.

5.3.4. Sensores

Actualmente se han definido 10 tipos de sensores: contacto, proximidad, luz amarilla, luz azul, luz roja, suelo, suelo memoria y 3 baterías. Cada sensor viene definido por un array de N sensores.

Contacto

Existen 8 sensores de contacto ubicados en el perímetro del robot a ángulos fijos con respecto al frente del robot en sentido anti-horario. Los sensores de contacto se encuentran en los siguientes ángulos del perímetro del robot.

```
double CContactSensor::m_fContactSensorDir[NUM_CONTACT_SENSORS] =
       {0.2967, 0.8727, 1.5708, 2.6180, 3.6652, 4.7124, 5.4105, 5.9865};
```

Los sensores de contacto se activan cuando un robot choca con un objeto (muro, obstáculo, robot,...) en el entorno. Cuando un sensor es presionado contra un objeto adquiere el valor 1.0, mientras que cuando el robot no se ha chocado con nada tiene el valor 0.0.

Para leer los valores es necesario obtener el array de los 8 sensores mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Esta función devuelve un puntero a un array donde se encuentra el valor de los 8 sensores, ordenados en función de la posición angular previamente definida.

Proximidad

Existen 8 sensores de proximidad ubicados en el perímetro del robot a ángulos fijos con respecto al frente del robot en sentido anti-horario. Los sensores de proximidad se encuentran en los siguientes ángulos del perímetro del robot.

```
double CEpuckProximitySensor::m_fIrSensorDir[NUM_PROXIMITY_SENSORS] =
        {0.2967, 0.8727, 1.5708, 2.6180, 3.6652, 4.7124, 5.4105, 5.9865};
```

Los sensores de proximidad detectan un obstáculo cuando el robot se encuentra a una distancia menor de 28 cm del mismo. Los valores son mayores cuanto más cerca se encuentra el sensor del obstáculo (robot, muro, ...).

Para leer los valores es necesario obtener el array de los 8 sensores mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Esta función devuelve un puntero a un array donde se encuentra el valor de los 8 sensores, ordenados en función de la posición angular previamente definida. Los valores devueltos pertenecen al intervalo [0,1].

Luz

Existen 8 sensores de luz ubicados en el perímetro del robot a ángulos fijos con respecto al frente del robot en sentido anti-horario. Los sensores de luz se encuentran en los siguientes ángulos del perímetro del robot.

```
double CLightSensor::m_fLightSensorDir[NUM_LIGHT_SENSORS] =
        {0.2967, 0.8727, 1.5708, 2.6180, 3.6652, 4.7124, 5.4105, 5.9865};
```

Los sensores de luz detectan la presencia de objetos de luz a un máximo definido en el fichero de configuración como `LIGHT SENSOR RANGE` para la luz amarilla, `BLUE LIGHT SENSOR RANGE` para la luz azul y `RED LIGHT SENSOR RANGE` para la luz roja. Sólo 2 sensores se activan al mismo tiempo, los 2 más cercanos a la fuente de luz (`LightObject`). El valor devuelto depende de la distancia a la fuente de la luz así como del ángulo de incidencia sobre el sensor.

Para leer los valores es necesario obtener el array de los 8 sensores mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Esta función devuelve un puntero a un array donde se encuentra el valor de los 8 sensores, ordenados en función de la posición angular previamente definida. Los valores devueltos pertenecen al intervalo [0,1].

Se ha creado un método que permite a los robots apagar/encender la luz más cercana al robot. Dicho método, `void SwitchNearestLight (int`

`n_value)`, se ha incorporado a la clase de sensores de luz (amarilla, azul y roja), donde `n_value=0` apaga la luz y `n_value=1` la enciende.

De esta manera, definiendo una variable `m_seLight` como:

```
CLightSensor* m_seLight =
(CLightSensor*) m_pcEpuck->GetSensor(SENSOR_LIGHT);
```

podremos encender/apagar la luz amarilla más próxima mediante las instrucciones:

`m_seLight->SwitchNearestLight(1)` y `m_seLight->SwitchNearestLight(0)` respectivamente

Igualmente haríamos uso del actuador para las luces azules:

```
CBLueLightSensor* m_seLight =
(CBlueLightSensor*) m_pcEpuck->GetSensor(SENSOR_BLUE_LIGHT);
```

y rojas:

```
CRedLightSensor* m_seLight =
(CRedLightSensor*) m_pcEpuck->GetSensor(SENSOR_RED_LIGHT);
```

Téngase en cuenta que el funcionamiento de este método con el de secuencias de encendido no es posible, ya que ambos sistemas pretenderían apagar/encender las luces en momentos distintos, siendo en algunos casos órdenes opuestas.

Suelo

Existen 3 sensores de suelo ubicados en la parte inferior-frontal del robot. Estos sensores de suelo distinguen entre 3 niveles de grises (blanco, gris y negro). Como se ha comentado anteriormente existen unas baldosas de suelo (ver Sección 5.3.2), que son detectadas por los sensores de suelo. Estos objetos son circulares y pueden tomar los valores 0.0 (negro) o 0.5 (gris). La arena es blanca (1.0), por lo que el sensor es capaz de detectar si el robot se encuentra en la arena o sobre alguna de las baldosas.

Para leer los valores es necesario obtener el array de los 3 sensores mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Esta función devuelve un puntero a un array donde se encuentran el valor de los 3 sensores, ordenados de la siguiente manera: izquierda, centro y derecha. Los valores devueltos pertenecen al conjunto {0.0, 0.5, 1.0}.

Suelo con memoria

El sensor de suelo con memoria es un sensor virtual basado en los sensores de suelo. Sólo tiene en cuenta el sensor central y se define como un dispositivo de retención. El sensor se inicializa a 0.0. Si el robot entra en un área gris el sensor pasa a tener el valor 1.0. Este valor se mantiene hasta que el robot entra en un área negra que pasa de nuevo a tener el valor 0.0.

Para leer los valores es necesario obtener el array de 1 sensor mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Esta función devuelve un puntero a un array donde se encuentra el valor del sensor. Los valores devueltos pertenecen al conjunto {0.0, 1.0}.

Batería

El sensor de batería es un sensor virtual que nos dice cual es el estado de la batería del robot. Se han definido 3 baterías asociadas a las diferentes fuentes de luz.

La batería se inicializa a 1.0 y su valor va disminuyendo cada *time step* un valor constante definido en el fichero de configuración por `BATTERY DISCHARGE COEF`. Si el robot se encuentra a menos de una distancia, definida en el fichero de configuración por `BATTERY SENSOR RANGE` en metros de una fuente de luz, la batería comienza a cargarse [1]. La carga depende de la energía almacenada y cumple la siguiente ecuación $bat(t+1) = \beta(1-bat(t)^2)$, donde β es el parámetro `BATTERY CHARGE COEF` definido en el fichero de configuración.

Para leer los valores es necesario obtener el array de 1 sensor mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Esta función devuelve un puntero a un array donde se encuentran el valor del sensor. Los valores devueltos pertenecen al intervalo [0, 1].

Encoder

El encoder es un sensor que nos indica el desplazamiento de cada una de las ruedas en cada tiempo de simulación. Se han definido 2 encoders, uno por cada rueda, y el valor devuelto es el desplazamiento lineal de cada rueda. Para leer los valores es necesario obtener el array de 2 sensores mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

Brújula

La brújula es un sensor que nos indica la orientación del robot con respecto a un sistema inercial. El valor devuelto es un doble que pertenece al intervalo $[0,2\pi]$. Para leer los valores es necesario obtener el array de 1 sensor mediante la siguiente instrucción.

```
double* GetSensorReading(CEpuck* pc_epuck);
```

[1] Suponemos que existe una célula fotoeléctrica en el robot.

5.3.5. Controladores

En el directorio `controllers` encontramos los controladores asociados a los experimentos. El controlador es quien define las acciones a realizar por los robots en cada instante de simulación. Aquí podremos incorporar una arquitectura de Braitenberg, una arquitectura basa en comportamientos, una red neuronal, o cualquier controlador que se nos ocurra. En este apartado definimos los métodos necesarios para cada controlador.

Un controlador viene definido por los siguientes métodos:

- **Constructor**: Construye el controlador e inicia los valores por defecto.
- **Destructor**: Destruye el controlador.
- **SimulationStep**: Es donde se encuentra el código a ejecutar por el robot.

Igual que en el apartado de experimentos, en este apartado hacemos referencia al controlador `iri1controller`.

Constructor

La definición del constructor es la siguiente:

```
CIri1Controller::CIri1Controller( const char* pch_name,
                                  CEpuck* pc_epuck,
                                  int n_write_to_file);
```

En el constructor inicializamos las variables necesarias para la ejecución del experimento y definimos los sensores y actuadores de nuestro robot.

```
/* Set Write to File */
m_nWriteToFile = n_write_to_file;

/* Set epuck */
m_pcEpuck = pc_epuck;
/* Set Wheels */
m_acWheels = (CWheelsActuator*) m_pcEpuck->GetActuator(ACTUATOR_WHEELS);
/* Set Prox Sensor */
m_seProx = (CEpuckProximitySensor*) m_pcEpuck->GetSensor(SENSOR_PROXIMITY);
/* Set light Sensor */
m_seLight = (CLightSensor*) m_pcEpuck->GetSensor(SENSOR_LIGHT);
/* Set contact Sensor */
m_seContact = (CContactSensor*) m_pcEpuck->GetSensor (SENSOR_CONTACT);
/* Set ground Sensor */
m_seGround = (CGroundSensor*) m_pcEpuck->GetSensor (SENSOR_GROUND);
/* Set ground memory Sensor */
m_seGroundMemory = (CGroundMemorySensor*)
        m_pcEpuck->GetSensor (SENSOR_GROUND_MEMORY);
/* Set battery Sensor */
m_seBattery = (CBatterySensor*) m_pcEpuck->GetSensor (SENSOR_BATTERY);
```

Nótese que `m_nWriteToFile`, `m_pcEpuck`, `m_acWheels`, `m_seProx`, `m_seLight` y `m_seContact` han sido previamente definidos en `iri1controller.h`.

`m_nWriteToFile` es una variable que nos permite saber si se ha introducido un 0 o 1 en el parámetro `WRITE TO FILE` del fichero de configuración. `m_pcEpuck` es un puntero al robot, que nos permitirá obtener su posición y orientación para su escritura a ficheros.

Destructor

La definición del destructor es la siguiente:

```
CIriController::~CIri1Controller ( void );
```

Actualmente no realiza ninguna acción

SimulationStep

Sintaxis:

```
void CIri1Controller::SimulationStep(unsigned n_step_number,
        double f_time, double f_step_interval);
```

Este método es el más importante del controlador. Aquí se definen las acciones a realizar por el robot. Normalmente, dichas acciones se encuentran divididas en 3 partes importantes: lectura de sensores, controlador y actuación.

Para leer cada uno de los sensores, debemos hacer uso de las funciones definidas en la Sección 5.3.4. Los 3 controladores generados a modo de ejemplo (`iri1controller`, `iri2controller` e `iri3controller`), realizan la lectura de todos los sensores:

```
/* Leer Sensores de Contacto */
double* contact = m_seContact->GetSensorReading(m_pcEpuck);
/* Leer Sensores de Proximidad */
double* prox = m_seProx->GetSensorReading(m_pcEpuck);
/* Leer Sensores de Luz */
double* light = m_seLight->GetSensorReading(m_pcEpuck);
/* Leer Sensores de Suelo */
double* ground = m_seGround->GetSensorReading(m_pcEpuck);
/* Leer Sensores de Suelo Memory */
double* groundMemory = m_seGroundMemory->GetSensorReading(m_pcEpuck);
/* Leer Battery Sensores de Suelo Memory */
double* battery = m_seBattery->GetSensorReading(m_pcEpuck);
```

Para poder utilizar los sensores, cada uno de ellos ha tenido que ser inicializado en el constructor. Igualmente, es importante recordad que todos los sensores definidos en el constructor y utilizados en el `SimulationStep` han tenido que ser creados y añadidos al robot en la definición del experimento (ver Sección 5.3.2).

Por último, se activan los motores con los valores definidos por el controlador.

```
/* Option 1: Speed between -1000, 1000*/
m_acWheels->SetSpeed(500,500);

/* Option 2: Speed between 0,1*/
m_acWheels->SetOutput(0,0.75);
m_acWheels->SetOutput(1,0.75);
```

5.4. Visualización

Para ejecutar un experimento es necesario introducir unos comandos una vez que el entorno gráfico ha arrancado. Podemos realizar una ejecución paso a paso mediante el comando `<Ctrl>+o` o una ejecución contínua mediante el comando `<Ctrl>+p`. Para pausar el simulador presionaremos de nuevo `<Ctrl>+p`.

Una vez arrancada la simulación existen diferentes opciones que poder configurar para obtener una mejor visualización.

5.4.1. Movimiento de la cámara

Para visualizar nuestra arena desde diferentes ángulos, debemos utilizar el ratón. Si presionamos el botón izquierdo y lo mantenemos presionado a la vez que movemos el ratón, obtendremos un desplazamiento rotacional de la cámara de visualización. Si por el contrario realizamos la misma operación con el botón derecho, realizaremos movimientos de translación. Por último, si mantenemos presionados los dos botones, nos alejaremos/acercaremos a la arena.

5.4.2. Mostrar sensores

Se han definido unas interfaces para poder visualizar los diferentes sensores que incorpora nuestro robot:

- Si presionamos la tecla i se mostrarán los sensores de proximidad.
- Si presionamos la tecla l se mostrarán los sensores de luz amarilla.
- Si presionamos la tecla b se mostrarán los sensores de luz azul.
- Si presionamos la tecla c se mostrarán los sensores de contacto.

5.4.3. Suprimir la visualización

Puesto que en ciertos momentos es posible que no sea necesario visualizar el entorno gráfico, se ha implementado un comando que permite ejecutar el simulador en modo `NO VISUAL`. Esto permite realizar ejecuciones más rápidas sin depender de la tarjeta gráfica.

Para ello debemos ejecutar el simulador con la opción "-v":

```
./irsim -E <EXP> -v
```

5.4.4. Otros parámetros

Por último existen otras combinaciones que nos ofrecen diferentes posibilidades:

- `<Ctrl>+w` recoge imágenes y las almacena en el directorio `frames`.
- `<Ctrl>+v` muestra las coordenadas de posicionamiento de la cámara.
- `<Ctrl>+s` activa/desactiva las sombras.
- `<Ctrl>+t` activa/desactiva las texturas.

5.5. Gráficas

En el subdirectorio `graphics` se encuentra el programa `irsimGnuplot` que permite generar gráficas a partir de ficheros de datos utilizando el programa GNUPLOT. Deberá instalarse del repositorio todo lo necesario para que funcione `gnuplot`. Deberá también instalarse el programa de conversión de formatos `sam2p` ya que `irsimGnuplot` hace uso de él.

Para crear el ejecutable `irsimGnuplot` deberá ejecutarse la instrucción `make` en línea de comandos y dentro del subdirectorio `graphics`. Una vez hecho esto, se ejecutará el comando:

```
./irsimGnuplot
```

Aparecerá el siguiente menú:

```
      j:ejes; g:rejilla
     RATÓN: (B3:ZOOM; ESC:salir,n:siguiente,p:previo,u:deshacer zoom)
            (B1:copiar)(B2:marcar)
0:salir
1:plot
2:multiplot (no admite raton)
3: presenta fichero configuración(<FILENAME>.gnu)(¡no funciona en multiplot!)
tipo=
```

Una vez que se haya generado la gráfica en la pantalla del ordenador podrá utilizarse el ratón para hacer un ZOOM con el botón derecho (B3).

La opción `multiplot`, *tipo* = 2, permite la creación de dos gráficas distintas en la misma imagen.

Cuando se elige la opción *tipo* = 1 se crea un fichero con la extensión `.gnu` que guarda la última configuración que se haya realizado, y que puede ser leído mediante la opción *tipo* = 3.

La opción *tipo* = 1 irá solicitando la siguiente información de configuración de los gráficos:

1. Ubicación del fichero de datos.

   ```
   NOMBRE DE FICHERO DE DATOS: bb/aa
   ```

   Esto significa que se tiene el fichero de datos `aa` que se encuentra en el subdirectorio `bb`. Los ficheros de datos deben estar formados po tantas columnas de números (en notación decimal) como se desee. Las gráficas resultantes podrán estar formadas por varias curvas superpuestas donde el eje X será una de las columnas y el eje Y otra de ellas o la misma, como se explica más adelante.

2. Opciones de presentación.

   ```
   1.gráfico por defecto
   2.gráfico con opciones de presentación
   opc= 2
   ```

   Si se elige *opc* = 1 el programa deja de pedir información a excepción de qué columnas deben cogerse del fichero de datos para la representación gráfica.

3. Ficheros de gráficas.

   ```
   MODO SUPERPOSICIÓN con el mismo eje X
   opciones terminal
   1:terminal x11
   2:imagen gif
   3:imagen png y eps
   opc= 1
   ```

   La opción *opc* = 1 elige el terminal para Linux, es decir la pantalla del ordenador. Cuando se salga del programa se perderá la imagen.

   Las opciones 2 y 3 generan imágenes con formatos gif, png y eps respectivamente, que dejará en el mismo subdirectorio que el fichero de datos. No funciona con la opción Multiplot. Con estas opciones no hace una presentación en pantalla.

4. Tìtulo general de la gráfica.

```
1: TITULO nombre de fichero
2: TITULO que se quiera
 opción= 1
```

5. Estilo del trazo de las curvas.

```
 ESTILO DEL TRAZO
1. lineas
2. puntos
3. impulsos
 trazo= 1
```

6. Títulos en los ejes.

```
 TÍTULOS EN LOS EJES X-Y? (S/N) n
```

7. Rangos de división de los ejes

```
  RANGOS DE DIVISIÓN EN LOS EJES X-Y

> AUTOAJUSTE EN X? (S/N) s

> AUTOAJUSTE EN Y? (S/N) s
```

8. Curvas a partir de un fichero de datos.

   Por último es necesario indicarle al programa qué columnas del fichero de datos constituirán los ejes X e Y de las gráficas. Lo que hay que indicarle es la posición de la columna en dicho fichero de datos. Se pueden representar varias curvas superpuestas, hasta que se le de el valor 0.

```
  GRÁFICAS. x:y:leyenda  (0:SALIR)
>> x1= 1
>> y1= 2
>> leyenda y1= posición
>> x2= 1
>> y1= 3
>> leyenda y1= velocidad
>> x2= 0
```

   En este ejemplo se obtendría en la misma gráfica, dos curvas superpuestas, siendo la primera columna del fichero de datos el eje X, y la segunda y tercera los valores del eje Y.

5.6. Ejemplos

Para mostrar el funcionamiento del simulador se han desarrollado diferentes ejemplos que hacen uso de los distintos sensores y actuadores. En esta

sección explicamos brevemente dichos ejemplos para que sirvan de guía al desarrollo de nuevos controladores. Antes de comenzar con los experimentos vamos a dar una pequeña introducción sobre una parte importante que es la escritura en ficheros.

5.6.1. Escritura en ficheros

Para poder escribir en un fichero es necesario crear un descriptor de fichero, donde se especifica la ruta y formato. Por ejemplo:

```
FILE* filePosition = fopen("outputFiles/robotPosition", "a");
```

Donde el nombre al que haremos referencia es `filePosition`, que se creará en la ruta `outputFiles/robotPosition` y se abrirá en formato `añadir`. Este formato añade lo que escribamos a continuación de lo que existe en el fichero.

Una vez abierto el fichero podemos escribir datos en él a través de la función `fprintf`. Por ejemplo:

```
fprintf(filePosition,"%2.4f %2.4f %2.4f %2.4f\n", m_fTime,
  m_pcEpuck->GetPosition().x,
  m_pcEpuck->GetPosition().y,
  m_pcEpuck->GetRotation());
```

De esta forma escribimos el instante de tiempo, la posición (x, y) y la orientación del robot en el fichero definido como `filePosition`.

Por último, antes de finalizar debemos cerrar el fichero, mediante la instrucción:

```
fclose(filePosition);
```

En algunos de los ejemplos propuestos detallaremos específicamente implementaciones sobre la escritura en ficheros.

5.6.2. TestWheels

Este ejemplo muestra el funcionamiento y uso de los motores del robot. Para ello se ha creado un controlador que hace que el robot gire sobre sí mismo. Los ficheros asociados al experimento son `testwheelsexp` y `testwheelscontroller` [2].

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 1
```

[2]Recuérdese que los ficheros de experimentos se encuentran en el directorio `experiments`, mientras que los ficheros de controladores se encuentran en el directorio `controllers`. Se considera importante tener los ficheros visibles a la vez que se está leyendo el manual para una mejor comprensión de los ejemplos.

En el experimento (`testwheelsexp.h` y `testwheelsexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0,0) con orientación 0 rad. El robot únicamente incorpora los motores como actuadores.

El controlador actualiza la velocidad de los motores en cada instante de muestreo (`SimulationStep`).

```
m_acWheels->SetOutput(0,0.75);
m_acWheels->SetOutput(1,0.25);
```

Si ejecutamos el simulador, observaremos que el robot gira sobre si mismo. Si quisiéramos modificar el movimiento del robot es necesario cambiar el valor del segundo parámetro de las funciones SetOutput.

Igualmente, como se ha comentado en la Sección 5.3.3, podemos utilizar la función no normalizada, cuyo equivalente al ejemplo anterior es el siguiente:

```
m_acWheels->SetSpeed(500,-500);
```

Se recomienda al alumno que trabaje sobre este ejemplo modificando los valores de velocidad de los motores para observar las modificaciones en el comportamiento del robot.

5.6.3. TestContactSensor

Este ejemplo muestra el funcionamiento y uso de los sensores de contacto. Para ello se ha creado un controlador en el que el robot se mueve en línea recta hasta que choca con una pared. En cada instante de muestreo, el simulador muestra por pantalla el valor de los sensores de contacto. Se observa que mientras que el robot se encuentra libre de obstáculos, todos los valores de los sensores están a 0.0. Cuando el robot colisiona con una pared, los valores de los sensores de contacto se ven modificados, cambiando a 1.0 los sensores que se encuentran en contacto con la pared. Los ficheros asociados al experimento son `testcontactexp` y `testcontactcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 2
```

En el experimento (`testcontactexp.h` y `testcontactexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0,0) con orientación 0 rad. El robot incorpora los sensores de contacto y los motores.

El controlador lee el valor de los sensores de contacto, los imprime por pantalla y actualiza la velocidad de los motores en cada instante de muestreo (`SimulationStep`):

```
/* Lee sensor de contacto */
double* contact = m_seContact->GetSensorReading(m_pcEpuck);
```

```
/* Imprimir los valores por pantalla */
printf("Contact Sensor Value: ");
for ( int i = 0 ; i < m_seContact->GetNumberOfInputs() ; i++)
{
        printf("%2f " , contact[i]);
}
printf("\n");

/* Muevete en línea recta */
m_acWheels->SetOutput(0,0.75);
m_acWheels->SetOutput(1,0.75);
```

Se recomienda que el alumno modifique la posición inicial, orientación y velocidad de movimiento del robot para adquirir manejo con el funcionamiento de los sensores.

5.6.4. TestProximitySensor

Este ejemplo muestra el funcionamiento y uso de los sensores de proximidad. Para ello se ha creado un controlador en el que el robot gira sobre sí mismo en una posición cercana a la pared. En cada instante de muestreo, el simulador muestra por pantalla el valor de los sensores de proximidad. Se puede observar que los sensores más cercanos a la pared obtienen valores más altos que los que se encuentran a mayor distancia. Los ficheros asociados al experimento son `testproxexp` y `testproxcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 3
```

En el experimento (`testproxexp.h` y `testproxexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.0,1.2) y con orientación 1.57 rad. El robot incorpora los sensores de proximidad y los motores.

Durante la ejecución del ejemplo se recomienda presionar la tecla "i" que mostrará el rango de visión de los sensores de infrarrojo y se podrá apreciar más fácilmente que sensores se encuentran más cercanos a la pared. Se recomienda que el alumno modifique la posición inicial, orientación así como el movimiento del robot para adquirir manejo con el funcionamiento de los sensores.

5.6.5. TestLightSensor

Este ejemplo muestra el funcionamiento y uso de los sensores de luz. Para ello se ha creado un controlador en el que el robot gira sobre sí mismo en una arena donde existe una fuente de luz amarilla. En cada instante de muestreo el simulador muestra por pantalla el valor de los sensores de luz amarilla. Se puede observar que los 2 sensores más cercanos a la fuente de

luz son los que obtienen valores distintos de cero. Observando la diferencia entre los dos valores se puede extraer la orientación de la luz. Los ficheros asociados al experimento son `testlightexp` y `testlightcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 4
```

En el experimento (`testlightexp.h` y `testlightexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.5,0.5) y con orientación 0 rad. Además se introduce una fuente de luz amarilla en la posición 0.0. El robot incorpora los sensores de luz amarilla y los motores.

Durante la ejecución del ejemplo se recomienda presionar la tecla "l" que mostrará el rango de visión de los sensores de luz y se podrá apreciar más fácilmente que sensores se encuentran más cercanos a la fuente de luz. Se recomienda que el alumno modifique la posición inicial, orientación así como el movimiento del robot para adquirir manejo con el funcionamiento de los sensores. Igualmente se recomienda modificar tanto la posición de la fuente de luz como el rango de los sensores de luz para observar las diferencias de funcionamiento.

5.6.6. TestBlueLightSensor

El objetivo de este ejemplo es idéntico que el anterior pero usando los sensores y fuentes de luz azul. Los ficheros asociados al experimento son `testbluelightexp` y `testbluelightcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 5
```

5.6.7. TestRedLightSensor

El objetivo de este ejemplo es idéntico que el anterior pero usando los sensores y fuentes de luz red. Los ficheros asociados al experimento son `testredlightexp` y `testredlightcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 6
```

5.6.8. TestSwitchLigthSensor

Se ha creado un método que permite a los robots apagar/encender la luz más cercana al robot. Dicho método, `void SwitchNearestLight (int n_value)`, se ha incorporado a la clase de sensores de luz (amarilla, azul y roja), donde `n_value=0` apaga la luz y `n_value=1` la enciende.

De esta manera, definiendo una variable `m_seLight` como:

`CLightSensor* m_seLight = (CLightSensor*) m_pcEpuck->GetSensor(SENSOR_LIGHT);`
podremos encender/apagar la luz amarilla más próxima mediante las instrucciones:

`m_seLight->SwitchNearestLight(1)` y `m_seLight->SwitchNearestLight(0)` respectivamente

Igualmente haríamos uso del actuador para las luces azules:

`CBLueLightSensor* m_seLight = (CBlueLightSensor*) m_pcEpuck->GetSensor(SENSOR_BLUE_LIGHT);` y rojas: `CRedLightSensor* m_seLight = (CRedLightSensor*) m_pcEpuck->GetSensor(SENSOR_RED_LIGHT);`

Téngase en cuenta que el funcionamiento de este método con el de secuencias de encendido no es posible, ya que ambos sistemas pretenderían apagar/encender las luces en momentos distintos, siendo en algunos casos órdenes opuestas.

Para comprobar el funcionamiento del método, se ha creado el experimento 7:

`./irsim -E 7`

Dicho experimento ha sido programado en `experiments/testswitchlightexp` y `controllers/testswitchlightcontroller`. El robot se mueve por el entorno hacia la luz más cercana y cuando el valor del sensor de luz amarilla está por encima de un umbral la apaga.

5.6.9. TestGroundSensor

Este ejemplo muestra el funcionamiento y uso de los sensores de suelo y sensores de memoria de suelo. El robot se inicializa en el centro de la arena y es programado para avanzar en línea recta. En cada instante de muestreo el simulador muestra por pantalla el valor de los sensores de suelo y suelo con memoria. Durante el ejemplo el robot pasará por 2 baldosas, una gris y otra negra. Cuando el robot esté encima de cada una de las baldosas, se observará como se modifica la lectura del sensor de suelo. Igualmente, se puede observar como el paso de una baldosa gris a una negra es memorizado por el sensor de suelo con memoria. Los ficheros asociados al experimento son `testgroundexp` y `testgroundcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 8
```

En el experimento (`testgroundexp.h` y `testgroundexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.0,0.0) y con orientación $\pi/4$ rad. Además se introducen dos baldosas: una gris en la posición (0.1,0.1) de 0.1 m de radio y otra negra en la posición (1.1, 1.1) de 0.5 m de radio.

Se recomienda que el alumno modifique la posición inicial, orientación así como el movimiento del robot para adquirir manejo con el funcionamiento de los sensores. Igualmente se recomienda modificar el número y posición de las baldosas de suelo para observar las diferencias de funcionamiento.

5.6.10. TestBatterySensor

En este ejemplo se muestra el funcionamiento del sensor de batería. En cada instante de muestreo el simulador muestra por pantalla el valor del sensor de batería. Los ficheros asociados al experimento son `testbatteryexp` y `testbatterycontroller`.

Para mostrar el funcionamiento completo del sensor, ha sido necesario realizar una máquina de estados finitos sencilla. Inicialmente el robot se mueve hacia delante hasta que la batería se ve reducida por debajo del 50 %. En ese momento el robot cambia de estado y modifica el sentido de la marcha. Éste permanece en este estado hasta que la batería alcanza el 90 % y entonces el robot vuelve a modificar el sentido de la marcha.

La máquina de estados ha sido implementada de la siguiente manera:

```
if ( m_dStatus == 0 & battery[0] < 0.5 )
        m_dStatus = 1;
else if ( m_dStatus == 1 & battery[0] > 0.9 )
        m_dStatus = 0;

if ( m_dStatus == 0 )
        m_acWheels->SetSpeed(200,200);
        else
        m_acWheels->SetSpeed(-50,-50);
```

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 9
```

En el experimento (`testbatteryexp.h` y `testbatteryexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.0,0.0) y con orientación $\pi/4$ rad. Además se introduce una fuente de luz en la posición (0.25, 0.25).

Los valores de rango, carga y descarga de la batería se han introducido directamente en el fichero del experimento `testbatteryexp.cpp`.

Se recomienda que el alumno modifique la posición inicial, orientación así como el movimiento del robot para adquirir manejo con el funcionamiento del sensor. Igualmente se recomienda modificar el número y posición de las fuentes de luz y los parámetros de la ecuación de carga y descarga de la batería.

5.6.11. TestEncoderSensor

En este ejemplo se muestra el funcionamiento del encoder. En cada instante de muestreo el simulador muestra por pantalla el desplazamiento de cada una de las ruedas en el periodo de muestreo. Además, se incluye un método para imprimir por pantalla la posición real del robot.

Los ficheros asociados al experimento son `testencoderexp` y `testencodercontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 10
```

En el experimento (`testencoderexp.h` y `testencoderexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.0,0.0) y con orientación $\pi/4$ rad.

Se recomienda que el alumno modifique el código para obtener la posición del robot a través de los valores de los encoders.

5.6.12. TestCompassSensor

En este ejemplo se muestra el funcionamiento de la brújula. Además se añade el código del encoder para la comprobación de la posición y orientación del robot. En cada instante de muestreo el simulador muestra por pantalla la orientación del robot. Además, se incluye un método para imprimir por pantalla la posición real del robot.

Los ficheros asociados al experimento son `testcompassexp` y `testcompasscontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 11
```

En el experimento (`testcompassexp.h` y `testcompassexp.cpp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.0,0.0) y con orientación $\pi/4$ rad.

Se recomienda que el alumno modifique el código para obtener la orientación del robot a través de los valores de los encoders.

5.6.13. IriXExp

Como ya se ha comentado anteriormente, se han creado 3 experimentos (`iri1exp`, `iri2exp` y `iri3exp`) para que el alumno pueda implementar sus algoritmos dentro de la asignatura. Estos experimentos se encuentran vacíos, y describimos aquí únicamente las partes necesarias para su comprensión. Los controladores de cada experimento son `iri1controller`, `iri2controller` y `iri3controller`, y los encontramos en el directorio `controllers`. Los experimentos se ejecutan mediante la instrucción:

```
./irsim -E <NUM> -p <PARAM_FILE>
```

Donde `NUM` corresponde a los números 30 (`iri1exp`), 31 (`iri2exp`) y 32 (`iri3exp`). `PARAM_FILE` es el fichero de parámetros.

Se han creado 3 ficheros de parámetros (`iri1Param.txt`, `iri2Param.txt` e `iri3Param.txt`) para que estén disponibles para el alumnos. Estos ficheros se encuentran bajo el subdirectorio `paramFiles`.

Los ficheros siguen la estructura comentada en la Sección 5.3.1.

A modo de ejemplo se muestra el fichero iri3Param.txt que tiene 4 luces amarillas, 1 luz azul y 2 baldosas de suelo (1 gris y 1 negra):

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%% EXTRA %%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NUMBER OF ROBOTS                = 1
ROBOT 1: X POSITION (meters)    = 0.0
ROBOT 1: Y POSITION (meters)    = 1.0
ROBOT 1: ORIENTATION (radians)  = 1.5
WRITE TO FILE (0 No, 1 YES )    = 1

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%% ENVIRONMENT %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NUMBER OF LIGHT OBJECTS                     = 4
X POSITION                                  = -1
Y POSITION                                  = 1
X POSITION                                  = 1
Y POSITION                                  = -1
X POSITION                                  = 1
Y POSITION                                  = 1
X POSITION                                  = -1
Y POSITION                                  = -1
NUMBER OF BLUE LIGHT OBJECTS                = 1
BLUE LIGHT 1: X POSITION                    = 1
BLUE LIGHT 1: Y POSITION                    = 0
NUMBER OF GROUND AREA                       = 2
GROUND AREA 1 X POSITION                    = 0.5
GROUND AREA 1 Y POSITION                    = 0.5
GROUND AREA 1 RADIUS                        = 0.1
GROUND AREA 1 COLOR (0.0 Black, 0.5 Grey)   = 0.5
GROUND AREA 1 X POSITION                    = -0.5
GROUND AREA 1 Y POSITION                    = -0.5
GROUND AREA 1 RADIUS                        = 0.2
GROUND AREA 1 COLOR (0.0 Black, 0.5 Grey)   = 0.0

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%% SENSORS %%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

LIGHT SENSOR RANGE (meters)       = 4.5
BLUE LIGHT SENSOR RANGE (meters)  = 1.0
BATTERY SENSOR RANGE (meters)     = 0.5
```

```
BATTERY CHARGE COEF              = 0.01
BATTERY DISCHARGE COEF           = 0.0001
```

Si ejecutamos el simulador mediante la instrucción:

```
./irsim -E 32 -p paramFiles/iri3Param.txt
```

observamos que el robot avanza en línea recta hasta que se choca con la pared superior. Durante toda la ejecución podemos observar los valores leídos por los sensores.

Es entre la lectura de sensores y la actuación de los motores donde el alumno debe desarrollar su código, ya sea un vehículo de braitenberg o una arquitectura de subsunción como se verá en los próximos capítulos.

Capítulo 6

Vehículos de Braitenberg

6.1. Programación de vehículos de Braitenberg

La arquitectura del controlador de los vehículos de Braitenberg debe ser programada íntegramente, aunque se ha preparado una opción que facilita esta tarea, así como un ejemplo sencillo.

Los ficheros programados son:
`experiments/braitenbergvehicle2exp.h`,
`experiments/braitenbergvehicle2exp.cpp`,
`controllers/braitenbergvehicle2controller.h` y
`controllers/braitenbergvehicle2controller.cpp`.

El ejemplo consiste en lograr que el robot se aproxime a una fuente de luz. Sabemos que esto puede hacerse mediante una conexión directa (o conexión lateral) de los sensores a los motores, y una polaridad de activación positiva (+): a mayor intensidad de luz (I), mayor velocidad del motor (V). Se ha programado una relación V-I lineal en el método `SimulationStep` de la clase `CBraitenbergVehicle2Controller`. Es suficiente con que haya dos sensores, y en el ejemplo se han utilizado los sensores de luz $(0, 7)$.

Los experimentos con la arquitectura de Braitenberg se ejecutan mediante la instrucción:

```
./irsim -E 20
```

En los ficheros
(`experiments/braitenbergvehicle2exp.h` y
`experiments/braitenbergvehicle2exp.cpp`
se crea una arena de $3 \times 3\, m^2$ y se introduce un robot en la posición (0,5, 0,5) y con orientación $4{,}71\, rad$. Además se introduce una fuente de luz en la posición (0, 0).

Los ficheros
`controllers/braitenbergvehicle2controller.h` y
`controllers/braitenbergvehicle2controller.cpp`

definen los elementos necesarios para el funcionamiento de la arquitectura de Braitenberg. La programación de la arquitectura debe hacerse en el método `SimulationStep` de la clase `CBraitenbergVehicle2Controller`. En el ejemplo que está programado, el controlador lee el valor de los sensores de luz, los imprime por pantalla y actualiza la velocidad de los motores en cada instante de muestreo. El código es el siguiente:

```
void CBraitenbergVehicle2Controller::SimulationStep(unsigned n_step_number,
 double f_time, double f_step_interval)
{

        double* light = m_seLight->GetSensorReading(m_pcEpuck);

        printf ("LIGHT: ");
        for ( int i = 0 ; i < m_seLight->GetNumberOfInputs() ; i++)
        {
                printf("%2f " , light[i]);
        }

        printf("\n");
        m_acWheels->SetOutput(0, 0.5 + (light[7] / 2));
        m_acWheels->SetOutput(1, 0.5 + (light[0] / 2));
}
```

Un cambio en la topología de conexión entre sensores y motores (una conexión contralateral o cruzada, por ejemplo), o un cambio en la polaridad de activación $(-)$ provocaría un comportamiento distinto del robot. Estas variedades de robots suponen cambiar las dos últimas líneas del código anterior:

1. Conexión lateral y polaridad positiva

   ```
   m_acWheels->SetOutput(0, 0.5 + (light[7] / 2));
   m_acWheels->SetOutput(1, 0.5 + (light[0] / 2));
   ```

2. Conexión contralateral y polaridad positiva

   ```
   m_acWheels->SetOutput(1, 0.5 + (light[7] / 2));
   m_acWheels->SetOutput(0, 0.5 + (light[0] / 2));
   ```

6.1.1. Ejercicios.

1. Variaciones en cada tipo de vehículo.

   ¿Cómo se comportaría el robot si las funciones de polaridad fuesen distintas para cada vía sensorial?

   ¿Cómo se comportaría si en vez de utilizar los sensores de luz $(0, 7)$ se utilizasen los sensores $(1, 6)$, u otros, manteniendo la conectividad y la polaridad?

2. Tipos de robots.

   Se recomienda hacer ejercicios con conexiones lateral y contralateral y polaridades negativas, así como definir funciones de activación no lineales, continuas y discontinuas. Es decir, programar los Vehículos I a IV según la terminología de Braitenberg.

   ¿Cómo programar Vehículos V? Esto exige implementar **dispositivos de umbral** que conecten sensores con motores. Las conexiones de los sensores a estos dispositivos pueden ser excitadoras o inhibidoras, con polaridades positivas o negativas, con conectividad lateral o contralateral.

3. Efectos del entorno.

   Hay una colección de situaciones que harían cambiar el comportamiento del robot: ¿cómo puede introducirse en el simulador la **fricción** entre el suelo y las ruedas?

4. Diversidad de comportamientos y de trayectorias.

   Cada uno de los vehículos de Braitenberg pretende ser un exponente de una clase o tipo de vehículos caracterizados por una categoría de comportamientos generales o instintos: los Vehículos II, son COBARDES o AGRESIVOS, y así sucesivamente. Sin embargo también pueden ser clasificados por la forma de las trayectorias que pueden realizar en diferentes entornos.

   Un problema interesante es cómo lograr que el robot gire alrededor de una fuente de luz de manera elíptica, estando la luz en uno de los focos de la elipse. O cómo lograr que el robot realice una trayectoria en forma de "ocho" estando la fuente de luz en el centro de uno se sus círculos.

Capítulo 7

Arquitectura Neuronal.

7.1. Programación de la Arquitectura Neuronal.

El diseño del controlador de los robots con una a arquitectura neuronal no requiere apenas programación. La estructura de capas con sus características facilita el diseño enormemente. Es necesario, no obstante su definición concreta en el **fichero de configuración**, así como disponer el valor de los parámetros (pesos y sesgos) en un **fichero de datos**.

La arquitectura neuronal está programada en los ficheros:
`experiments/testneuronexp.cpp`,
`controllers/nndistributedcontroller.cpp` y
`controllers/layercontroller.cpp`.

Cada experimento se ejecuta mediante la instrucción:

```
./irsim -E 21 -p fichero_de_configuracion -c fichero_de_datos
```

como por ejemplo,

```
./irsim -E 21 -p paramFiles/configuracionNeuron_libro.txt
-c paramFiles/datosNeuron_libro
```

La arquitectura del controlador neuronal consiste en un conjunto de capas neuronales interconectadas. Capa capa consta de un conjunto de neuronas no conectadas entre sí que reciben como entradas todas las salidas de la capa o capas precedentes, y posiblemente una entrada sensorial adicional distinta para cada neurona. Aunque cada neurona puede recibir como entrada la salida de un sensor distinto, todas las neuronas de la misma capa reciben entradas sensoriales del mismo tipo de sensor. En la Figura 7.1 puede verse una arquitectura de 5 capas en la que se indica el número de salidas de cada capa y el tipo de sensor de entrada, especificado en el fichero de configuración:
`configuracionNeuron_libro.txt`, con el nombre `SENSOR TO LAYER`.

Se explica a continuación el grupo **NEURAL** del fichero de configuración `configuracionNeuron_libro.txt` para el ejemplo de la figura:

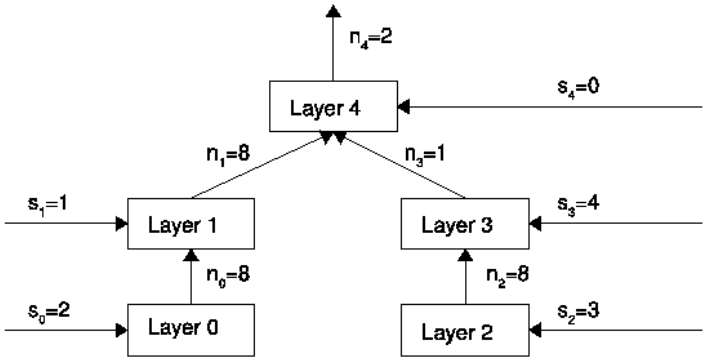


Figura 7.1: Arquitectura neuronal.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% NEURAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

WEIGTH UPPER BOUND                                              = 5.0
WEIGTH LOWER BOUND                                              = -5.0
NUMBER OF LAYERS                                                = 5
SENSOR TO LAYER 0 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 2
SENSOR TO LAYER 1 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 1
SENSOR TO LAYER 2 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 3
SENSOR TO LAYER 3 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 4
SENSOR TO LAYER 4 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 0
ACTIVATION FUNCTION LAYER 0        (0 IDENTITY, 1 SIGMOID,
                                2 STEP, 3 LINEAR, 4 PROGRAM ) = 0
ACTIVATION FUNCTION LAYER 1        (0 IDENTITY, 1 SIGMOID,
                                2 STEP, 3 LINEAR, 4 PROGRAM ) = 1
ACTIVATION FUNCTION LAYER 2        (0 IDENTITY, 1 SIGMOID,
                                2 STEP, 3 LINEAR, 4 PROGRAM ) = 0
ACTIVATION FUNCTION LAYER 3        (0 IDENTITY, 1 SIGMOID,
                                2 STEP, 3 LINEAR, 4 PROGRAM ) = 1
ACTIVATION FUNCTION LAYER 4        (0 IDENTITY, 1 SIGMOID,
                                2 STEP, 3 LINEAR, 4 PROGRAM ) = 1
NUMBER OF OUTPUTs LAYER 0                  = 8
NUMBER OF OUTPUTs LAYER 1                  = 8
NUMBER OF OUTPUTs LAYER 2                  = 8
NUMBER OF OUTPUTs LAYER 3                  = 1
NUMBER OF OUTPUTs LAYER 4                  = 2
RELATION LAYER 0         = 0 1 0 0 0
```

```
RELATION LAYER 1        = 0 0 0 0 1
RELATION LAYER 2        = 0 0 0 1 0
RELATION LAYER 3        = 0 0 0 0 1
RELATION LAYER 4        = 0 0 0 0 0
```

Los pesos serán normalizados a los valores $[-5, 5]$, extremos definidos en `WEIGTH UPPER BOUND = 5.0` y `WEIGTH LOWER BOUND = -5.0`. Se supone que en el fichero de datos figuran con valores en $[0, 1]$.

La línea `NUMBER OF LAYERS= 5` indica el número de capas. A continuación habrá cuatro bloques obligatorios (`SENSOR TO LAYER`, `ACTIVATION FUNCTION`, `NUMBER OF OUTPUTS` y `RELATION LAYER`.

Además se supone que en cada **CAPA SENSORIAL** hay tantos sensores del mismo tipo como neuronas, sensores distribuidos en el robot de acuerdo con su morfología sensorial que debe haberse definido en el grupo `MORPHOLOGY`. Cada neurona de este tipo de capa recibe la información de un único sensor. Por ejemplo, la capa 0 tiene 8 neuronas, por lo que habrá 8 sensores distintos del mismo tipo, y en este caso sensores de proximidad, como se indica en el fichero de configuración con `SENSOR TO LAYER 0=2`. La capa 1 también es una capa sensorial en el sentido de que, además de recibir como entradas las salidas de la capa 0, recibe como entradas la lectura de los sensores de contacto (`SENSOR TO LAYER 1=1`). El robot dispondrá, por lo tanto, de 8 sensores de contacto distintos.

La capa 2 es la capa asociada a los sensores de luz, y la capa 3 asociada al sensor de batería.

Se supone además que hay una conectividad completa entre capas conectadas, es decir, cada neurona de una capa recibe como entradas las salidas de todas las neuronas de la capa o capas inferiores. Por ejemplo cada neurona de la capa 4 recibe 9 entradas, 8 de la capa 1 y 1 de la capa 3. Además esta capa no tiene entradas sensoriales, como se indica en el fichero de configuración haciendo `SENSOR TO LAYER 4 =0`.

En este experimento la capa 4 tiene dos salidas que serán las señales de actuación (velocidades) de cada una de las ruedas del robot. La denominaremos **CAPA MOTORA**.

Las capas 0 y 2 no son estrictamente capas neuronales ya que solo reciben entradas sensoriales, y sus salidas son sus mismas entradas, si bien se distribuyen a todas las neuronas de la capa siguiente (1 y 3 respectivamente). Por esto se las llama capas `IDENTITY`. Se han incluido como capas con el fin de facilitar la programación de la arquitectura. Esta característica se indica en el fichero de configuración haciendo `ACTIVATION FUNCTION LAYER 0=0` y `ACTIVATION FUNCTION LAYER 2=0`.

La arquitectura neuronal puede verse como un grafo dirigido cuya matriz de adyacencia se recoge en el fichero de configuración en el bloque `RELATION LAYER`. En esta matriz se indica con un 1 si una capa está conectada a otra, y con un 0 si no lo está. Por ejemplo la capa 0 se conecta a la capa 1, y a

ninguna otra. Y la capa 4 no se conecta a ninguna, como puede verse en la Figura 7.1.

7.1.1. La función de activación

Todas las neuronas de la misma capa llevan asociada la misma función de activación indicada en `ACTIVATION FUNCTION LAYER` con un número distinto de cero. Están programados tres tipos de funciones de activación, sigmoidal, escalón y lineal, pero también se ha abierto la posibilidad de que el usuario programe su propia función de activación, utilizando la opción `ACTIVATION FUNCTION LAYER =4`. La programación se hace en el fichero `layercontroller.cpp`. El código programado es el siguiente

```
switch ( m_unActivationFunction)
{
        /* If IDENTITY, no bias nothing to do */
        case IDENTITY_ACTIVATION:
        break;

        /* If SIGMOID, use bias and calc sigmoid */
    case SIGMOID_ACTIVATION:
                /* Add scaled BIAS */
           m_pfOutputs[i] += pf_weights[i * (m_unNumberOfLayerInputs + 1)] *
                        (m_fUpperBounds - m_fLowerBounds) + m_fLowerBounds;
                /* Calc Sigmoid */
           m_pfOutputs[i] = 1.0/( 1 + exp( - m_pfOutputs[i]) );
        break;

        /* IF STEP, use not scaled BIAS as a THRESHOLD */
    case STEP_ACTIVATION:
                /* If output bigger than THRESHOLD output 1 */
        if ( m_pfOutputs[i] > pf_weights[i * (m_unNumberOfLayerInputs + 1)] )
                m_pfOutputs[i] = 1.0;
                /* If not, output 0 */
                else
                m_pfOutputs[i] = 0.0;
    break;

        /* If LINEAR, do not use BIAS and create y=1-x, function */
    case LINEAR_ACTIVATION:
                m_pfOutputs[i] = 1 - m_pfOutputs[i];
    break;

        /* If PROGRAM, create your own equation */

    case PROGRAM_ACTIVATION:
     /* YOU NEED TO PROGRAM HERE YOUR EQUATION */
    break;
}
```

7.1.2. El fichero de datos o de pesos y sesgos

Los datos del fichero `paramFiles/datosNeuron_libro` (al cual se le puede dar cualquier otro nombre y estar en cualquier directorio) pueden introducirse por cualquier procedimiento (utilizando el programa evolutivo o manualmente, por ejemplo), pero debe tener la estructura indicada a continuación:

```
101 0.09369266279198995573 0.40639694584081498263 0.00000000000000000000
1.00000000000000000000 0.00000000000000000000 0.42489314511716447242
0.57381162838016175343 0.83240847586976940420 0.81314648554946744596
0.00000000000000000000 0.36031304284303977692 0.90208182319033791696
0.00000000000000000000 0.11685669225009889804 0.41918835112966518208
0.06791464054260859529 0.29340173221489401767 0.98632874299503647819
1.00000000000000000000 0.96270329697152534631 0.04799630205669536132
0.53662923993648237175 0.24288492093475291811 0.55742333889141237879
0.07203409800330307089 0.16710939385936207646 0.00000000000000000000
0.26242571240679629652 0.80574341675848570876 0.08159470022367511233
0.00000000000000000000 0.65263310576748045921 0.00000000000000000000
0.99868420362113197175 0.76351010496735405297 0.34155914629854955411
0.80401720317782710001 0.57485593124675304910 0.81721424196286407415
0.52301147666819347570 0.76639038263455905309 0.64817485213594172588
0.57035139218145991524 1.00000000000000000000 1.00000000000000000000
0.57297800309203605895 0.30877945892790431559 0.17709020164130798158
0.51074109165824099765 1.00000000000000000000 1.00000000000000000000
0.70340874095123440135 0.77129203052380745920 0.11791494586028042346
0.59757259912505755750 0.91345977540849587761 0.35159820590081708458
0.51876414488816990911 1.00000000000000000000 0.82604955271742863676
0.34742807635000710897 0.74410423761001076581 0.47208842155658919948
0.62150129388212482784 1.00000000000000000000 0.29286149610887063366
0.78885196036225180283 0.33393245984950925553 0.23309376379589263895
0.05140556171567216281 0.11377280603504143219 0.14653467002784908990
0.83290725153798661484 0.34237342744452542442 1.00000000000000000000
1.00000000000000000000 0.80164133112330548947 0.41477594467811396139
0.52805168997422569088 0.00000000000000000000 0.08335983997613173602
0.72016446001363498830 0.11949296924069030545 0.40107832582031838209
0.87489214125246839160 0.96279151497762693879 0.42550424580718698708
0.99474854453198569004 0.46060267135270205330 1.00000000000000000000
0.31098919323066220866 1.00000000000000000000 0.26417062851847100680
0.84288871626141126381 0.04266083571031273336 0.30284966729112788864
0.92913383837333207715 0.81106555491437548611 0.91498372191518806407
0.74291951187035720761 1.00000000000000000000
```

Aunque los datos de este fichero puedan introducirse mediante cualquier procedimiento, está pensado para constituir la salida del programa evolutivo. Por esta razón el número `101` representa la longitud del cromosoma (`CHROMOSOME LENGTH` definido en el fichero de configuración en el grupo `GENETIC`).

Los restantes números representan los parámetros de la arquitectura neuronal, ordenados neurona a neurona y capa a capa en orden ascendente. En este ejemplo la capa 1 tiene 8 neuronas, cada una de ellas con 8 entradas. El

primer número es el sesgo de la primera neurona, y los 8 siguientes números representan sus pesos.

Por ejemplo, el primer número **0.09369266279198995573** es el sesgo de la primera neurona de la capa 1 y los ocho siguientes números sus pesos, que deberán ser valores en $[0, 1]$:

```
0.40639694584081498263 0.00000000000000000000 1.00000000000000000000
0.00000000000000000000 0.42489314511716447242 0.57381162838016175343
0.83240847586976940420 0.81314648554946744596
```

El programa neuronal se encarga de normalizar los pesos a los límites fijados en el fichero de configuración como se ha dicho antes.

7.1.3. Cálculo de la longitud del cromosoma, para una sintonización de pesos mediante un proceso evolutivo

La configuración de esta opción se explica en el Capítulo 8.

Para calcular la longitud del cromosoma debe contarse el número de pesos y sesgos de la arquitectura neuronal. En la Figura 7.1 puede verse que la capa 1 tiene 8 neuronas con 8 entradas cada una (además de la sensorial que no debe considerarse en la longitud del cromosoma). Por lo tanto cada neurona de esta capa tendrá 9 parámetros, los 8 pesos y el sesgo de la función de activación. Así la capa 1 tiene 72 parámetros, la capa 3 tiene 9 parámetros y la capa 4 tiene 20 parámetros, que en total suman 101.

7.1.4. Escritura estado neuronal

El controlador **nndistributedcontroller.cpp** incorpora una implementación para escribir en ficheros de texto:

```
/*Write a file for each layer with the input, weights and output */
char fileName[100];
sprintf(fileName, "outputFiles/layer%dOutput", k);
FILE* fileOutput = fopen(fileName, "a");

/* Print TIME */
fprintf(fileOutput,"%2.4f ", m_fTime);
/* Print INPUTS */
for ( int j = 0 ; j < m_unNumberOfSensorInputs[k] ; j++)
{
        fprintf(fileOutput,"%2.4f ",pfSensorInputs[j]);
}
/* Print WEIGHTS */
for ( int j = 0 ; j < m_unNumberOfParameters[k] ; j++ )
{
        fprintf(fileOutput,"%2.4f ",m_pfWeightMatrix[k][j]);
}
/* Print OUTPUTS */
for ( int j = 0 ; j < m_unNumberOfLayerOutputs[k] ; j++)
```

```
{
        fprintf(fileOutput,"%2.4f ",m_fOutputMatrix[k][j]);
}
fprintf(fileOutput,"\n");
fclose(fileOutput);
```

Observamos que para cada capa se crea un fichero:
`outputFiles/layer<X>Output`
donde `<X>` representa el número de la capa, en el cual se imprime el tiempo, las entradas sensoriales de las capas, los pesos y sesgos y las salidas.

Igualmente se crea un ficheros en los que se escribe la posición y orientación del robot `robotPosition` y otro fichero que escribe la velocidad de las ruedas `robotWheels`.

```
/* Write robot position and orientation */
FILE* filePosition = fopen("outputFiles/robotPosition", "a");
fprintf(filePosition,"%2.4f %2.4f %2.4f %2.4f\n",
        m_fTime, m_pcEpuck->GetPosition().x,
        m_pcEpuck->GetPosition().y, m_pcEpuck->GetRotation());
fclose(filePosition);

/* Write robot wheels speed */
FILE* fileWheels = fopen("outputFiles/robotWheels", "a");
fprintf(fileWheels,"%2.4f %2.4f %2.4f \n", m_fTime,
        m_fOutputMatrix[(m_unNumberOfLayers -1 )][0],
        m_fOutputMatrix[(m_unNumberOfLayers -1 )][1]);
fclose(fileWheels);
```

Capítulo 8

Robótica evolutiva

8.1. Introducción

En este Capítulo nos centramos en las explicaciones de la implementación de robótica evolutiva en el simulador IRSIM. Como se ha visto en la teoría vamos a hacer uso de redes neuronales y algoritmos genéticos. La estructura de la redes neuronales dentro del simulador ya han sido explicadas en el Capítulo 7, por lo que en este Capítulo nos centramos en los algoritmos genéticos. Posteriormente introduciremos unos ejemplos para facilitar la comprensión al alumno.

Es necesario tener en cuenta que durante el periodo de evolución, la posición y orientación inicial del robot en la arena es aleatorizada, de tal manera que no se tiene en cuenta la entrada en el fichero de parámetros. Esto se realiza para evitar que el proceso evolutivo converja a una solución óptima para unas condiciones iniciales determinas, pero que pueda ser subóptima en el resto de condiciones iniciales. Posteriormente, en el proceso de evaluación, sí se tienen en cuenta la posición y orientación definidas por el usuario.

8.2. Fichero de configuración

Recuérdese que el algoritmo genético es el encargado de sintonizar los parámetros de la red neuronal para cumplir los objetivos marcados por el diseñador. De tal manera, es necesario configurar en el fichero de configuración cuales son los operadores genéticos así como los tiempos de ejecución, características de la población y los individuos.

El grupo necesario incluir para la ejecución de un algoritmo evolutivo es `GENETIC`. Este grupo tiene la siguiente estructura:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% GENETIC %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
CHROMOSOME LENGTH                        = entero
POPULATION SIZE                          = entero
NUMBER OF GENERATIONS                    = entero
EVALUATION TIME                          = entero
DO CROSSOVER ( 0 No, 1 Yes )             = binario
NUMBER OF CROSSOVERS ( Always 1 )        = 1
CROSSOVER DISTANCE   ( Always 1 )        = 1
MUTATION RATE                            = real
NUMBER OF ELITES                         = entero
FITNESS FUNCTION                         = entero
```

El parámetro `CHROMOSOME LENGTH` define la longitud del cromosoma. Como se ha explicado en el Capítulo 7, es necesario conocer la arquitectura neuronal para facilitar este parámetro al fichero. `POPULATION SIZE` define el número de individuos en una generación. `NUMBER OF GENERATIONS` define el número de generaciones máximas que se van a ejecutar. En `EVALUATION TIME` describimos el tiempo en el que cada individuo (cromosoma) va a ser evaluado. `DO CROSSOVER` define si se va a realizar la operación de reproducción sexual o cruce. Por otro lado `NUMBER OF CROSSOVERS` y `CROSSOVER DISTANCE` definen el número de cruces y el mínimo número de genes en cada parte que se van a utilizar. Actualmente en el simulador sólo es posible obtener un punto de cruce donde al menos un único gen debe ser cruzado. `MUTATION RATE` define la probabilidad de mutación para cada gen de cada cromosoma excluyendo a la elite. `NUMBER OF ELITES` representa el número de los mejores individuos que pasan a la siguiente generación idénticamente (sin cruces ni mutaciones). Por último, `FITNESS FUNCTION` define la función de fitness a utilizar.

La clave para el buen funcionamiento del proceso evolutivo está en definir una función de fitness adecuada al problema. Estas funciones se encuentran definidas como clases en el directorio `fitnessfunctions`. Actualmente hemos desarrollado 3 funciones de fitness: `avoidcollisionsfitnessfunction`, `garbagefitnessfunction` y `loadfitnessfunction`, mas una cuarta para que el alumno realice sus experimentos `irifitnessfunction.cpp`. Cada una de estas funciones tiene asociado un número definido en el fichero `main.cpp`:

```
#define FITNESSFUNCTION_AVOIDCOLLISIONS 1
#define FITNESSFUNCTION_GARBAGE 2
#define FITNESSFUNCTION_LOAD 3
#define FITNESSFUNCTION_IRI 4
```

Explicaremos las tres primeras funciones en los ejemplos que se detallan en este Capítulo.

8.3. Ejecución

Para comprobar el funcionamiento del experimento es necesario dividir la ejecución del simulador en dos partes. Una primera parte será la encargada

de correr el algoritmo genético obteniendo los diferentes cromosomas que maximizan la función del fitness. Posteriormente será necesario ejecutar el simulador con el cromosoma resultante, o que interese al usuario, como si se tratase de una ejecución neuronal cualquiera.

8.3.1. Evolución

Para comenzar la evolución del cromosoma necesitamos introducir el parámetro "-e", de tal manera que en función del `FICHERO_DE_CONFIGURACION` tendremos:

```
./irsim -E 21 -p <FICHERO_DE_CONFIGURACION> -e
```

Una vez ejecutado dicho comando el simulador arranca, genera una población aleatoria de `POPULATION SIZE` individuos y los va evaluando. El simulador se ejecuta en modo texto (no hay visualización gráfica) y nos irá escribiendo en pantalla el estado de las generaciones, por ejemplo:

```
Creating random generation...
Generation 0 - best fitness: 0.818005, average: 0.574981, worst: 0.213513
Generation 1 - best fitness: 0.827454, average: 0.575191, worst: 0.216646
Generation 2 - best fitness: 0.812638, average: 0.598370, worst: 0.229405
Generation 3 - best fitness: 0.809978, average: 0.636251, worst: 0.231453
Generation 4 - best fitness: 0.805802, average: 0.648463, worst: 0.305943
```

Observamos que el simulador nos muestra cual ha sido el mejor, medio y peor individuo de acuerdo con la función de fitness definida.

Durante esta ejecución el simulador se encarga de guardar cierta información en el directorio `geneticDataFiles`. En ese directorio encontramos:

- `fitness.log` es un fichero dedicado al histórico de las generaciones. Se almacena lo mismo que se ha ido mostrando por pantalla.
- `generationX.log` almacena los `POPULATION SIZE` cromosomas de cada una de las generaciones. Con motivo de no almacenar demasiados datos en el disco duro, únicamente se almacenan las generaciones múltiplos de 10, así como la primera y la última.
- `bestX.log` almacena el mejor individuo (de acuerdo a la fitness) de cada una de las generaciones.
- `currentbest` almacena el mejor individuo de todas las generaciones hasta el momento.
- `maxgeneration` almacena el número de la generación actual. Se utiliza a nivel informativo para saber cual ha sido la última generación en ejecución, y para poder reiniciar una evolución a partir de un cromosoma

dado. Para esta última aplicación es necesario modificar en el fichero `main.cpp` la variable `g_bRestartEvolution` y asignarla a `true`. En caso contrario, por defecto, se reiniciará siempre desde la generación 0.

8.3.2. Evaluación

Una vez evolucionadas todas la generaciones o alcanzado un máximo de valor de fitness podremos visualizar cual ha sido el resultado obtenido por el algoritmo genético. Para ello ejecutaremos:

```
./irsim -E 21 -p <FICHERO_DE_CONFIGURACION>
```

Esta ejecución coge el fichero `geneticDataFile/currentbest` lo carga en la red neuronal y ejecuta la simulación. En caso de querer evaluar un cromosoma distinto al mejor, será necesario suministrárselo por linea de comando mediante el comando "-c":

```
./irsim -E 21 -p <FICHERO_DE_CONFIGURACION> -c <CROMOSOMA>
```

En las siguientes secciones nos centramos en experimentos concretos que complementan las explicaciones anteriores.

8.4. Ejemplos

8.4.1. NeuronEvoAvoidExp

Es este experimento se programa una arquitectura neuronal cuyos pesos y sesgos (o umbrales) son sintonizados utilizando un proceso evolutivo. El experimento pretende sintonizar una red neuronal mediante la que el robot sea capaz de esquivar obstáculos en el entorno maximizando la velocidad de movimiento. Los ficheros asociados al experimento son `testneuronexp.cpp`, `nndistributedcontroller.cpp` y `layercontroller.cpp`, los mismos para todas la arquitecturas neuronales. El fichero de configuración se encuentra en `paramFiles/paramFileNeuroEvoAvoid.txt`.

La arquitectura neuronal a utilizar se muestra en la Figura 8.1. Observamos una estructura sencilla de 2 capas neuronales. Una de entrada que recibe información de los sensores de proximidad (8 neuronas de entrada) y una de salida conectada a los motores (2 neuronas de salida). La capa de entrada lleva asociada una función de activación identidad mientras que la de salida una sigmoide.

Para la descripción del experimento nos vamos a centrar en los grupos `GENETIC` y `NEURAL` del fichero de configuración utilizado. Comenzamos explicando el grupo neuronal:

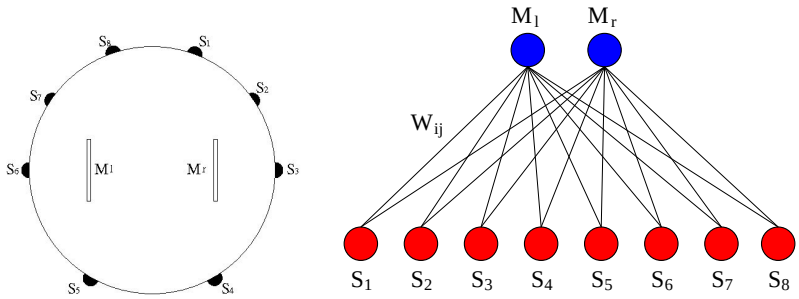


Figura 8.1: Arquitectura neuronal del experimento NeuronEvoAvoidExp

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%% NEURAL %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

WEIGTH UPPER BOUND  = 5.0
WEIGTH LOWER BOUND  = -5.0
NUMBER OF LAYERS    = 2
SENSOR TO LAYER 0 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 2
SENSOR TO LAYER 1 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 0
ACTIVATION FUNCTION LAYER 0 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 0
ACTIVATION FUNCTION LAYER 1 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 1
NUMBER OF OUTPUTs LAYER 0     = 8
NUMBER OF OUTPUTs LAYER 1     = 2
RELATION LAYER 0  = 0 1
RELATION LAYER 1  = 0 0
```

Como se observa en el código anterior, los pesos son normalizados entre $[-5, 5]$ y tenemos 2 capas. La Capa 0 lleva asociada los sensores de proximidad, mientras que la Capa 1 no tiene entradas sensoriales. La Capa 0 mantiene una función de activación identidad y dispone de 8 salidas, correspondientes a los 8 sensores de infrarrojos [1]. La Capa 1 es definida por una función de activación sigmoide y 2 neuronas de salida. La relación entre las capas, como ya se ha visto en la Figura 8.1, es una conexión completa entre la Capa 0 y la Capa 1.

Los parámetros del algoritmo genético se han definido como sigue:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% GENETIC %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

[1] Nótese que en el grupo `MORPHOLOGY` se han definido todos los sensores de infrarrojo a 1

```
CHROMOSOME LENGTH                = 18
POPULATION SIZE                  = 100
NUMBER OF GENERATIONS            = 10000
EVALUATION TIME                  = 100
DO CROSSOVER ( 0 No, 1 Yes )     = 1
NUMBER OF CROSSOVERS ( Always 1 ) = 1
CROSSOVER DISTANCE   ( Always 1 ) = 1
MUTATION RATE                    = 0.05
NUMBER OF ELITES                 = 5
FITNESS FUNCTION                 = 3
```

El grupo `GENETIC` define 18 genes correspondientes a los pesos de 8 neuronas de entrada por 2 de salida (16 pesos) más los 2 sesgos de las neuronas de salida. Se define una población de 100 individuos y un máximo de 10.000 generaciones. Cada individuo es evaluado durante 100 periodos de muestreos (10 segundos). Se acepta el operador de crossover con un único punto de cruce y un mínimo de 1 gen por parte. La probabilidad de mutación de cada gen es de 0.05 y se han definido 5 elites por generación.

Finalmente la función de fitness seleccionada ha sido la 3, correspondiendo al código existente en `avoidfitnessfunction.cpp`. Esta fitness implementa la ecuación $fit = V \cdot (1 - \sqrt{\Delta v}) \cdot (1 - i)$ definida en la teoría.

Primero normalizamos las velocidades de la rueda derecha e izquierda entre 0 y 1.

```
double leftSpeed = 0.0;
double rightSpeed = 0.0;

m_pcEpuck->GetWheelSpeed(&leftSpeed,&rightSpeed);
leftSpeed   = 0.5 + ( leftSpeed  / ( 2.0 *  m_pcEpuck->GetMaxWheelSpeed()) );
rightSpeed  = 0.5 + ( rightSpeed / ( 2.0 *  m_pcEpuck->GetMaxWheelSpeed()) );
```

Posteriormente evaluamos las diferentes partes de la ecuación. Comenzamos con V correspondiente a la maximización de la velocidad en cada rueda:

```
/* Eval maximum speed partial fitness */
double maxSpeedEval = (fabs(leftSpeed - 0.5) + fabs(rightSpeed - 0.5));
```

Continuamos con $(1 - \sqrt{\Delta v})$, correspondiente a la maximización en el giro de ambas ruedas en el mismo sentido:

```
/* Eval same direction partial fitness */
double sameDirectionEval = 1 - sqrt(fabs(leftSpeed - rightSpeed));
```

Por último obtenemos el máximo valor leído de los sensores de proximidad:

```
/* Eval minimum sensor reading partial fitness */
double maxProxSensorEval = 0;
double maxLightSensorEval = 0;

TSensorVector vecSensors = m_pcEpuck->GetSensors();
for (TSensorIterator i = vecSensors.begin(); i != vecSensors.end(); i++)
{
  if ( (*i)->GetType() == SENSOR_PROXIMITY)
  {
    unsigned int unThisSensorsNumberOfInputs = (*i)->GetNumberOfInputs();
    double* pfThisSensorInputs = (*i)->GetComputedSensorReadings();

    for (int j = 0; j < unThisSensorsNumberOfInputs; j++)
    {
      if ( pfThisSensorInputs[j] > maxProxSensorEval )
      {
        maxProxSensorEval = pfThisSensorInputs[j];
      }
    }
  }
}

maxProxSensorEval = 1 - maxProxSensorEval;
```

Calculamos la fitness mediante el siguiente código:

```
/* Max Speed * Same Direction * Min Prox * go forwards */
double fitness = maxSpeedEval * sameDirectionEval *
                 maxProxSensorEval * (leftSpeed * rightSpeed);
m_fComputedFitness += fitness;
```

Nótese que se ha incluido un término `leftSpeed * rightSpeed` para maximizar el movimiento del robot en línea recta.

Por último comentar que esta fitness es evaluada en cada instante de muestreo (`SimulationStep`), por lo que es necesario calcular el valor promedio al final del experimento (`GetFitness`) mediante el siguiente código:

```
double fit = m_fComputedFitness / (double) m_unNumberOfSteps;
```

Una vez comprendido el funcionamiento del fichero de configuración y la función de fitness, podemos poner a evolucionar el simulador:

```
./irsim -E 21 -p paramFiles/paramFileNeuronEvoAvoid.txt -e
```

Tras unas cuantas generaciones observamos que la fitness comienza a superar valores de 0.9. Ahora podemos parar la simulación y ejecutar el código con el mejor cromosoma obtenido:

```
./irsim -E 21 -p paramFiles/paramFileNeuronEvoAvoid.txt
```

Finalmente observar que si definimos en el fichero de configuración `WRITE TO FILE` a 1, escribiremos en el directorio `outputFiles` la salida de cada una de las capas neuronales en el tiempo, así como las velocidades impuestas a las ruedas y la posición del robot en cada instante de tiempo.

8.4.2. NeuronEvoLoadExp

En este experimento pretendemos que el robot navege por el entorno a la vez que mantiene su batería lo más cargada posible. Para ello se ha programado una arquitectura neuronal cuyos pesos y sesgos (o umbrales) son sintonizados utilizando un proceso evolutivo. El fichero de configuración asociado al experimento es `paramFileNeuronEvoLoad.txt`.

El entorno incorpora una luz amarilla que será usada por el robot para cargar la batería:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% ENVIRONMENT %%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

NUMBER OF LIGHT OBJECTS       = 1
LIGHT 1 X POSITION            = -1
LIGHT 1 Y POSITION            = -1
NUMBER OF BLUE LIGHT OBJECTS  = 0
NUMBER OF GROUND AREA         = 0
```

La arquitectura neuronal viene definida por el siguiente código:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%% NEURAL %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

WEIGTH UPPER BOUND  = 5.0
WEIGTH LOWER BOUND  = -5.0
NUMBER OF LAYERS    = 4
SENSOR TO LAYER 0 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 2
SENSOR TO LAYER 1 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 4
SENSOR TO LAYER 2 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 3
SENSOR TO LAYER 3 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 0
ACTIVATION FUNCTION LAYER 0 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                    3 LINEAR, 4 PROGRAM )  = 0
ACTIVATION FUNCTION LAYER 1 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                    3 LINEAR, 4 PROGRAM )  = 0
ACTIVATION FUNCTION LAYER 2 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                    3 LINEAR, 4 PROGRAM )  = 0
ACTIVATION FUNCTION LAYER 3 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                    3 LINEAR, 4 PROGRAM )  = 1
NUMBER OF OUTPUTs LAYER 0     = 8
NUMBER OF OUTPUTs LAYER 1     = 1
NUMBER OF OUTPUTs LAYER 1     = 8
NUMBER OF OUTPUTs LAYER 2     = 2
RELATION LAYER 0  = 0 0 0 1
RELATION LAYER 1  = 0 0 0 1
RELATION LAYER 2  = 0 0 0 1
RELATION LAYER 3  = 0 0 0 0
```

Observamos que tenemos 3 capas sensoriales con entradas de los sensores de proximidad, batería y luz. Todas estas capas están conectadas directamente a la capa motora.

El algoritmo genético viene caracterizado por los siguientes parámetros:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% GENETIC %%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CHROMOSOME LENGTH                = 36
POPULATION SIZE                  = 100
NUMBER OF GENERATIONS            = 10000
EVALUATION TIME                  = 300
DO CROSSOVER ( 0 No, 1 Yes )     = 1
NUMBER OF CROSSOVERS ( Always 1 ) = 1
CROSSOVER DISTANCE   ( Always 1 ) = 1
MUTATION RATE                    = 0.05
NUMBER OF ELITES                 = 5
FITNESS FUNCTION                 = 3
```

La función de fitnesss 3 (`loadfitnessfunction.cpp`) se ha diseñado en relación a 3 comportamientos:

- Maximizar la velocidad de movimiento en línea recta. Al igual que en el ejemplo anterior.
- Maximizar la carga de la batería
- Maximizar el valor de los sensores de luz

A continuación describimos cada uno de estos comportamientos codificados en la función de fitness. La maximización de la velocidad de movimiento se ha realizado igual que en el ejemplo anterior:

```
maxSpeedEval * sameDirectionEval * (1 - maxProxSensorEval) *
  (leftSpeed * rightSpeed )
```

La carga de batería es evaluada mediante el estado instantáneo del sensor:

```
battery[0]
```

Finalmente, el máximo del sensor de luz es añadido a la ecuación:

```
case SENSOR_LIGHT:
unThisSensorsNumberOfInputs = (*i)->GetNumberOfInputs();
pfThisSensorInputs = (*i)->GetComputedSensorReadings();

for (int j = 0; j < unThisSensorsNumberOfInputs; j++)
{
        if ( pfThisSensorInputs[j] > maxLightSensorEval )
```

```
        {
                maxLightSensorEval = pfThisSensorInputs[j];
        }
}
```

De esta manera la fitness evaluada en cada instante de tiempo viene representada por la siguiente ecuación:

```
fitness = coef1 * ( maxSpeedEval *
                    sameDirectionEval *
                    (1 - maxProxSensorEval) *
                    (leftSpeed * rightSpeed ) ) +
          coef2 * ( battery[0] ) +
          (1 - coef1 - coef2 ) * ( maxLightSensorEval );
```

donde `coef1=0.5` y `coef2=0.3`.

Para la evaluación completa del experimento se tiene en cuenta el número de colisiones que el robot ha tenido durante el tiempo de evaluación. Permitimos un máximo de 10 colisiones, a partir de las cuales el individuo recibe una fitness 0.0:

```
int coll = (CCollisionManager::GetInstance()->GetTotalNumberOfCollisions());
double fit = ((m_fComputedFitness / (double) m_unNumberOfSteps) *
              (1 - ((double) (fmin(coll,10.0)/10.0))));
if ( fit < 0.0 ) fit = 0.0;
```

Al igual que en todos los experimentos evolutivos, es necesario realizar la evolución primero

```
./irsim -E 21 -p paramFiles/paramFileNeuronEvoLoad.txt -e
```

y posteriormente la evaluación

```
./irsim -E 21 -p paramFiles/paramFileNeuronEvoLoad.txt
```

8.4.3. NeuronEvoGroundExp

En este experimento pretendemos que el robot navege por el entorno, localice "basura" (baldosas grises) y las deposite en un "vertedero" (baldosas negras). Para ello se ha programado una arquitectura neuronal cuyos pesos y sesgos (o umbrales) son sintonizados utilizando un proceso evolutivo. El fichero de configuración del experimento es `paramFileNeuronEvoGround.txt`.

El entorno incorpora una baldosa negra simulando un vertedero, con una luz amarilla en su centro para conseguir dirigir el robot hacia el mismo. Además existe 3 zonas de basura dispersas por la arena:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% ENVIRONMENT %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

```
NUMBER OF LIGHT OBJECTS                     = 1
LIGHT 1 X POSITION                          = 1.5
LIGHT 1 Y POSITION                          = 1.5
NUMBER OF BLUE LIGHT OBJECTS                = 0
NUMBER OF GROUND AREA                       = 4
GROUND AREA 1 X POSITION                    = 1.3
GROUND AREA 1 Y POSITION                    = 1.3
GROUND AREA 1 RADIUS                        = 0.5
GROUND AREA 1 COLOR (0.0 Black, 0.5 Grey) = 0.0
GROUND AREA 2 X POSITION                    = -1
GROUND AREA 2 Y POSITION                    = -1
GROUND AREA 2 RADIUS                        = 0.2
GROUND AREA 2 COLOR (0.0 Black, 0.5 Grey) = 0.5
GROUND AREA 3 X POSITION                    = 1
GROUND AREA 3 Y POSITION                    = -1
GROUND AREA 3 RADIUS                        = 0.2
GROUND AREA 3 COLOR (0.0 Black, 0.5 Grey) = 0.5
GROUND AREA 4 X POSITION                    = -1
GROUND AREA 4 Y POSITION                    = 1
GROUND AREA 4 RADIUS                        = 0.2
GROUND AREA 4 COLOR (0.0 Black, 0.5 Grey) = 0.5
```

La arquitectura neuronal viene definida por el siguiente código:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% NEURAL %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

WEIGTH UPPER BOUND            = 5.0
WEIGTH LOWER BOUND            = -5.0
NUMBER OF LAYERS              = 5
SENSOR TO LAYER 0 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 2
SENSOR TO LAYER 1 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 1
SENSOR TO LAYER 2 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 3
SENSOR TO LAYER 3 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 5
SENSOR TO LAYER 4 ( NONE 0, CONTACT 1, PROX 2, LIGHT 3, BATTERY 4,
                    GROUND_MEMORY 5, GROUND 6, BLUE LIGHT 7) = 0
ACTIVATION FUNCTION LAYER 0 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 0
ACTIVATION FUNCTION LAYER 1 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 1
ACTIVATION FUNCTION LAYER 2 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 0
ACTIVATION FUNCTION LAYER 3 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 1
ACTIVATION FUNCTION LAYER 4 ( 0 IDENTITY, 1 SIGMOID, 2 STEP,
                              3 LINEAR, 4 PROGRAM )  = 1
NUMBER OF OUTPUTs LAYER 0     = 8
NUMBER OF OUTPUTs LAYER 1     = 8
```

```
NUMBER OF OUTPUTs LAYER 2     = 8
NUMBER OF OUTPUTs LAYER 3     = 1
NUMBER OF OUTPUTs LAYER 4     = 2
RELATION LAYER 0  = 0 1 0 0 0
RELATION LAYER 1  = 0 0 0 0 1
RELATION LAYER 2  = 0 0 0 1 0
RELATION LAYER 3  = 0 0 0 0 1
RELATION LAYER 4  = 0 0 0 0 0
```

Obsérvese que en este ejemplo hacemos uso de 5 capas neuronales. Dos de ellas (Capa 0 y Capa 2) son capas sensoriales, otras dos son asociativas (Capa 1 y Capa 3) y la última (Capa 4) es la capa motora.

Debido a la distribución de las capas necesitamos un cromosoma de 101 genes tal y como se aprecia en el grupo `GENETIC`:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%% GENETIC %%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

CHROMOSOME LENGTH                = 101
POPULATION SIZE                  = 100
NUMBER OF GENERATIONS            = 10000
EVALUATION TIME                  = 300
DO CROSSOVER ( 0 No, 1 Yes )     = 1
NUMBER OF CROSSOVERS ( Always 1 ) = 1
CROSSOVER DISTANCE   ( Always 1 ) = 1
MUTATION RATE                    = 0.05
NUMBER OF ELITES                 = 5
FITNESS FUNCTION                 = 2
```

La función de fitness `2` se ha definido en `garbagefitnessfunction.cpp`. La función de fitness se encuentra dividida en dos estados, uno de búsqueda y otro de depósito. Para mantener visibles los estados se ha creado la variable global `m_unState` en el fichero `garbagefitnessfunction.h`. Esta variable se encarga de mantener el estado y en función del mismo el individuo es evaluado.

Cuando el robot se encuentra en el estado búsqueda (`SEARCH`), la función de fitness se maximiza con respecto al movimiento en línea recta y la máxima velocidad del robot. Si el robot encuentra una zona de comida se le incrementa la fitness instantánea en 10 y se modifica el estado.

```
case SEARCH:
        fitness = maxSpeedEval * sameDirectionEval *
                  maxProxSensorEval * (leftSpeed * rightSpeed);
        if ( groundMemory[0] == 1.0 )
        {
                fitness += 10;
                m_unState = DEPOSIT;
```

```
        }
        break;
```

Una vez en el estado de depósito (`DEPOSIT`) la fitness es directamente proporcional a la medida del sensor de luz. En caso de que el robot alcance el vertedero, se incrementa la fitness instantánea en 10 y se vuelve al estado de búsqueda.

```
case DEPOSIT:
        fitness = maxLightSensorEval;
        if ( groundMemory[0] = 0.0 )
        {
                fitness += 10;
                m_unState = SEARCH;
        }
        break;
```

Al igual que en los ejemplos anteriores la fitness del individuo es la media de las fitness en cada instante de muestreo.

Capítulo 9

Arquitecturas Subsunción

9.1. Introducción

En este capítulo desarrollamos el funcionamiento de los ejemplos expuestos en teoría de la Arquitectura Subsunción en el simulador IRSIM. Se han implementado dos ejemplos:

- *SubsumptionLightExp*: El robot navega por el entorno y vuelve a la zona de carga en cuanto detecta un nivel de batería inferior a un umbral.
- *SubsumptionGarbageExp*: El robot navega por el entorno intentando localizar "zonas de basura" (baldosas grises) y una vez localizadas debe encontrar una "vertedero" (baldosas negras) donde depositarlas. Una vez depositada, vuelve a buscar baldosas grises. Se mantiene el comportamiento de carga de batería.

9.2. Ejemplos

9.2.1. SubsumptionLightExp

Este experimento utiliza los motores, sensores de luz, sensores de proximidad y el sensor de batería. Los ficheros asociados al experimento son `subsumptionlightexp` y `subsumptionlightcontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 22 -p paramFiles/paramFileSubsumptionLight.txt
```

En el experimento (`subsumptionlightexp`) se crea una arena de 3x3 m^2 y se introduce un robot en la posición (1.0,1.0) y con orientación 0.0 rad. Además se introduce una luz en la posición (-1, -1) y unos obstáculos.

La arquitectura programada se muestra en la Figura 9.1. Hay que tener en cuenta que estamos trabajando en una máquina secuencial. Por lo tanto

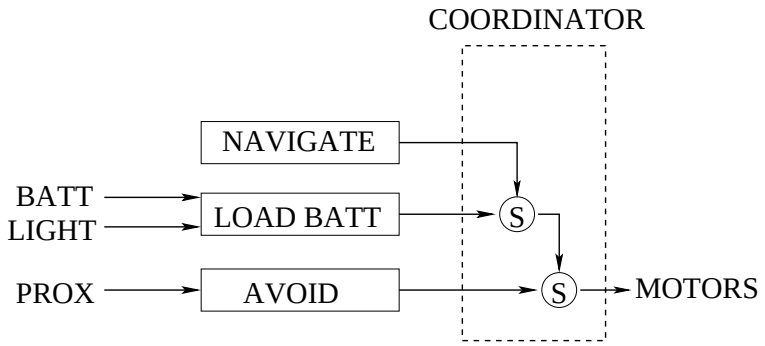


Figura 9.1: Arquitectura subsunción del experimento `subsumptionLightExp`.

los diferentes niveles de competencia no se ejecutan en paralelo. Igualmente, al ser una implementación software es necesario crear un coordinador que se encargue de suprimir/inhibir las tareas.

El coordinador se basa en una tabla que representa los valores de las salidas de cada nivel de competencia así como un flag de activación (ver Figura 9.2). Cada nivel de competencia se ejecuta y pone su velocidad a los motores en la fila correspondiente. Si el nivel de competencia va a suprimir a otros niveles pone su flag a `TRUE`.

	SpeedLeft	SpeedRight	Flag
Level 0	V_l^0	V_r^0	T/F
Level 1	V_l^1	V_r^1	T/F
Level 2	V_l^2	V_r^2	T/F

Figura 9.2: Tabla del coordinador del experimento `subsumptionLightExp`.

Una vez que se han ejecutado todos los niveles de competencia, el coordinador recorre la tabla desde el nivel más bajo al más alto. En cuanto encuentra un flag activado, recoge los valores de las velocidades y las pasa a los actuadores. A continuación se muestra el código del coordinador:

```
int nBehavior;
for ( nBehavior = 0 ; nBehavior < BEHAVIORS ; nBehavior++ )
{
        if ( m_fActivationTable[nBehavior][2] == 1.0 )
```

```
        {
                break;
        }
}

m_fLeftSpeed = m_fActivationTable[nBehavior][0];
m_fRightSpeed = m_fActivationTable[nBehavior][1];
```

Como se ha indicado anteriormente, el controlador se compone de 3 niveles de competencia:

- **Navigate**: Este nivel de competencia pone la señal SPEED en ambos motores, haciendo que el robot se mueva en línea recta.
- **Load Battery**: Este nivel de competencia se encarga de leer la señal de la batería. Si el sensor de la batería es inferior a 0.5 el robot suprime la salida de *Navigate* y hace girar el robot hasta orientarse en el sentido de la luz.
- **Avoid**: Este nivel de competencia es el encargado de hacer que el robot evite obstáculos. El nivel de competencia comprueba si hay un obstáculo delante del robot. En caso afirmativo, suprime la señal que viene de los otros niveles de competencia, calcula un vector en dirección opuesta al obstáculo que será enviado a los motores.

A continuación detallamos como se ha construido cada nivel de competencia.

Navigate:

```
void CSubsumptionLightController::Navigate ( unsigned int un_priority )
{
        m_fActivationTable[un_priority][0] = SPEED;
        m_fActivationTable[un_priority][1] = SPEED;
        m_fActivationTable[un_priority][2] = 1.0;
}
```

Como se observa en el código el nivel de competencia pone una velocidad definida (`SPEED`) en ambas ruedas forzando un movimiento en línea recta. Igualmente modifica el flag de activación de su nivel de competencia.

Load Battery:

```
/* Read  Battery Sensores */
double* battery = m_seBattery->GetSensorReading(m_pcEpuck);

if ( battery[0] < BATTERY_THRESHOLD )
{
  /* Set Leds to RED */
  m_pcEpuck->SetAllColoredLeds( LED_COLOR_RED);
```

```
  /* Leer Sensores de Luz */
  double* light = m_seLight->GetSensorReading(m_pcEpuck);

  /* If not pointing to the light */
  if ( ( light[0] * light[7] == 0.0 ) )
  {
    m_fActivationTable[un_priority][2] = 1.0;

    double lightLeft  = light[0] + light[1] + light[2] + light[3];
    double lightRight = light[4] + light[5] + light[6] + light[7];

    if ( lightLeft > lightRight )
    {
      m_fActivationTable[un_priority][0] = -SPEED;
      m_fActivationTable[un_priority][1] = SPEED;
    }
    else
    {
      m_fActivationTable[un_priority][0] = SPEED;
      m_fActivationTable[un_priority][1] = -SPEED;
    }
  }
}
```

Observamos que el nivel de competencia comprueba si la batería está por debajo de un determinado umbral (`BATTERY_THRESHOLD`).

En caso negativo, sale de la ejecución sin modificar el flag de activación.

En caso afirmativo, pone el color del robot a rojo (simplemente para visualización), mediante la siguiente instrucción:

```
m_pcEpuck->SetAllColoredLeds ( LED_COLOR_RED );
```

Además, lee los sensores de luz y comprueba si $L_0 * L_7 \neq 0$, donde L_0 es el sensor de luz a la izquierda de la dirección de movimiento del robot y L_7 el de la derecha.

En caso negativo, sale de la ejecución sin modificar el flag de activación, ya que el robot se encuentra orientado hacia la luz.

En caso afirmativo, se activa el flag del nivel de competencia.

Además, el robot obtiene la suma de los valores de los sensores de su izquierda $\sum_{i=0}^{3} L_i$ y su derecha $\sum_{i=4}^{7} L_i$. Si la luz se encuentra a la izquierda del robot éste girará a izquierda, y si se encuentra a la derecha girará a la derecha.

Avoid:

```
/* Leer Sensores de Proximidad */
double* prox = m_seProx->GetSensorReading(m_pcEpuck);
```

```
double fMaxProx = 0.0;
const double* proxDirections = m_seProx->GetSensorDirections();

dVector2 vRepelent;
vRepelent.x = 0.0;
vRepelent.y = 0.0;

/* Calc vector Sum */
for ( int i = 0 ; i < m_seProx->GetNumberOfInputs() ; i ++ )
{
        vRepelent.x += prox[i] * cos ( proxDirections[i] );
        vRepelent.y += prox[i] * sin ( proxDirections[i] );

        if ( prox[i] > fMaxProx )
                fMaxProx = prox[i];
}

/* If above a threshold */
if ( fMaxProx > PROXIMITY_THRESHOLD )
{
        /* Set Leds to GREEN */
        m_pcEpuck->SetAllColoredLeds( LED_COLOR_GREEN);

        /* Calc pointing angle */
        float fRepelent = atan2(vRepelent.y, vRepelent.x);
        /* Create repelent angle */
        fRepelent -= M_PI;
        /* Normalize angle */
        while ( fRepelent > M_PI ) fRepelent -= 2 * M_PI;
        while ( fRepelent < -M_PI ) fRepelent += 2 * M_PI;

        double fCLinear = 1.0;
        double fCAngular = 1.0;
        double fC1 = SPEED / M_PI;

        /* Calc Linear Speed */
        double fVLinear = SPEED * fCLinear * ( cos ( fRepelent / 2) );

        /*Calc Angular Speed */
        double fVAngular = fRepelent;

        m_fActivationTable[un_priority][0] = fVLinear + fC1 * fVAngular;
        m_fActivationTable[un_priority][1] = fVLinear - fC1 * fVAngular;
        m_fActivationTable[un_priority][2] = 1.0;
}
```

El nivel de competencia lee los sensores de proximidad, calcula el máximo y genera un vector opuesto a la dirección del obstáculo. Si el máximo de los sensores se encuentra por encima de un umbral (`PROXIMITY_THRESHOLD`), se considera que existe un obstáculo.

Además se pone el robot en color verde (sólo para visualización), y se

calcula una velocidad lineal y angular en función del vector obtenido anteriormente. A partir de esos valores se calcula las velocidades de la rueda derecha e izquierda y se activa el flag del nivel de competencia.

Por último vamos a describir las diferentes escrituras en fichero que se han llevado a cabo para la post-evaluación de la arquitectura.

En cada instante de muestreo escribiremos todos los datos que a continuación presentamos.

En el método `SimulationStep`, escribimos en dos ficheros: `robotPosition` y `robotWheels`. Para el primero utilizamos las siguientes instrucciones:

```
/* Write robot position and orientation */
FILE* filePosition = fopen("outputFiles/robotPosition", "a");
fprintf(filePosition,"%2.4f %2.4f %2.4f %2.4f\n",
        m_fTime, m_pcEpuck->GetPosition().x,
        m_pcEpuck->GetPosition().y, m_pcEpuck->GetRotation());
fclose(filePosition);
```

De tal forma escribimos en el fichero `outputFiles/robotPosition` el tiempo, posición y orientación del robot. Igualmente escribimos la velocidad de las ruedas en el fichero `outputFiles/robotWheels` mediante las siguientes instrucciones:

```
/* Write robot wheels speed */
FILE* fileWheels = fopen("outputFiles/robotWheels", "a");
fprintf(fileWheels,"%2.4f %2.4f %2.4f \n",
        m_fTime, m_fLeftSpeed, m_fRightSpeed);
fclose(fileWheels);
```

Posteriormente hacemos que cada vez que se ejecute el coordinador escriba, el tiempo, nivel de competencia que ejecuta y las velocidades de las ruedas:

```
/* Write coordinator ouputs */
FILE* fileOutput = fopen("outputFiles/coordinatorOutput", "a");
fprintf(fileOutput,"%2.4f %d %2.4f %2.4f \n",
        m_fTime, nBehavior, m_fLeftSpeed, m_fRightSpeed);
fclose(fileOutput);
```

Por último cada tarea imprime en un fichero sus valores sensoriales, si se activa o no la tarea y las velocidades propuestas para los motores:

En la tarea `Avoid` tenemos:

```
/* Write level of competence ouputs */
FILE* fileOutput = fopen("outputFiles/avoidOutput", "a");
fprintf(fileOutput, "%2.4f %2.4f %2.4f %2.4f %2.4f %2.4f
        %2.4f %2.4f %2.4f %2.4f %2.4f ", m_fTime,
        prox[0], prox[1], prox[2], prox[3], prox[4],
        prox[5], prox[6], prox[7], fMaxProx, fRepelent);
fprintf(fileOutput, "%2.4f %2.4f %2.4f\n",
```

```
        m_fActivationTable[un_priority][2],
        m_fActivationTable[un_priority][0],
        m_fActivationTable[un_priority][1]);
fclose(fileOutput);
```

En la tarea `LoadBattery`:

```
/* Write level of competence ouputs */
FILE* fileOutput = fopen("outputFiles/batteryOutput", "a");
fprintf(fileOutput, "%2.4f %2.4f %2.4f %2.4f %2.4f %2.4f
        %2.4f %2.4f %2.4f %2.4f ", m_fTime, battery[0],
        light[0], light[1], light[2], light[3],
        light[4], light[5], light[6], light[7]);
fprintf(fileOutput, "%2.4f %2.4f %2.4f\n",
        m_fActivationTable[un_priority][2],
        m_fActivationTable[un_priority][0],
        m_fActivationTable[un_priority][1]);
fclose(fileOutput);
```

Finalmente, en la tarea `Navigate` tenemos:

```
/* Write level of competence ouputs */
FILE* fileOutput = fopen("outputFiles/navigateOutput", "a");
fprintf(fileOutput,"%2.4f %2.4f %2.4f %2.4f \n", m_fTime,
        m_fActivationTable[un_priority][2],
        m_fActivationTable[un_priority][0],
        m_fActivationTable[un_priority][1]);
fclose(fileOutput);
```

Por lo tanto al final del experimento tendremos los ficheros: `avoidOutput`, `batteryOutput`, `coordinatorOutput`, `navigateOutput`, `robotPosition` y `robotWheels`. A modo de ejemplo mostramos un resumen del fichero `avoidOutput` escrito por la tarea `Avoid`

```
0.0000 0.0000 0.0000 [...] 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 [...] 0.0000 0.0000 0.0000
0.0000 0.0000 0.0000 [...] 0.0000 0.0000 0.0000
[...]
[...]
[...]
19.1000 0.2696 0.1636 [...] 0.0000 0.0000 0.0000
19.2000 0.2818 0.1856 [...] 0.0000 0.0000 0.0000
19.4000 0.3101 0.2186 [...] 1.0000 500.9366 -495.7818
19.5000 0.2825 0.1099 [...] 1.0000 546.7495 -283.9763
```

Observamos que para cada fila.

- Columna 1: Tiempo.
- Columnas 2-9: Sensores de Infrarrojos.

- Columna 10: Valor máximo de los sensores.
- Columna 11: Ángulo de evitación de obstáculo.
- Columna 12: Flag de activación de la tarea.
- Columnas 13-14: Velocidad de las ruedas (left, right).

9.2.2. SubsumptionGarbageExp

Este experimento utiliza los motores, sensores de luz, sensores de proximidad, sensor de suelo de memoria y el sensor de batería. Los ficheros asociados son `subsumptiongarbageexp` y `subsumptiongarbagecontroller`.

El experimento se ejecuta mediante la instrucción:

```
./irsim -E 23 -p paramFiles/paramFileSubsumptionGarbage.txt
```

En el experimento (`subsumptiongarbageexp` se crea una arena de 3x3 m^2 y se introduce un robot en la posición (0.0,0.0) y con orientación 0.0 rad. Gracias al fichero `paramFileSubsumptionGarbage.txt`, introducimos una luz en la posición (-1.5, -1.5), una baldosa negra en la posición (1.5, 1.5) de radio 0.5 y cuatro baldosas grises. La baldosa negra representa un vertedero y las grises basura. Si el robot pasa por una baldosa gris se activará el sensor de suelo memoria, representando que se ha recogido basura del entorno. Si el robot pasa por la baldosa negra, se desactivará el sensor de suelo memoria, representando que se ha dejado la basura en el vertedero.

Este experimento implementa una arquitectura subsunción con 4 niveles de competencia como se muestra en la Figura 9.3.

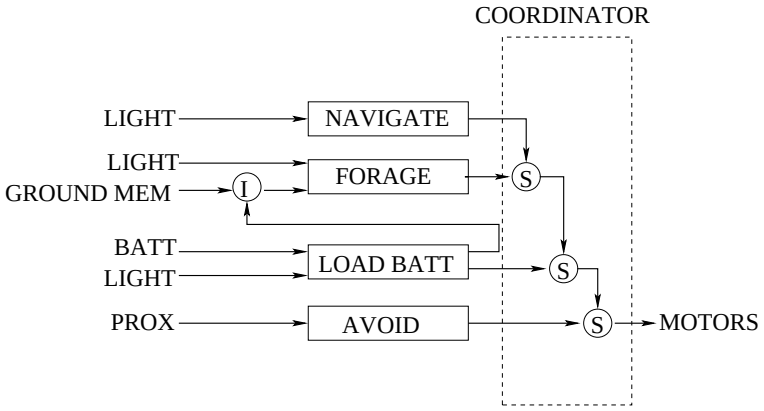


Figura 9.3: Arquitectura subsunción del experimento `subsumptionGarbageExp`.

El coordinador sigue la misma estructura que en el ejemplo anterior, con la diferencia que ahora existen 4 niveles de competencia.

Se han modificado algunos niveles de competencia para adecuarse a la arquitectura actual. A continuación detallamos cada uno de ellos.

Avoid:
Este nivel de competencia es exactamente igual que en el ejemplo anterior.

Load Battery:
Este nivel de competencia se ha modificado ligeramente, incorporando una salida que será la encargada de inhibir o no, la señal del nivel FORAGE con la siguiente instrucción.

```
fBattToForageInhibitor = 0.0;
```

Esta instrucción se ejecuta en el caso de que el nivel de batería esté por debajo del umbral definido.

Forage:
Este nuevo nivel de competencia sigue una estructura similar a la del nivel `LoadBattery`:

```
/* Leer Sensores de Suelo Memory */
double* groundMemory = m_seGroundMemory->GetSensorReading(m_pcEpuck);

/* If with a virtual puck */
if ( ( groundMemory[0] * fBattToForageInhibitor ) == 1.0 )
{
        /* Set Leds to BLUE */
        m_pcEpuck->SetAllColoredLeds( LED_COLOR_BLUE);
        /* Leer Sensores de Luz */
        double* light = m_seLight->GetSensorReading(m_pcEpuck);

        /* Go oposite to the light */
        if ( ( light[3] * light[4] == 0.0 ) )
        {
                m_fActivationTable[un_priority][2] = 1.0;

                double lightLeft = light[0] + light[1] + light[2] + light[3];
                double lightRight= light[4] + light[5] + light[6] + light[7];

                if ( lightLeft > lightRight )
                {
                        m_fActivationTable[un_priority][0] = SPEED;
                        m_fActivationTable[un_priority][1] = -SPEED;
                }
                else
                {
                        m_fActivationTable[un_priority][0] = -SPEED;
                        m_fActivationTable[un_priority][1] = SPEED;
                }
```

```
        }
}
```

El nivel de competencia lee el estado del sensor de memoria de suelo multiplicado por la variable del inhibidor. En caso de estar activo, significando que ha pasado por una zona gris y por lo tanto tiene un elemento virtual que transportar al vertedero, nos lo señaliza con el color azul. En este caso, y dado la distribución de la arena, intenta orientarse de tal manera que deje la fuente de luz en la parte trasera: $L_3 * L_4 \neq 0$. En caso de que ya esté orientado no suprime ninguna tarea, ya que será el nivel `NAVIGATE` quien se encargue de llevar al robot en sentido contrario a la luz. En el caso en el que el robot no esté orientado en el sentido opuesto a la luz, el nivel de competencia se encarga de hacer rotar al robot siguiendo los mismos principios que en el nivel `LOAD BATTERY`.

Navigate:

Por último se ha modificado este nivel de competencia para conseguir un mayor aprovechamiento de la carga de la batería.

```
/* Leer Sensores de Luz */
double* light = m_seLight->GetSensorReading(m_pcEpuck);
double fTotalLight = 0.0;
for ( int i = 0 ; i < m_seLight->GetNumberOfInputs() ; i ++ )
{
        fTotalLight += light[i];
}

if ( fTotalLight >= NAVIGATE_LIGHT_THRESHOLD )
{
        m_fActivationTable[un_priority][0] = SPEED/4;
        m_fActivationTable[un_priority][1] = SPEED/4;
}
else
{
        m_fActivationTable[un_priority][0] = SPEED;
        m_fActivationTable[un_priority][1] = SPEED;
}

m_fActivationTable[un_priority][2] = 1.0;
```

El nivel lee los sensores de luz y calcula la suma de todos ellos ($\sum_{i=0}^{7} L_i$). Si el sumatorio es mayor a un umbral el robot reduce su velocidad a `SPEED/4`. De esta forma conseguimos que si el robot se encuentra cerca de la luz reduzca su velocidad y consiga cargarse durante más tiempo.