industriales
etsii

Escuela Técnica Superior de Ingeniería Industrial

UNIVERSIDAD POLITÉCNICA DE CARTAGENA

Escuela Técnica Superior de Ingeniería Industrial

Diseño de un robot móvil de servicio con capacidades de interacción natural con humanos.

TRABAJO FIN DE GRADO

GRADO EN INGENIERÍA ELECTRÓNICA Y AUTOMÁTICA

Autor: Roberto Oterino Bono
Directora: Dra. Nieves Pavón Pulido
Codirector: Dr. Jorge Juan Feliú Batlle

Cartagena, a 06 de abril de 2021

Universidad Politécnica de Cartagena

Agradecimientos

A mis padres y a mi hermano que siempre me han apoyado por muchos errores que haya cometido. Y a mi abuela y a mi abuelo que nos dejó hace poco luchando hasta el final.

RESUMEN

La Robótica de Servicios está sufriendo un auge en los últimos años. Sin embargo, sigue manteniendo dos problemas principalmente: el coste económico y, en el caso de la Robótica Social la interacción con las personas. No todo el mundo sabe programar un robot, y la idea no es que todo el mundo sepa. Cada vez se tiene que facilitar más la comunicación entre una persona y un robot.

Un claro ejemplo es el robot *Pepper,* un robot humanoide, programable y diseñado para interactuar con personas. Su tecnología le permite detectar tanto el lenguaje verbal como el no verbal. Sin embargo su precio no es nada asequible, ronda los 20.000€.

Actualmente hay diferentes tecnologías que permiten diseñar un robot reduciendo su coste. Por un lado han aumentado los servicios que proporcionan diferentes empresas como *Amazon*, *Google* y *Microsoft* entre otros, en el contexto de soluciones en la Nube. Entre estos servicios se encuentran la detección facial, el reconocimiento de voz, la interpretación del lenguaje, la síntesis de voz… Sin embargo, presentan limitaciones entre las que cabe destacar: no suelen ser flexibles y se requiere una conexión a internet para poder usarlos.

Por otro lado, existen modelos de inteligencia artificial, tales como los de aprendizaje automático o aprendizaje profundo, que se pueden desplegar en prácticamente cualquier dispositivo. Existen varias bibliotecas que permiten trabajar con ellos como *Caffe*, *PyTorch*, pero las más importantes son *scikit-learn* y *TensorFlow*. Dado que solventa el problema de la conexión, no suponen ningún coste y permiten flexibilidad, se ha optado por usar esta tecnología para desarrollar los diferentes módulos del sistema .

Respecto a la navegación autónoma, existen diferentes algoritmos que permiten al robot alcanzar un objetivo y evitar obstáculos. El framework *ROS* (*Robotic Operating System)* proporciona un paquete que permite configurar varios de estos algoritmos de una forma sencilla. Esta facilidad ha determinado que se seleccione dicho framework, para este trabajo.

Por último queda el problema de la integración. Se debe encontrar un lazo de unión entre todos los componentes y que estén completamente sincronizados. De nuevo se ha recurrido a *ROS* que entre uno de sus modos de comunicación se encuentra el de cliente-servidor que permite coordinar los diferentes componentes del sistema, incluso si están en lenguajes de programación diferentes.

Ahora bien, en la diferentes pruebas realizadas se han encontrado una serie de limitaciones provenientes de los modelos de reconocimiento de voz e interpretación de lenguaje natural ya que suelen estar desarrollados para el inglés y no tanto para otros idiomas como el castellano. Estas limitaciones, así como las ventajas del sistema, se demuestran tras realizar una serie de pruebas controladas y analizar los resultados obtenidos.

Índice

Anexos

Figuras

Tablas

Fragmentos de código

1 Introducción.

En este capítulo, se presenta el problema que se va a tratar, relacionado con el ámbito de la Robótica de Servicios. También se describen los principales problemas que suelen estar relacionados con el coste de fabricación y la comunicación con el robot, y los objetivos que se quieren alcanzar para diseñar un robot móvil autónomo con el que los usuarios se puedan comunicar de forma natural. Por último, también resume el contenido de cada capítulo de forma independiente.

1.1 Motivación y objetivos.

La Robótica de Servicios está sufriendo un auge tecnológico impulsado por las innovaciones de Aprendizaje Automático, en inglés *Machile Learning (ML),* la Inteligencia Artificial, los sistemas de Visión Artificial, entre otro [1]. Los segmentos que más están creciendo en Robótica de Servicio que más están creciendo son: logístico, médico y robótica de campo.

El principal problema es la barrera económica que supone obtener un diseño funcional. Sin embargo, gracias al avance de la tecnología se podría minimizar el coste alquilando las capacidades de computación gracias a los servicios que presenta la Nube (*Cloud Services*) o utilizando modelos de aprendizaje profundo ligeros, fácilmente desplegables en dispositivos móviles.

Por otro lado, existe el problema de la comunicación. La comunicación no verbal es la manera más utilizada entre robots y humano mediante interfaces de usuarios típicas (pantallas táctiles, metáforas de escritorio, etc) [2]. Sin embargo, existe una tendencia para conseguir que un robot tenga la capacidad de comprender el lenguaje natural del ser humano. Y, por tanto, avanzar hacia una interacción natural con los humanos.

Con este proyecto se quiere romper tanto la barrera económica como el problema de la comunicación. No todo el mundo sabe programar un robot, y la idea no es que todo el mundo sepa. Cada vez se tiene que facilitar más la comunicación entre una persona y un robot.

El objetivo principal, por tanto, será diseñar un robot con navegación autónoma que permita comunicarse de forma natural con las personas. Para lograr este objetivo es necesario alcanzar una serie de subjetivos:

- Implementar la navegación autónoma. El robot debe de ser capaz de generar una trayectoria para alcanzar un destino y ser capaz de evitar obstáculos.
- Definir un sistema que permita detectar y reconocer personas.

- Desarrollar un módulo que permita comunicarse de forma sencilla con el robot y que éste sepa interpretarlo.
- Integrar todos los componentes, de forma que estén sincronizados.

1.2 Resumen de capítulos.

- Capítulo 1: Introducción:
  En este capítulo se ha introducido el tema de la robótica de servicios y se han analizado los problemas existentes. Así mismo, se han descrito los objetivos que se quieren alcanzar para diseñar un robot móvil autónomo con el que se pueda comunicar de forma natural.
- Capítulo 2: Estado del arte:
  En este apartado se define que es un robot de servicios y se mencionan tres robots de este tipo. También se describen las diferentes técnicas de navegación autónoma y se exponen las distintas formas que tiene un robot para interaccionar con una persona.
- Capítulo 3: Diseño del sistema:
  En esta sección, se explica la estructura del sistema. La arquitectura hardware se define de forma breve porque se ha usado el simulador *Gazebo*. La arquitectura software se desarrolla con más detalle. Por un lado, se presenta la estructura de los modelos de Inteligencia Artificial y como implementarlos. Por otro lado, se introduce el paquete de navegación de ROS y se configura. Por último, se explica cómo se han integrado todos los componentes.
- Capítulo 4: Análisis de los resultados:
  En este apartado se explican las pruebas realizadas y los resultados obtenidos. Además, se realiza un análisis de los beneficios y limitaciones del sistema.
- Capítulo 5: Conclusiones y trabajos futuros:
  En este capítulo se detallan las conclusiones obtenidas y se describen las posibles líneas de desarrollo que se pueden realizar a partir de este trabajo.

2 Estado del arte.

En este apartado, se definirá el concepto de Robótica de Servicios, y se mencionarán tres robots que presentan similitudes respecto al sistema que se quiere diseñar. Además, se describirán las diferentes técnicas de navegación autónoma disponibles, distinguiendo entre la planificación local que permite evitar los obstáculos del entorno y la planificación global cuyo objetivo es planificar la ruta para alcanzar un objetivo.

Por otro lado, se expondrán las diferentes formas que tiene un robot para interaccionar con una persona. En primer lugar, se explicarán las diferentes técnicas que se utilizan para el reconocimiento de personas y en que se basan. En segundo lugar los modelos existentes de reconocimiento de voz y en que se fundamentan. Seguidamente, se analizarán diferentes sintetizadores de voz. Por último, se describirán las funciones principales del reconocimiento de Lenguaje Natural y se nombrarán una serie de herramientas que disponen de esas funciones.

2.1 Introducción a la Robótica de Servicios.

La Federación Internacional de Robótica (IFR) ha propuesto una definición de robot de servicio: "es un robot que opera semi o totalmente autónomo para realizar servicios útiles para el bienestar de los seres humanos y equipos, con exclusión de las operaciones de fabricación" [3]. Los robots sociales son un subconjunto de los robots de servicios.

Uno de los grandes retos es conseguir diseñar un robot autónomo que sea capaz de interactuar con las personas sin ser tele operado por los humanos. Es un desafío porque el robot debe interpretar correctamente la comunicación, verbal o no verbal, con el humano y responder adecuadamente.

2.1.1 Pearl

*Pearl (*ver Fig. 1) es un robot móvil asistencial destinado a personas de avanzada edad [4]. Tiene dos funciones principalmente, la de recordar las actividades diarias esenciales como son comer, beber e ingerir alimentos; y la función de guiarlas dentro de su entorno.

A nivel de hardware posee dos ordenadores, una conexión vía *WiFi*, una pantalla táctil, altavoces y un micrófono. Además, tiene sensores (sonar y láser) para detectar objetos, y un sistema de cámara estéreo. Estos elementos permiten que tenga un sistema de navegación autónoma. En su conjunto el sistema puede capturar vídeo, realizar reconocimiento facial y seguimiento, adaptándose incluso a la velocidad de la persona.

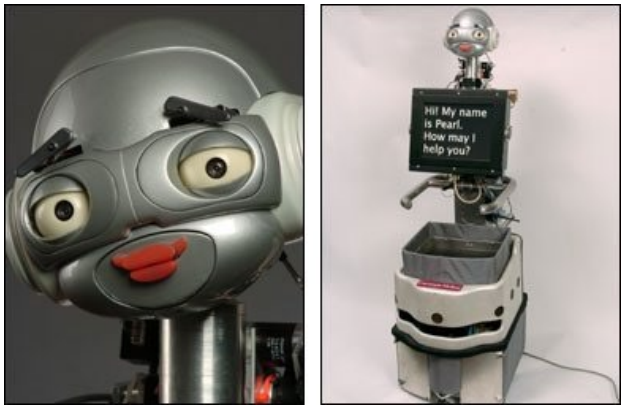

Fig. 1 Robot *Pearl*

2.1.2 Pepper

Pepper es un robot humanoide, programable y diseñado para interactuar con personas. Su tecnología le permite detectar tanto el lenguaje verbal como el no verbal, la posición de la cabeza y el tono de voz, para reconocer el estado emocional e individualizar cada interacción [5]. Presenta una gran gama de sensores como se ve en la Fig. 2, permitiéndole percibir su entorno, para que pueda realizar las tareas de la mejor forma posible.

La interacción es la funcionalidad clave del robot *Pepper*. Presenta una interfaz multimodal que incluye una pantalla táctil, sistema de voz, diodos emisores de luz y manos y cabezas táctiles. La estructura se diseñó con 17 articulaciones, de este modo se logra una mayor expresividad a través del lenguaje corporal.

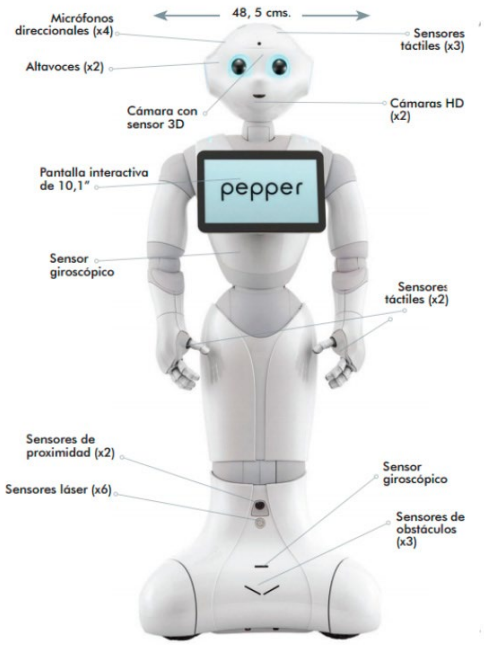

Fig. 2 Robot *Pepper*

2.1.3 TERESA

Teresa (*TElepresence REinforcement-learning Social Agent*) es un robot social semiautomático de tele presencia que está diseñado para ayudar a personas con algún problema de movilidad o de edad avanzada [6]. El objetivo que busca es conseguir que la persona pueda asistir a eventos sociales sin necesidad de estar presente.

El robot (ver Fig. 3), se controla a través de un ordenador, con el cual, gracias a una cámara, podrá retransmitir su imagen en la pantalla del robot. Éste, por su parte, también dispone de otra cámara que reproducirá la imagen a tiempo real en el ordenador, pudiendo entonces realizar una interacción humano-humano. Una característica importante es que puede adaptar su altura para que el interlocutor siempre esté de cara.

Por último, hay que destacar que es capaz de interaccionar con su entorno, debido a que incorpora un sistema de navegación que evita los obstáculos y permite trazar rutas.

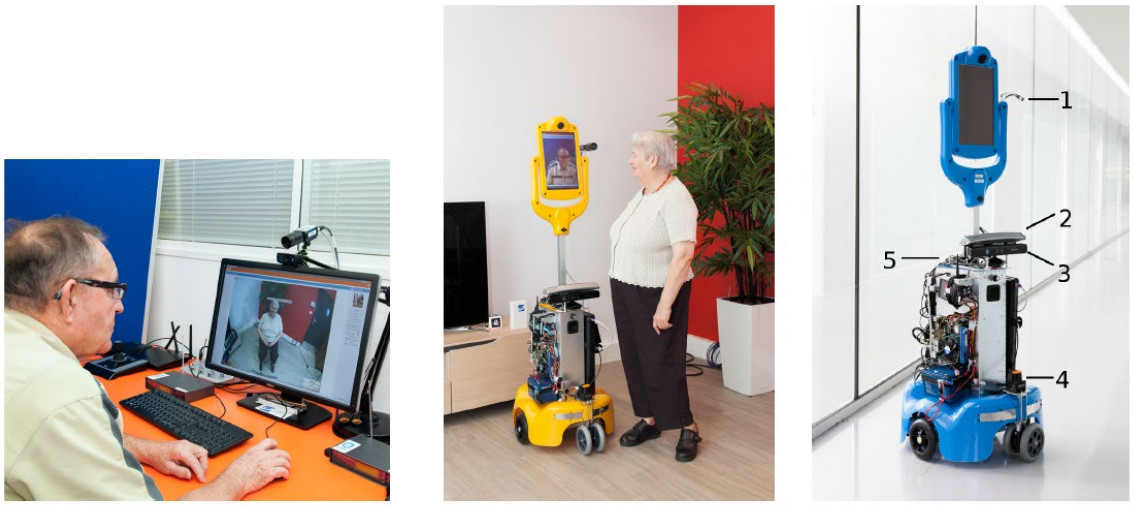

Fig. 3 Robot TERESA

2.2 Navegación autónoma.

La navegación autónoma es uno de los retos que se plantea en el diseño de los robots móviles autónomos. El robot debe poder localizarse en el entorno según un mapa y calcular una trayectoria que permita alcanzar su destino, teniendo en cuenta que pueden existir obstáculos. Por ello debe estar dotado de sensores de tal modo que pueda recibir información de su entorno en tiempo real para prever colisiones, actualizar la trayectoria y finalmente llegar a su objetivo.

Existen dos tipos de planificación: local y global. Ambas coexisten para optimizar el funcionamiento del robot, de tal modo que pueda reaccionar a obstáculos y planificar la trayectoria que permite ir desde una localización inicial a una localización final.

2.2.1 Planificador local.

Su objetivo es evitar la colisión del robot con los objetos del entorno, por ello necesita un flujo de información constante obtenida por los sensores exteroceptivos (aquellos que permiten obtener información del entorno).

Se han desarrollado numerosos algoritmos:

Algoritmos *BUG*: se basa en seguir el contorno de cada obstáculo que encuentre el robot, de esta manera se asegura llegar al objetivo, el problema es que es muy ineficiente y no está pensado para obstáculos en movimiento. Existen dos tipos:

- ***BUG1*:** recorre el obstáculo por completo y luego se aleja en el punto más cercano del objetivo (ver Fig. 4)

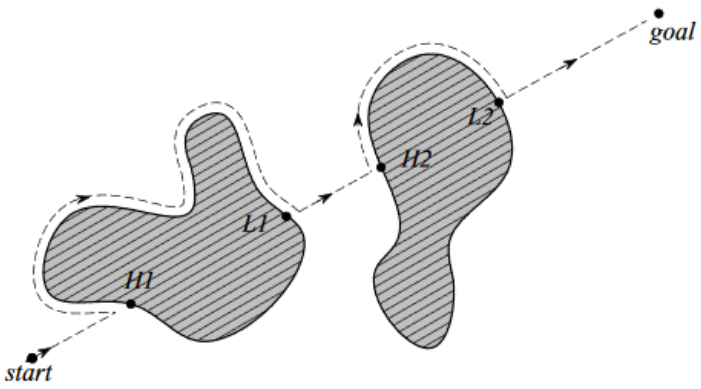

Fig. 4 Algoritmo *BUG1*

- ***BUG2*:** recorre el obstáculo y se aleja cada vez que se cruza con la recta que une el objetivo con el punto inicial (ver Fig. 5).

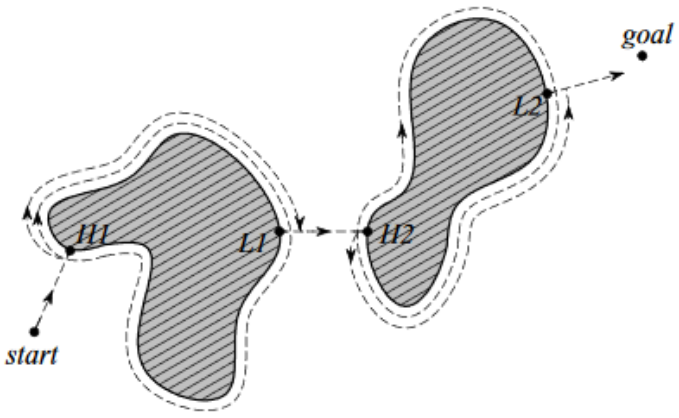

Fig. 5 Algoritmo *BUG2*

- ***Elastic Bands*:** consiste en crear un camino libre de colisiones y cambiante mediante burbujas [7]. En primer lugar, se conecta el inicio y destino (ver Fig. 6 (a)). A continuación, se reduce la trayectoria aplicando una fuerza de contracción en las bandas para eliminar holgura en el camino (ver Fig. 6 (b)), y otra de repulsión que responda a los obstáculos, de tal forma que se obtiene un equilibrio (ver Fig. 6 (c)). En el caso de aparecer nuevos obstáculos se generarían nuevas fuerzas para alcanzar una nueva posición de equilibrio (ver Fig. 6 (d)).

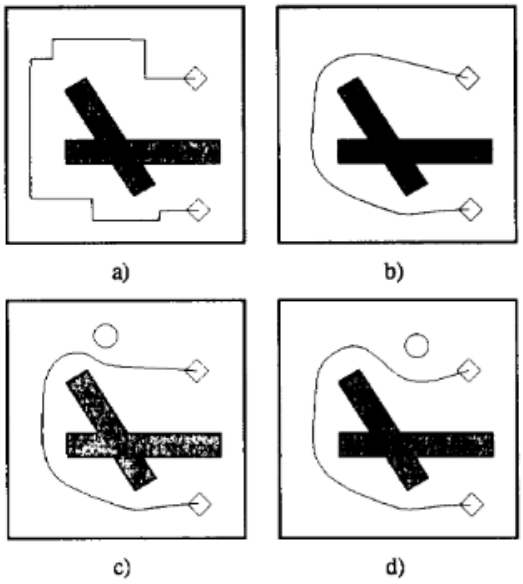

Fig. 6 Algoritmo *Elastic Bands*

Vector field histogram (VFH): crea un mapa local del entorno del robot, cuyas celdas almacenan la probabilidad de contener un obstáculo. Para evitarlos, VFH genera un histograma polar: el eje X representa el ángulo en el que se encontró un obstáculo y el eje Y representa la probabilidad de que realmente haya un obstáculo en la dirección del ángulo marcado por el eje X (ver Fig. 7).

A partir de este histograma se obtienen todos los pasos posibles y se selecciona el que menor función de coste G tenga. Esta función se basa en tres términos:

$$G = a \cdot dirección\_objetivo + b \cdot orientación\_ruedas + c \cdot dirección\_anterior.$$

Donde:

- Dirección_objetivo: es la alineación del robot con el objetivo.
- Orientación_ruedas: es la diferencia entre la nueva dirección y la orientación actual de las ruedas.
- Dirección_anterior: es la diferencia entre la dirección previa y la nueva.
- a, b, c: son parámetros de ajuste del comportamiento del robot y normalizado.

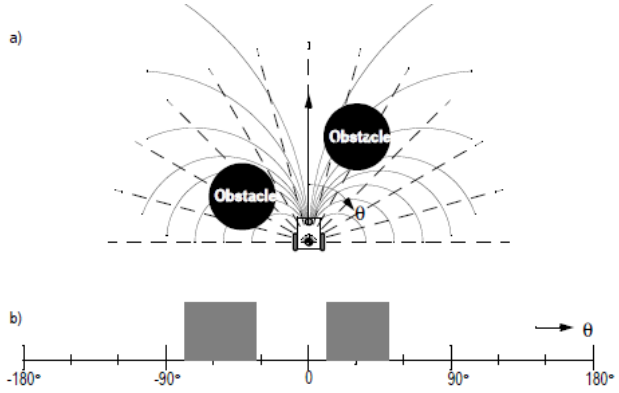

Fig. 7 Algoritmo VFH

***Dynamic Windows Approach (DWA)*:** permite obtener una dirección basándose en un tamaño de ventana que variará según la cinemática del robot. El algoritmo tiene dos fases: en primer lugar, se construye la ventana dinámica del robot y en segundo lugar se obtiene la nueva dirección (ver Fig. 8).

Considerando la velocidad del robot, el algoritmo genera una ventana dinámica basada en los pares de velocidad lineal (v) y velocidad angular (ω) que se pueden alcanzar. A continuación, se reduce dicha ventana seleccionando tan solo los pares que eviten chocar con un obstáculo, dichas velocidades son las velocidades admisibles.

En segundo lugar, se obtiene la nueva dirección aplicando una función objetivo sobre los pares de las velocidades admisibles. Dicha función objetivo es la siguiente:

$$O(v,\omega) = a \cdot progreso(v,\omega) + b \cdot velocidad(v,\omega) + c \cdot distancia(v,\omega)$$

Donde:

- Progreso: es la media del progreso hacia el lugar del destino con el par de velocidades seleccionado.
- Velocidad: es la velocidad lineal del robot.
- Distancia: es la distancia entre el objeto más cercano en su trayectoria
  - a, b, c: son parámetros de ajuste del comportamiento del robot y normalizado.

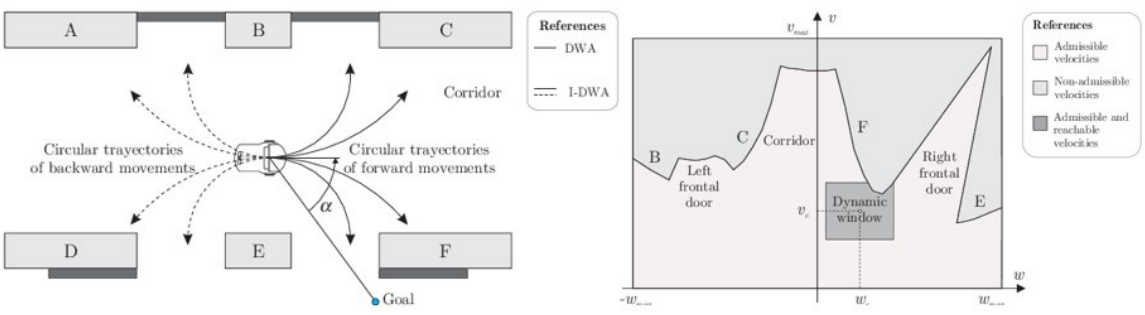

Fig. 8 Algoritmo *DWA*

2.2.2 Planificador global.

Se encarga de planificar la ruta que lleva al robot a las metas deseadas. Para lograr este objetivo hay que distinguir una serie de componentes:

Localización:

Es necesario saber la posición del robot dentro del entorno usando un mapa. Los algoritmos más utilizados para estimar la posición del robot en dicho mapa usan métodos probabilísticos. Estos métodos buscan estimar el estado del robot en el momento actual, *k,* conociendo el estado inicial y las medidas anteriores [8]. Habitualmente se obtiene el mapa mediante una acción de mapeo o *Mapping* y posteriormente se utilizan usan técnicas de localización en dicho mapa. También es posible generar el mapa y localizarse simultáneamente, (*SLAM, Simultaneous Localization and Mapping)*. La posición del robot se obtiene en dos fases:

- **Fase de predicción:** donde se usa un modelo de movimiento para predecir la posición del robot actual $p(x_k|Z^{k-1})$, teniendo solo en cuenta el movimiento. Se asumen que el estado actual solo depende del estado anterior (suposición de *Markov*) y de una entrada de control conocida. Además, el modelo de movimiento es especificado como una función de densidad condicionada $p(x_k|x_{k-1},\ u_{k-1})$ [8].
- **Fase de actualización:** donde se incorpora la información de los sensores. Se asume que las mediciones son independientes a las mediciones anterior, Z^{k-1} y que el modelo de medida se da en términos de probabilidad de que el robot se encuentra en una posición si se observa dicha medición, $p(z_k, x_k)$ [8].

Dependiendo de cómo se decida representar la función de densidad $p(x_k|Z^{k-1})$, se obtienen algoritmos variados:

- **Filtro de *Kalman*:** en caso de que el modelo de movimiento y de medición y el estado inicial se puedan describir usando una función de densidad gaussiana, entonces la función de densidad $p(x_k|Z^{k-1})$, también será una gaussiana [9].
- **Localización de *Markov* basada en cuadrícula**: se usan para tratar funciones de densidad multimodales y no gaussianas con una buena resolución, pero sufren de una gran carga computacional [10].
- **Localización topológica de *Markov***: se usa para la navegación de pasillos donde existen puntos de referencia [11].
- **Localización de Monte Carlo:** representan la función de densidad $p(x_k|Z^{k-1})$, como un conjunto de muestras que se extraen al azar [12].

Mapeo:

Un mapa es la representación del entorno que rodea al robot, luego influye directamente en la localización del robot. La precisión del mapa debe coincidir con la que se obtiene de los sensores del robot y debe ser apropiada al objetivo final que quiera alcanzar [13].

Como **representación continua** se encuentra los mapas geométricos, basados en la arquitectura del entorno. Requieren una elevada carga computacional, debido al uso de un conjunto de líneas infinitas. Por contra obtiene una gran precisión [13].

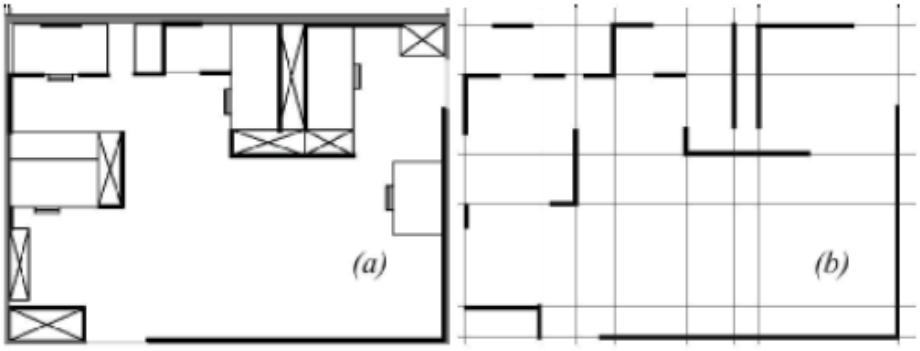

Fig. 9 Mapa geométrico

En la **representación discreta** se encuentran distintos métodos:

- **Descomposición en celdas exactas**: las celdas deben estar definidas de tal manera que se adapten a la configuración de los obstáculos. La posición particular del robot no importa, sino la habilidad del robot para atravesar desde una zona libre a otra adyacente (ver Fig. 10) [13].

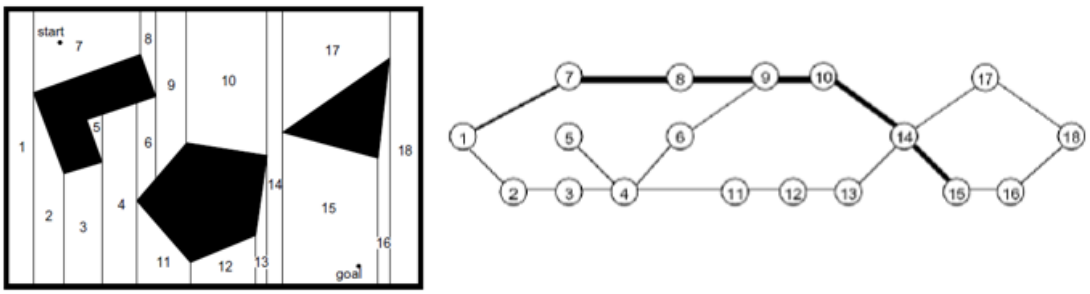

Fig. 10 Descomposición en celdas exactas

- **Descomposición en celdas fijas**: se descompone el espacio en celdas fijas con un valor binario: 0 si está libre y 1 si está ocupado. También es posible representar un estado desconocido. El principal problema es que los pasillos estrechos desaparecen (ver Fig. 11) [13].

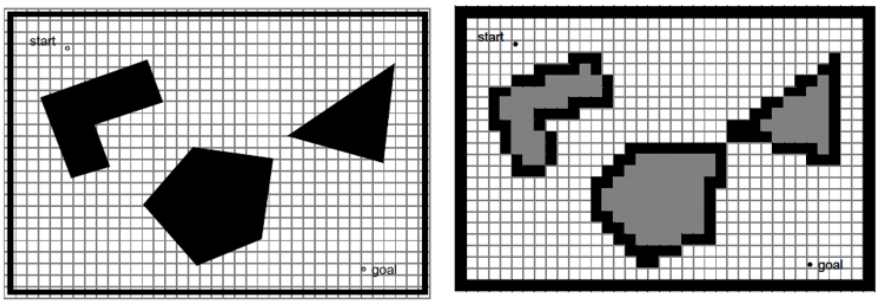

Fig. 11 Descomposición en celdas fijas

- **Descomposición en celdas de tamaño variables**: se parte de un tamaño fijo de celda, que se divide en cuatro si la zona está ocupada, hasta una resolución indicada. De esta manera se resuelve el problema de los pasillos estrechos (ver Fig. 12) [13].

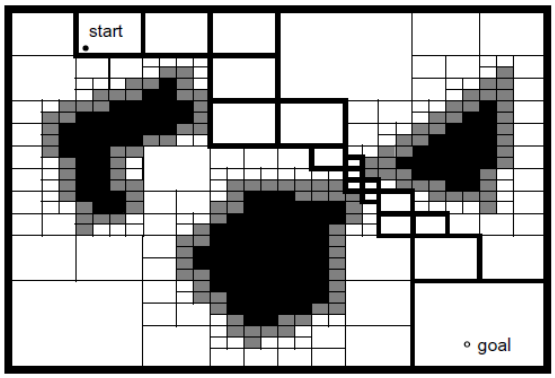

Fig. 12 Descomposición en celdas de tamaño variables

- **Rejillas de ocupación:** cada celda tiene un contador, en el caso de valer 0 indica que la celda está libre. A medida que aumenta, indica que ha sido alcanzado por el impacto de un sensor y disminuye con el impacto en una celda oculta tras de esta. El principal problema es que el tamaño del mapa aumenta conforme lo hace el entorno (ver Fig. 13) [13].

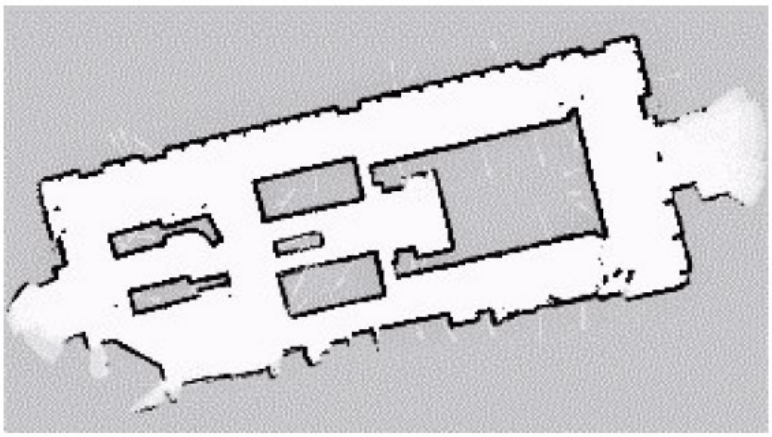

Fig. 13 Rejillas de ocupación

- **Representación topológica**: se centra en las características relevantes del entorno que faciliten alcanzar un objetivo en vez de medidas geométricas. Utiliza nodos para denotar áreas de interés y arcos para denotar pares de nodos adyacentes. Cuando un arco conecta dos nodos, indica que puede pasar de uno a otro sin necesidad de atravesar otro distinto (ver Fig. 14) [13].

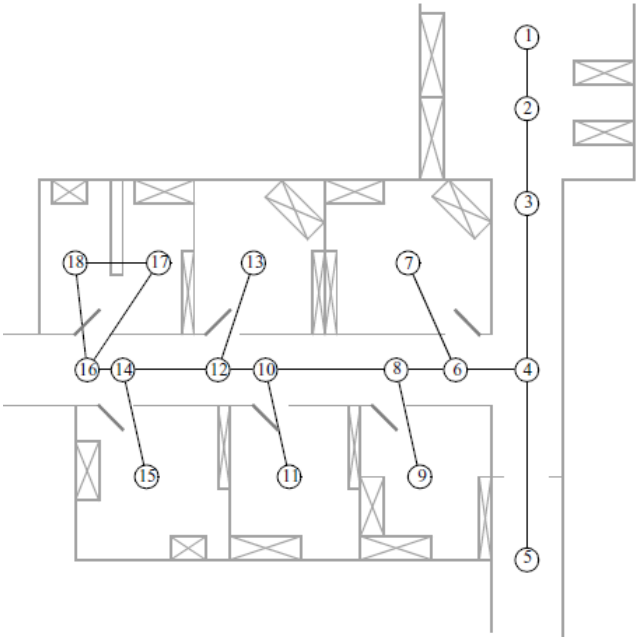

Fig. 14 Representación topológica

Planificación de la trayectoria:

Se encarga de designar un camino entre el punto inicial del robot y el objetivo final evitando colisiones con los obstáculos y teniendo en cuenta las limitaciones cinemáticas del robot [14].

Hay dos tipos de algoritmos para llevar a cabo la planificación de la trayectoria:

- **Algoritmos de búsqueda continua**: el robot va calculando el camino desde su posición durante todo el movimiento.
- **Algoritmos de búsqueda discretos**: el camino se divide en objetivos intermedios que deben ser alcanzados para lograr llegar al destino final. Los más utilizados son el algoritmo de campos potenciales, el algoritmo A* y el algoritmo de *Dijkstra* [15], [16].

2.3 Interacción humano robot.

El estudio de los distintos campos en donde los robots se relacionan, interactúan o comparten espacio de trabajo con los humano se le denomina interacción humano robot (*HRI del inglés Human Robot Interaction*) [17]. Existen principalmente dos tipos de interacción:

Interacción física humano robot

Este campo ha sido orientado principalmente al entorno industrial, donde operadores y robots comparten el mismo espacio de trabajo. El objetivo principal es mantener la integridad del operador, así como la del propio robot.

Una de las nuevas aplicaciones es conseguir que el operario pueda enseñar al robot manipulador un patrón de movimientos de tal manera que lo aprenda y lo optimice utilizando técnicas de Inteligencia Artificial.

Interacción social humano robot

Se busca conseguir que el robot tenga un comportamiento natural y social, desde el punto de vista humano. Es necesario establecer protocolos o mecanismos de comunicación, orientados a brindar una mayor comprensión de qué es lo que un humano quiere transmitir a un robot y que a su vez éste lo interprete de una manera correcta [17]. Los mecanismos de comunicación se pueden dividir en dos:

- **Comunicación verbal**: donde se han desarrollado métodos de reconocimiento de voz e interpretación del lenguaje natural.
- **Comunicación no verbal**: donde se han propuesto algoritmos de Inteligencia Artificial que permiten el reconocimiento de personas, gestos y emociones mediante el procesamiento de imágenes recibidas a partir de sensores de visión artificial.

2.3.1 Reconocimiento de personas.

Cada persona tiene características morfológicas que las diferencia del resto, por ejemplo, la forma de la cara, la voz, la huella.... Existen diversos sistemas biométricos para el reconocimiento de dichas características.

Un sistema biométrico está formado por componentes hardware, normalmente sensores que son los dispositivos encargados de extraer las características deseadas y de componentes software que permiten analizarlas y obtener unos patrones únicos. A continuación, se describen algunos tipos de sistemas biométricos existentes [18]–[20]:

- **Reconocimiento del hablante por voz**

La voz es una característica que las personas utilizan para identificar a los demás. Los sistemas de reconocimiento por voz tratan de identificar una serie de sonidos y patrones que permiten decidir si el usuario es quien dice ser.

El usuario se acredita pronunciando ciertas frases prediseñadas que permiten maximizar la cantidad de datos que se pueden analizar. Para autenticar a un usuario de forma fiable, se debe disponer de ciertas condiciones que permiten un registro correcto de los datos, como son la ausencia de ruido y ecos.

- **Reconocimiento por huella dactilar**

La huella dactilar es la estructura formada en la yema de los dedos por las crestas papilares [21]. Se trata de una estructura única para cada individuo y permaneces invariables a lo largo de la vida salvo lesión. En la Fig. 15 se muestran los puntos característicos de las huellas, que reciben el nombre de minucias.

Fig. 15 Características huella dactilar

El proceso para reconocer una huella se divide en dos partes. En primer lugar, es necesario obtener una imagen de esta y deber ser procesada para poder analizarla y extraer todas sus características. En segundo lugar, se aplican una serie de algoritmos que permiten compararla con las huellas que están almacenas en la base de datos del sistema.

- **Reconocimiento del iris**

El iris es la membrana coloreada del ojo que contiene la pupila en su centro. Los rasgos únicos que aparecen en el iris humano permiten identificar a un individuo de forma unívoca.

Los algoritmos para reconocimiento del iris analizan imágenes de alta calidad para obtener características clave, dicha información se guarda como una plantilla biométrica. Para autenticar a una persona, tan sólo es necesario realizar un escaneo en vivo del iris, aplicar de nuevo los algoritmos y comparar con las plantillas almacenadas.

- **Reconocimiento facial**

El rostro humano está compuesto está compuesto por diferentes estructuras y características. Los sistemas de reconocimiento facial no requieren de la intervención de personas para su funcionamiento, por ello se han convertido en uno de los sistemas de autenticación biométrica más utilizados.

Sin embargo, aún deben abordarse varios restos como son las condiciones de iluminación, la orientación de las imágenes, las expresiones faciales… Para desarrollar un sistema de reconocimiento facial robusto es necesario completar las siguientes fases (ver Fig. 16):

- **Detección de rostros:** es el encargado de localizar los rostros humanos en una imagen particular. EL objetivo principal es determinar si la imagen de entrada tiene rostros o no. Para facilitar el diseño del sistema y hacerlo más robusto, se suele hacer un preprocesamiento de las imágenes. Por ejemplo, se puede usar el algoritmo *Viola-Jones* [22], histogramas de gradiente orientados (*HOG*) [23] o análisis de componentes principales (*PAC*) [24], entre otros. Además de para detectar rostros, estos algoritmos se pueden utilizar para detectar objetos, detectar regiones de interés, etc.
- **Extracción de características:** el objetivo principal es extraer las características de los rostros obtenidos en el paso anterior. Cada rostro está caracterizado por su geometría, por ejemplo, la localización de los ojos, de la nariz de la boca, su tamaño y forma… Existe una gran cantidad de algoritmos que permiten obtener las características de los rostros, entre los que destacan: *HOG* [23], *Eigenface* [25], transformadas de *Fourier* [26], técnicas de patrón binario [27], transformación de características invariantes de escala (SIFT) [28], etc.
- **Reconocimiento facial:** se encarga de comparar las características obtenidas del paso anterior con las características de otros rostros almacenados en una base de datos. Puede usarse para: la verificación, que consistiría en comparar la cara con una conocida de la basa de datos y tomar la decisión de aceptación o rechazo; e identificación, que consistiría, en compararlas con un conjunto de rostros y encontrar la coincidencia más probable. En este paso las técnicas que se suelen usar son: los filtros de correlación (FC) [29], clasificadores de patrones como K-medias [30], C-medias [31], K-vecinos más próximos [32]… y redes neuronales convolucionales (CNN) [33].

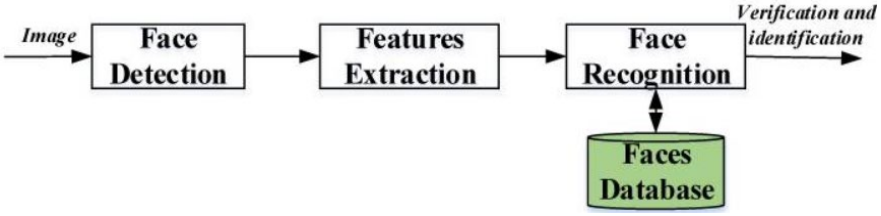

Fig. 16 Pasos para el reconocimiento facial

Clasificación de los sistemas de reconocimiento facial

Los sistemas se pueden clasificar según tres aproximaciones, basadas en los sistemas de detección y reconocimiento utilizados: enfoque local, enfoque holístico (vista general) y enfoque híbrido. El primero se basa en clasificar de acuerdo con ciertos rasgos faciales, sin considerar el rostro completo. El segundo emplea la cara entera como datos de entrada y luego se enfoca en ciertos planos de correlación. El tercero utiliza tanto características locales como globales para mejorar la precisión robustez de los sistemas (ver Fig. 17).

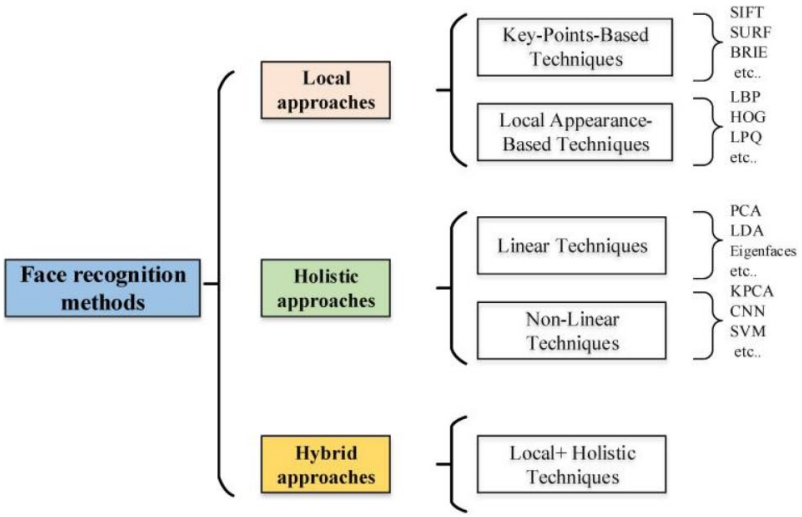

Fig. 17 Métodos de reconocimiento facial

Evaluación de los sistemas de reconocimiento facial

En el último paso del reconocimiento, se debe comparar el rosto extraído, con los rostros almacenados en una base datos. Existen varias técnicas para realizar dicha comparación y tomar una decisión:

- **Energía del pico de correlación**. Se utiliza la ecuación:

$$PCE = \frac{\sum_{i,j}^{N} E_{peak}(i,j)}{\sum_{i,j}^{M} E_{correlation-plane}(i,j)}$$

Donde:

- $E_{peak}(i,j)$: es el valor de la intensidad de pico de correlación
- $E_{correlation-plane}(i,j)$: es la energía total en el plano de correlación.

- **Distancia euclídea**. Se usa la ecuación:

$$D_E(P,Q) = \sqrt{\sum_i^n (p_i - q_i)^2}$$

Donde:

- P y Q son dos vectores.
- $p_i \; y \; q_i$ son los elementos i-ésimos de dichos vectores.

- **Distancia chi cuadrado.** Se usa la ecuación:

$$D(S_1,S_2) = \frac{1}{2}\sum_{i=1}^{m}\frac{(u_i - w_i)^2}{u_i + w_i}$$

Donde:

- $S_1 \; y \; S_2$ son dos histogramas.
- $u_i \; y \; w_i$ son los valores i-ésimos de dichos histogramas.

2.3.2 Reconocimiento de voz.

El reconocimiento de voz es el "subcampo interdisciplinario de lingüística computacional que desarrolla metodologías y tecnologías que permiten el reconocimiento y la traducción del lenguaje hablado en texto por la computadora" [34]. Existen diversos métodos para el reconocimiento de voz:

- **Modelos ocultos de *Markov* (HMM del inglés *Hidden Markov Models*) :** (ver Fig. 18) se tratan de modelos estadísticos, que permiten determinar parámetros desconocidos mediante parámetros observables. El proceso de reconocimiento consiste en el cálculo de la probabilidad de que una frase corresponda a una señal acústica sobre todo el conjunto de palabras posibles. El problema de este modelo es que requiere de un entrenamiento individualizado del modelo acústico, de la pronunciación y del modelo de lenguaje.

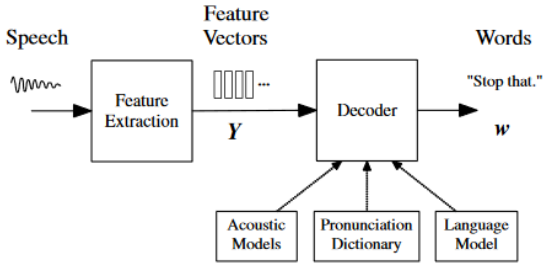

Fig. 18 Estructura de modelo de reconocimiento de voz basado en HMM

- **Reconocimiento de voz basado en *Dynamic Time Warping (DTW) :*** (ver Fig. 19) es un algoritmo que permite medir la semejanza entre dos secuencias que pueden variar en el tiempo o la velocidad. Cualquier dato que pueda transformarse en una representación lineal, puede analizarse con DTW, por eso se ha aplicado e video, gráficos y audio.

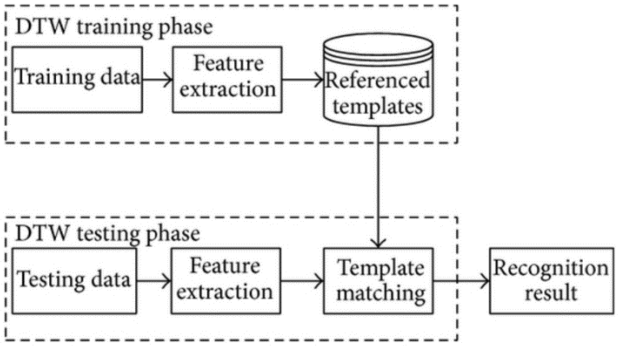

Fig. 19 Estructura de modelo de reconocimiento de voz basado en DTW

- **Reconocimiento automático de voz de extremo a extremo (E2E)** : (ver Fig. 20) está basado en redes neuronales. Los sistemas de reconocimiento tradicionales se componen de un modelo acústico, de un modelo de lenguaje y de un modelo de pronunciación, requieren un entrenamiento por separado de estas componentes. Por el contrario, los modelos E2E presentan un enfoque único integrado con una línea de entrenamiento mucho más simple. De este modo, se consigue reducir el tiempo de entrenamiento y el tiempo de decodificación.

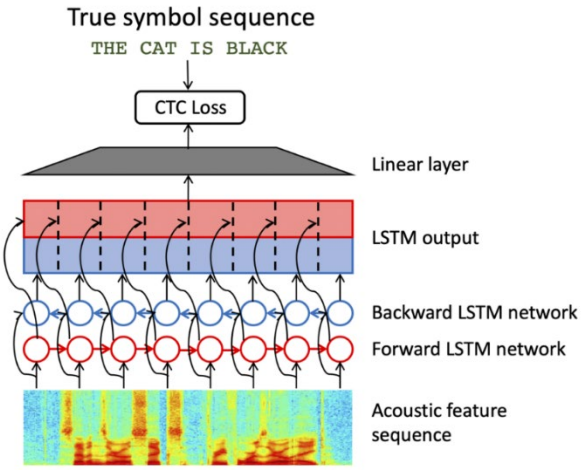

Fig. 20 Red BLSTM basada en bloque para reconocimiento de voz

Los métodos de reconocimiento de voz extremo a extremo han sido adoptados por *Google, Microsoft, Apple, Amazon, IBM*, entre otros.

2.3.3 Síntesis de voz.

Los sistemas TTS (del inglés *Text to Speech*) convierten el lenguaje de texto normal en habla (ver Fig. 21). La calidad de un sintetizador de voz se basa en el grado de similitud con una voz humana y de la capacidad de ser entendible.

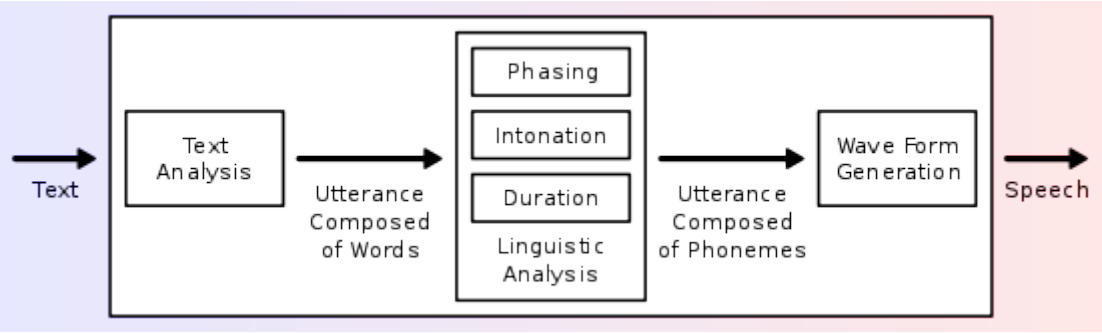

Fig. 21 Diagrama se un sistema TTS tradicional

Existen diversas empresas que ofrecen servicios de este tipo, de manera online. La ventaja principal es que permiten dotar a dispositivos hardware de bajas prestaciones en inteligentes. Sin embargo, hay que tener en cuenta que los servicios de la Nube están sujetos a la disponibilidad de una conexión a internet y de una tarifa por su uso. Entre otros, destacan:

Azure Cognitive Service

Es la familia de servicios de Inteligencia Artificial y API cognitivas que ayudan a crear aplicaciones inteligentes, por parte de la plataforma *Microsoft Azure* [35]. En concreto tiene una API TTS que permite dotar de voz natural a textos, disponiendo de 50 idiomas y 200 voces diferentes.

Google Cloud

Está incluido en Google Cloud. Entre otros, proporciona un servicio de síntesis de voz denominado *Text-to-Speech* basado en las tecnologías de IA de Google [36]. Permite seleccionar más de 40 idiomas y 220 voces y permite entrenar un modelo propio de síntesis de voz.

Amazon Web Services

Es la plataforma en la nube de *Amazon*. Dispone de *Amazon Polly*, que además de la transformación de texto a voz estándar, dispone de transformaciones con redes neuronales (NTTS) que proporciona mejoras avanzadas en la calidad el habla a través de aprendizaje automático [37].

Por otro lado, existen soluciones que no requieren conexión a internet. El principal problema es que se tratan de sintetizadores de menor calidad, debido a que la voz no suele ser realista. Por ejemplo, se pueden usar:

eSpeak

Es un sintetizador de voz *de código abierto* sencillo que permite controlar la velocidad y el tono [38]. Se basa en la conversión de texto a fonemas, por ello no requiere de un elevado cómputo computacional.

Festival

Se trata del sintetizador multilenguaje gratuito creado por la Universidad de Edimburgo [39]. La síntesis se basa en difonos [40], por ello se puede ejecutar en múltiples plataformas.

pyttsx3

Es un sintetizador que utiliza para su motor los paquetes de idiomas que están instalados en el sistema operativo [41], [42].

2.3.4 Reconocimiento del lenguaje natural.

El procesamiento de lenguaje natural (NLP, del inglés Natural Language Processing) es una rama de la Inteligencia Artificial que se encarga de procesar datos del lenguaje. Las funciones de los modelos de NLP son las siguientes:

- **Segmentación de palabras:** se basa en separar las palabras (*tokens*) de un texto.
- **Segmentación de oraciones:** se encarga de separar las oraciones de un texto.
- **Lematización o *stemming*:** el objetivo es obtener la forma base de la palabra. Por ejemplo, el lema de una conjugación verbal sería el infinitivo, mientras que de una palabra en plural, sería el singular.
- **Segmentación morfológica:** consiste en obtener los morfemas de las palabras.
- **Análisis sintáctico:** se tratan de las relaciones que guardan las palabras de una frase.
- **Etiquetado gramatical (POS):** determina la función de una palabra dentro de la frase.

La Fig. 22 muestra, de forma resumida, el proceso de extracción de estructuras con cierta semántica a partir del texto de un documento.

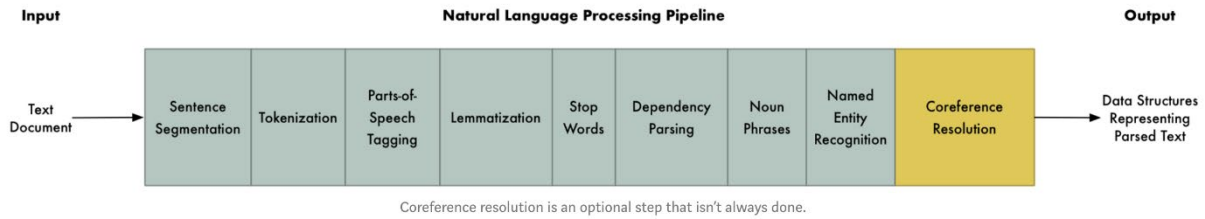

Fig. 22 Pipeline de procesamiento de lenguaje natural

Además de los servicios de procesamiento de lenguaje de natural que ofrecen las distintas nubes comentadas anteriormente, existen módulos que cuentan con prestaciones específicas:

NLTK

Es una de las plataformas líderes en el procesamiento de lenguaje natural. Proporciona algoritmos sencillos para el análisis de todo tipo de recursos léxicos tales como *WordNet* [43]. Y posee un conjunto de librerías que proporcionan las funciones de procesamiento de texto ya descritas, cuya disponibilidad dependerá del idioma utilizado [44].

Proporciona una gran flexibilidad dado que está disponible en los tres sistemas operativos principales (*Linux, Mac OS y Windows*). Además, es una herramienta gratuita de código abierto lo cual permite que sea accesible para investigadores y estudiantes.

Spacy

Es una biblioteca implementada en *Python* que contiene algoritmos específicamente diseñados para su uso en el desarrollo de soluciones en el ámbito industrial y productivo, dado que permite procesar grandes cantidades de texto [45].

Se integra a la perfección con las plataformas de inteligencia artificial como *TensorFlow y PyTorch*, permitiendo construir de forma sencilla modelos para problemas sofisticados en el ámbito del NPL (ver Fig. 23).

Fig. 23 Funciones de la herramienta *Spacy*

UDPipe

Es un paquete de procesamiento de texto diseñado tanto en R como en C++ que permite realizar tareas como segmentación, etiquetado, lematización y análisis sintáctico [46].

Contiene una serie de modelos previamente entrenados disponibles para más de 65 idiomas. Además, permite entrenar modelos propios mediante aprendizaje profundo.

Stanza

Es una colección de herramientas desarrolladas por la Universidad de Stanford para el análisis de lenguaje natural [47]. Permite realizar todas las funciones de NLP en un gran número de idiomas.

Está implementado en *Python* por lo que se requiere un esfuerzo mínimo de configuración. Es una librería que contiene modelos neuronales previamente entrenados de 66 idiomas, y también incluye modelos biomédicos (ver Fig. 24).

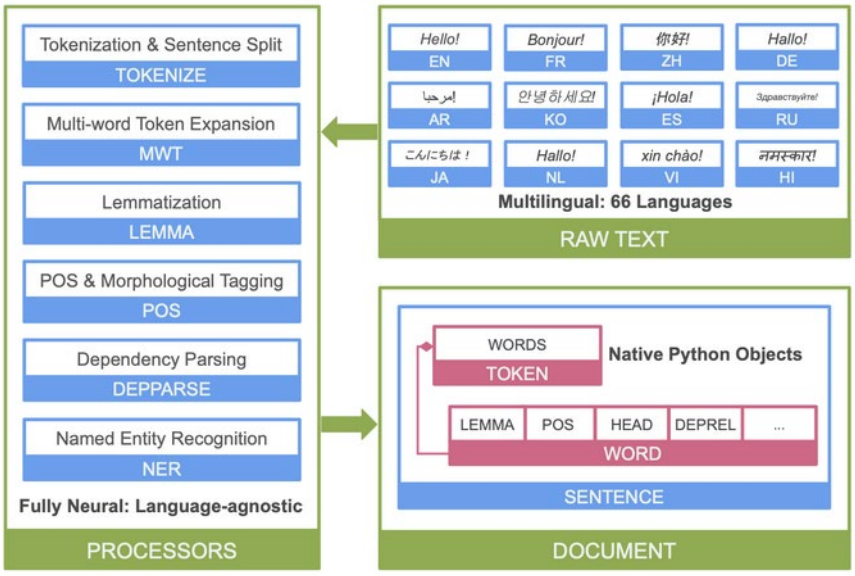

Fig. 24 Funciones de la herramienta *Stanza*

3 Diseño del sistema.

En este capítulo se expondrá brevemente la arquitectura hardware, ya que al final se ha trabajado con un simulador. Y se entrará más en detalle en la arquitectura software, donde se explicarán por un lado las aplicaciones utilizadas, *ROS* y el simulador *Gazebo*, y por otro las técnicas de Inteligencia Artificial.

Dentro de las técnicas de Inteligencia Artificial, en primer lugar se van a definir la estructura de los diferentes modelo y en segundo lugar una vez comprendidas se implementarán de forma independiente.

En cuanto a *ROS y Gazebo*, se describirán primero dichos entornos y para qué se usan. Posteriormente se explicará el paquete de navegación de *ROS* y cómo se implementa, además de la integración de todos los módulos mediante *ROS.*

3.1 Arquitectura del sistema.

A continuación, se explicarán con detalle los aspectos más importantes de este proyecto. Por un lado, la arquitectura hardware se describirá brevemente porque se ha optado por realizar simulaciones (apartado 3.2). Por otro lado se profundizará en la arquitectura software, cuyo peso es mayor, y, por tanto, se explicarán tanto las herramientas utilizadas como los modelos de aprendizaje profundo que se han usado junto a su arquitectura (apartado 3.3). Por último se explicará cómo se han integrado todos los módulos gracias a la estandarización que proporciona ROS (apartado 3.4).

3.2 Arquitectura hardware.

Dada la situación de pandemia causada por la Covid-19 no se ha podido trabajar con un hardware específico. Ya que se trata de un trabajo que se puede realizar mediante simuladores y es fácilmente extrapolable al hardware se ha considerado el uso de un simulador lo más realista posible que facilite el posible despliegue en un robot y escenario reales, cuando la situación epidemiológica mejore.

En un principio, parte del hardware que se quería utilizar además del chasis del robot junto a sus motores y sensores, era una *Raspberry Pi* modelo 4 y un sensor RGB-D *Azure Kinect DK* [48] .

La Tabla 1 muestra las características del modelo *Raspberry Pi 4* (ver Fig. 25).

La Tabla 2 muestra las características del sensor *Azure Kinect DK* (ver Fig. 26).

Tabla 1 Características *Raspberry Pi 4*

Características Raspberry Pi 4	
Procesador	CPU ARM Cortex-A72 de 64 bits con cuatro núcleos a 1,5 GHz
Memoria RAM	2 GB, 4 GB o 8 GB de SDRAM LPDDR4
Tarjeta gráfica	Gráficos de VideoCore VI, compatibles con OpenGL ES 3.x
Conectividad	Gigabit Ethernet , red inalámbrica 802.11ac de doble banda y bluetooth 5.0
Vídeo y sonido	Dos puertos micro-HDMI que admiten pantallas de 4K@60Hz a través de HDMI 2.0, puerto de pantalla MIPI DSI, puerto de cámara MIPI CSI, salida estéreo de 4 polos y puerto de vídeo compuesto.
Puertos	Dos puertos USB 3.0 y dos puertos USB 2.0
Alimentación	5V/3A vía USB-C, 5V vía cabezal GPIO

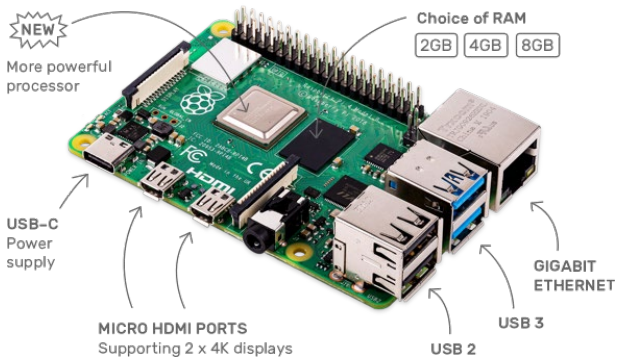

Fig. 25 *Raspberry Pi 4*

Tabla 2 Características *Azure Kinect DK*

Características Azure Kinect DK
1. Sensor de profundidad de 1 MP con opciones de campo de visión ancho y estrecho que permiten optimizarlo para una aplicación.
1. Matriz de 7 micrófonos para capturar sonidos y voz de campo lejano

2. Cámara de vídeo RGB de 12 MP para obtener una secuencia de colores adicional en línea con la secuencia de profundidad
3. Acelerómetro y giroscopio (IMU) para la orientación del sensor y el seguimiento espacial.
4. Conexiones de sincronización externas para sincronizar fácilmente secuencias de sensores procedentes de varios dispositivos Kinect de forma simultánea.

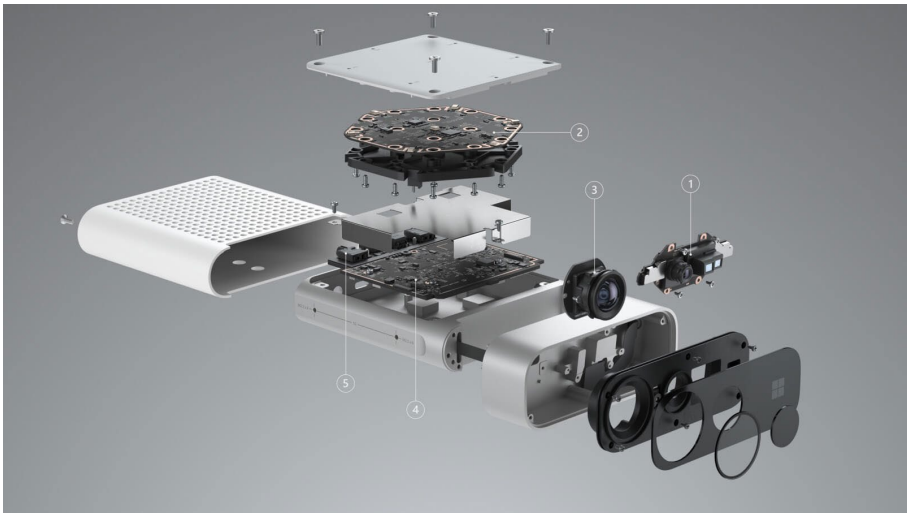

Fig. 26 Contenido de Azure Kinect DK

Al final se ha trabajado con un equipo doméstico donde se ha realizado el proyecto y las diferentes pruebas, cuyas características principales se pueden encontrar en la Tabla 3.

Tabla 3 Características equipo doméstico

Procesador	Intel core i7 2630QM 2.00GHz
Memoria RAM	8.00 GB DDR3 665MHz
Tarjeta Gráfica	ATI AMD Radeon HD 6700M 1024MB
Cámara Web	Logitech Carl Zeiss Tessar
Sistema Operativo	Ubuntu 18.04

3.3 Arquitectura software.

El objetivo es diseñar un robot con navegación autónoma que pueda interaccionar con las personas de forma sencilla. A pesar de que existen servicios que proporciona la Nube que pueden realizar ciertas aplicaciones requeridas, se han descartado principalmente por tres motivos:

1. No suelen ser flexibles.
2. Generan un coste continuo.
3. Se requiere conexión a Internet.

Para suplir esas desventajas, se han utilizado una serie de modelos de aprendizaje profundo gratuitos que no requieren conexión a Internet y se han adaptado al proyecto. Las principales librerías para trabajar con esos modelos que se han empleado han sido *TensorFlow, OpenCV* junto a *Caffe* y *spaCy*; bibliotecas que se muestran en la Fig. 27 Componentes de la arquitectura software.

Para integrar todos los componentes del sistema se ha utilizado ROS y el lenguaje *Python,* ya que la mayoría de las librería de aprendizaje profundo se han diseñado en este lenguaje. También se ha trabajado con *Gazebo* que permite realizar simulaciones del robot en un entorno 3D [49].

Se ha dividido el sistema en diferentes módulos para facilitar el trabajo, además se han creado librerías propias para cada uno de tal forma que se pueda simplificar el código y se pueda reutilizar en trabajos futuros. Los módulos en los que se ha dividido el sistema se pueden ver en la Fig. 27 Componentes de la arquitectura software y son los siguientes:

- Detección facial
- Reconocimiento facial
- Transcripción de voz a texto.
- Síntesis de voz.
- Procesamiento de lenguaje natural.
- Navegación.

Fig. 27 Componentes de la arquitectura software

3.3.1 Ecosistema Gazebo/ROS.

1) ROS.

Robot Operating System (ROS) es un *framework* que permite desarrollar e implementar software para robótica. Proporciona los servicios propios de sistemas operativos, aunque no lo es, tales como la abstracción de hardware, el control de dispositivos de bajo nivel o la implementación de funciones de uso común, entre otros [50]. El principal objetivo es la estandarización, beneficio que se vuelve sustancial cuando un sistema está compuesto de múltiples componentes o se requiere coordinación entre varios robots.

Este entorno de trabajo está soportado de forma oficial en *Linux* pero también se puede instalar en otros sistemas operativos como *Windows, Mac OS* e incluso en dispositivos móviles. Además, soporta diferentes lenguajes de programación como C++ y *Python*.

Una aplicación basada en ROS se compone de nodos, cada nodo es un proceso que se ejecuta en un hilo de procesamiento. El encargado de administrar estos nodos es el *ROS master* que mantiene el registro de todos los nodos activos en el sistema. Cada nodo, puede utilizar este registro para descubrir otros nodos y establecer líneas de comunicación con ellos. Además, el ROS master también contiene el servidor de parámetros, en inglés *parameter server,* que almacena parámetros y valores de configuración que son compartidos entre los nodos.

La comunicación entre nodos se basa en mensajes, que son los datos que se transfieren entre ellos. *ROS* es flexible en términos del lenguaje de programación; los nodos escritos en diferentes lenguajes pueden comunicarse siempre que la estructura del mensaje y los datos sean válidos. Existen dos tipos de comunicación principal: la comunicación publicador-subscriptor (en inglés, publisher-suscriber) y la comunicación cliente-servidor (services).

Publicador-subscriptor

Se trata del modo de comunicación más habitual. Hay un nodo que actúa como publicador de mensajes en un *topic*, otros nodos pueden suscribirse a éste. Un nodo de *ROS* puede publicar o suscribirse a varios *topics*. Se trata de una modo de comunicación desacoplado y anónimo, es decir, el publicador y el suscriptor no se conocen (ver Fig. 28 Comunicación publicador-subscriptor

Fig. 28 Comunicación publicador-subscriptor

Cliente-servidor

En este caso, un nodo funciona como servidor y otro como cliente consumiendo los servicios que proporciona el primero. A diferencia de la publicación-suscripción, se trata de una comunicación bidireccional síncrona: el cliente inicia la solicitud, el servidor la ejecuta y la comunicación finaliza hasta la próxima solicitud. Otra diferencia es que solo hay un servidor por servicio (ver Fig. 29).

La estructura de los servicios es muy variada, desde una petición que requiera una respuesta hasta una petición vacía que no requiera ninguna respuesta, pasando por todas las combinaciones posibles.

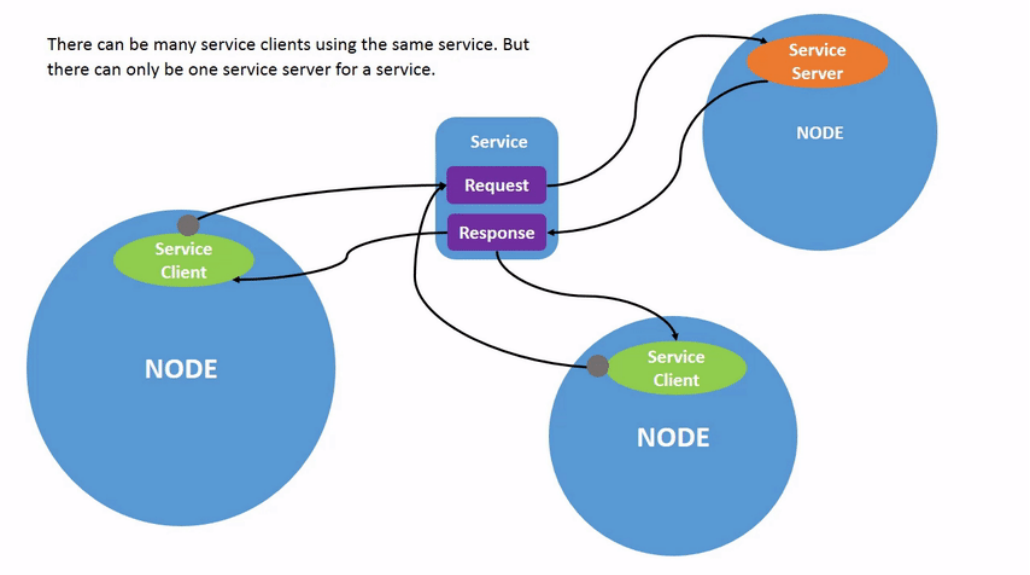

Fig. 29 Comunicación cliente-servidor

2) Gazebo.

Gazebo "es un simulador de entornos 3D que posibilita evaluar el comportamiento de un robot en un mundo virtual. Permite, entre muchas otras opciones, diseñar robots de forma personalizada, crear mundos virtuales usando sencillas herramientas CAD e importar modelos ya creados." [51]

Además, es posible sincronizarlo con ROS de forma que los robots emulados publiquen la información de sus sensores en nodos, así como implementar una lógica y un control que dé ordenes al robot.

Las simulaciones se han realizado con el robot *Pioneer P3-DX*. Se trata de un robot móvil de investigación con un sistema de tracción diferencial. Además, se ha incluido un sensor *2D-LiDAR LMS100* cuyo ángulo de abertura horizontal es de 270 ° con una resolución angular de 1 ° y un alcance de 20m (ver Fig. 30).

Fig. 30 Robot Pioneer P3-DX y sensor LMS100

3.3.2 Métodos de Inteligencia Artificial utilizados.

La Inteligencia Artificial es la disciplina que trata de crear sistemas capaces de razonar como un ser humano, aprender de la experiencia y averiguar cómo resolver problemas ante unas condiciones dadas [52]. Puede ser tan simple como seguir un diagrama de flujo lógico, o puede ser un ordenador que sea capaz de aprender de una gran cantidad de entradas sensoriales y aplicar ese conocimiento a nuevas situaciones. Los principales subconjuntos de la Inteligencia Artificial son los Sistemas Expertos, el Aprendizaje Automático (ML, del inglés, *Machine Learning*) y el Aprendizaje Profundo (DL, del inglés, Deep Learning).

Un sistema de conocimiento basados en reglas (Sistemas Expertos) es un programa diseñado para resolver problemas complejos y proporcionar la capacidad de tomar decisiones como un humano [53]. Para ello usan un conjunto de reglas condicionales (*if-then*) que permiten a la máquina realizar un proceso de razonamiento para alcanzar una conclusión o recomendación. El principal problema es que no son capaces de aprender por sí solos de la experiencia, por ello es necesario su constante actualización.

Este tipo de sistemas han sido utilizados, una vez aplicados los algoritmos de procesamiento de lenguaje natural, para diferenciar una serie de casos según el tipo de frase analizada. Se verá su uso en la sección 3.3.6.

El Aprendizaje Automático (ML) es una rama de la Inteligencia Artificial basada en la idea de que los sistemas pueden aprender de datos, identificar patrones y tomar decisiones con la mínima intervención humana. Emplea algoritmos basados en técnicas matemáticas (Estadística, Teoría de Probabilidad, Optimización, etc), que permiten entrenar un modelo para realizar inferencias, una vez implementado.

Este subconjunto de la IA se ha usado para el Procesamiento de Lenguaje Natural (PNL). En concreto, se ha utilizado un modelo entrenado usando *MorphoDiTa,* que es una herramienta de aprendizaje automático supervisado para entrenar modelos lingüísticos [54], [55]. El modelo entrenado consiste en un diccionario morfológico y etiquetador capaz de analizar las frases y realizar una clasificación sintáctica de los términos.

El aprendizaje profundo *(*DL) es un subconjunto del Aprendizaje Automático basado en Redes Neuronales Artificiales (RNA). El proceso de aprendizaje se llama profundo porque la estructura de redes neuronales artificiales se compone de varias capas de entrada, salida y ocultas. Cada capa contiene unidades que transforman los datos de entrada en información que la capa siguiente puede usar para realizar una tarea de predicción determinada. Gracias a esta estructura, una máquina puede aprender a través de su propio procesamiento de datos [56]. En la sección 3.3.2 1) se detallará el funcionamiento de una red neuronal y los modelos utilizados.

La Fig. 31 muestra, de forma resumida, los subconjuntos de IA descritos.

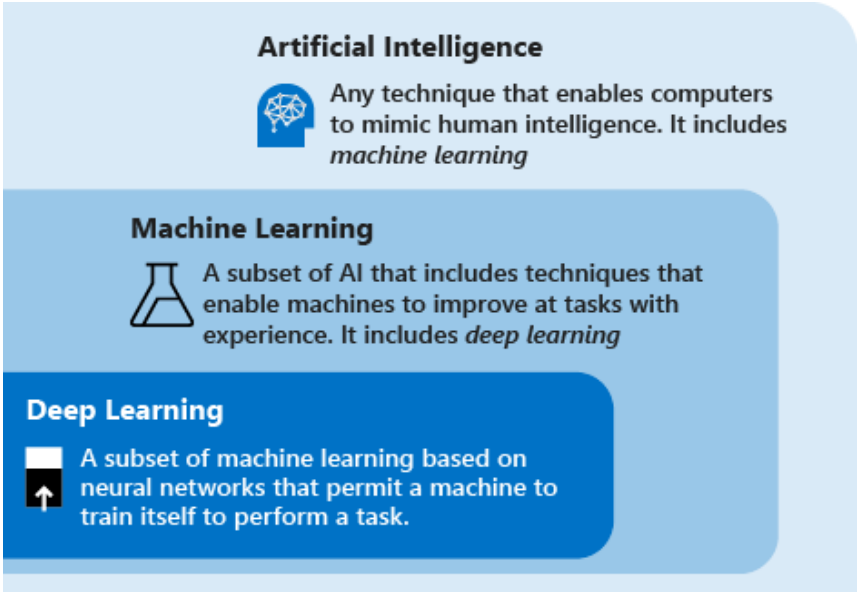

Fig. 31 Subconjuntos inteligencia artificial

1) Métodos de Deep Learning utilizados.

En la sección nº 3.3.2 se ha hablado de que el Aprendizaje Profundo está basado en RNAs, por ello, se van a explicar brevemente. Una red básica es el Perceptrón (con una única neurona) y mediante la unión de varios de ellos se puede crear una RNA multicapa. Cada unidad básica una función en base a sus entradas, y dará una señal de salida que enviará a otra unidad (ver Fig. 32) [57].

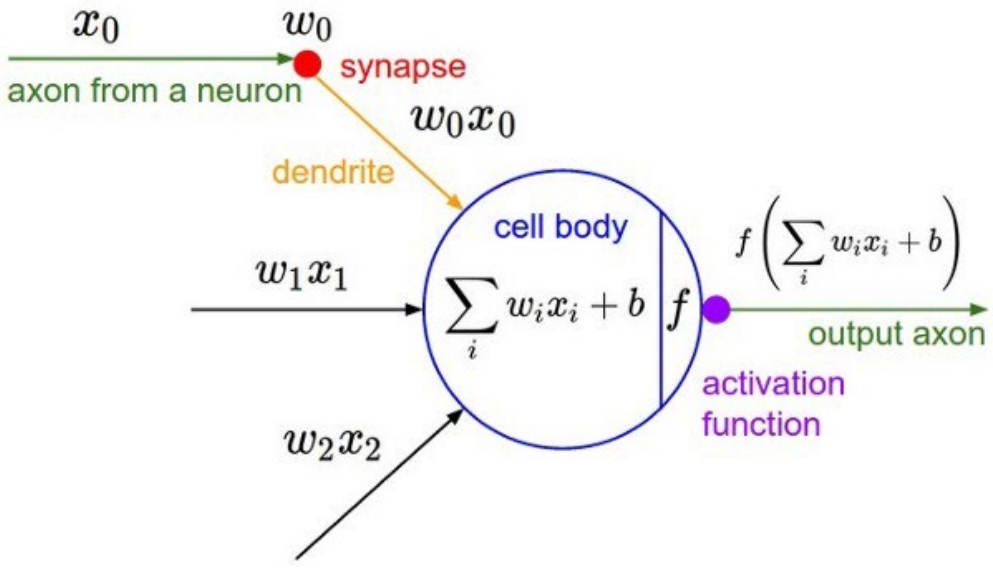

Fig. 32 Esquema de un perceptrón

Como se ve en la Fig. 32, el funcionamiento básico de una neurona se podría representar mediante la siguiente ecuación:

$$y = f(\sum_{i=0}^{n}(w_i x_i) + b)$$

La salida representada como y, es la activación (función f) de la suma de cada una de las entradas del perceptrón multiplicadas por su respectivo peso (w_i) y la suma de un parámetro denominado bias (b). La función de activación tendrá un peso importante en el funcionamiento de la red. Entre las funciones de activación más utilizadas están las que se encuentran en la Fig. 33.

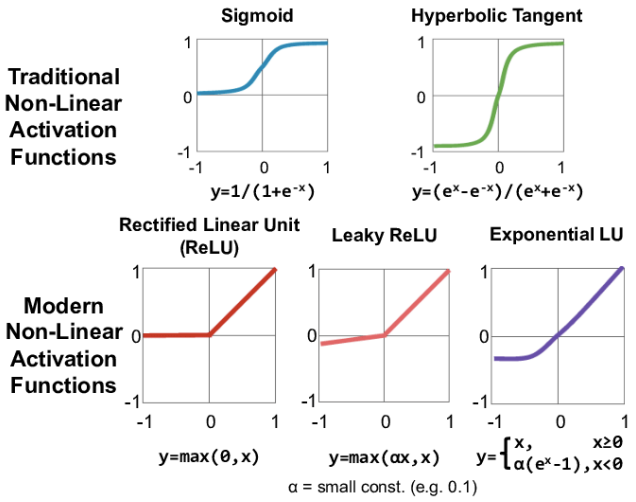

Fig. 33 Funciones de activación.

Por otro lado, en la creación de la red neuronal hay que tener en cuenta su arquitectura. Cuanto mayor sea el número de capas y neuronas, mayor será la complejidad, el nivel de abstracción y el coste computacional. El diseño de la estructura irá principalmente ligado a la complejidad del problema, sin embargo, la optimización de una arquitectura se conseguirá generalmente mediante un proceso de prueba y error hasta dar con la configuración más adecuada.

En la Fig. 34 se puede observar que las redes se componen de tres tipos de capas:

- **Capa de entrada**: que admite datos de entrada. El número de neuronas dependerá de la cantidad de la dimensión de cada entrada. Está al inicio de la red y sólo puede haber una.
- **Capas ocultas**: se encarga de procesar los datos para conseguir los resultados deseados, se pueden crear tantas capas con el número de neuronas que se desee, siempre teniendo en cuenta que repercutirá en el tiempo de procesamiento. Se encuentran entre la capa de entrada y de salida.
- **Capa de salida**: devuelve los datos de salida. Al igual que la capa de entrada solo puede haber una y el número de neuronas depende del problema a resolver.

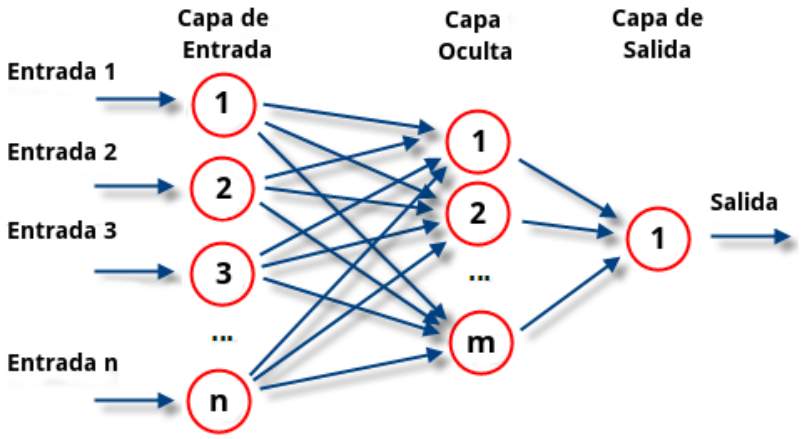

Fig. 34 Representación gráfica de la arquitectura de un perceptrón mutlicapa.

La estructura de modelos neuronales vistos se suele utilizar para el análisis de datos numéricos. Cuando se quiere analizar imágenes o volúmenes la estructura de la red requiere algunas modificaciones para poder aprovechar todo el potencial. En una imagen por ejemplo, es importante tanto el valor de un pixel como el valor de los más cercanos ya que aportan información, luego pueden ayudar a obtener la solución deseada. Las redes convolucionales se suelen componer de las siguientes capas ocultas [58]:

- **Capa convolucional**: aplica una convolución que consiste en realizar el producto escalar de un filtro (*kernel*) sobre los píxeles de entrada como se observa en la Fig. 35. Con esta operación se consigue varios píxeles aporten información en su conjunto. La dimensión del *kernel* será la misma que los datos de entrada, y el tamaño dependerá del problema, aunque se suelen utilizar filtros de 3x3. Del mismo modo, la salida de la capa convolucional tendrá la misma dimensionalidad que la entrada y será el resultado de la convolución del filtro con los datos de entrada.

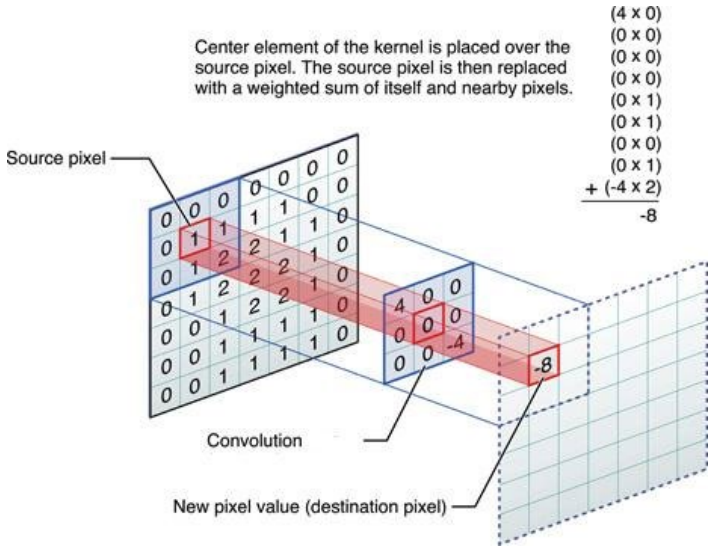

Fig. 35 Convolución de dos dimensiones

- **Capa de submuestreo (*Pooling layer*):** permite reducir el número de parámetros calculados ya que consiste en reducir el tamaño de una matriz. Existen varios tipos de submuestreo, pero el más habitual toma el máximo valor de un conjunto de píxeles y descarta los demás (ver Fig. 36). Éste se denomina submuestreo por máximos (*maxpooling*) . Otros tipos de submuestreo que utilizan el mismo planteamiento son el submuestreo por mínimos (*minpooling*) y el submuestreo por media (*average pooling*).

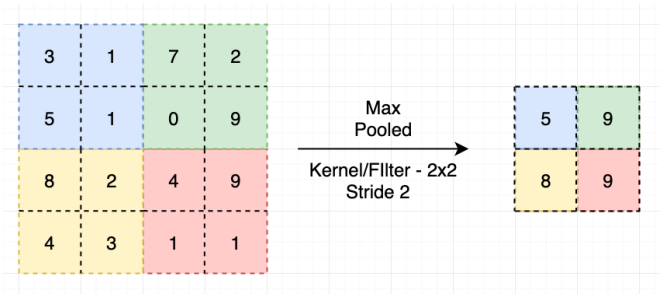

Fig. 36 Submuestreo por máximo

- **Capa de interpolación: (*Unsumpling layer*):** utiliza el planteamiento contrario al submuestreo, es decir, aumenta el número de píxeles de la imagen para que la capa de salida devuelva una imagen con las dimensiones iniciales.
- **Capa completamente conectada (*Fully-connected layer*):** se coloca al final de la arquitectura para cambiar la dimensión del problema, permitiendo utilizar las capas ocultas tradicionales (ver Fig. 37).

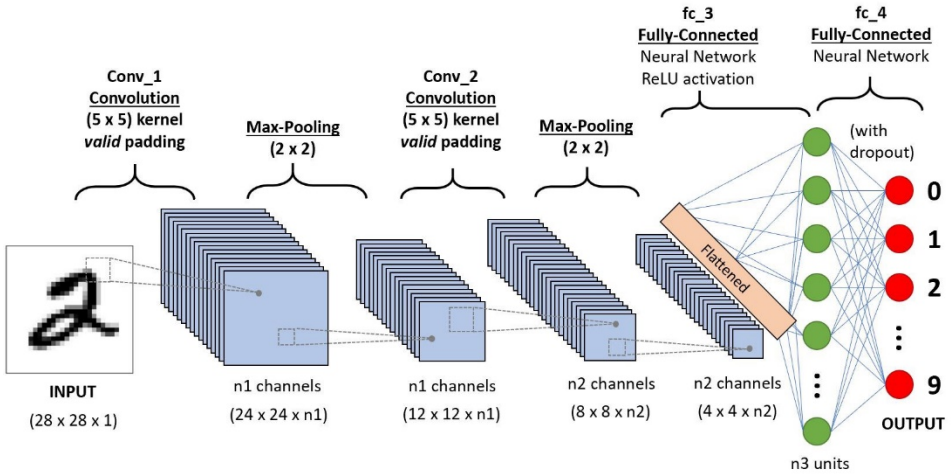

Fig. 37 Ejemplo de la arquitectura de una CNN

Una vez finalizado el diseño de la arquitectura de la red se pasa a la fase de entrenamiento. El objetivo de dicho entrenamiento es ajustar los pesos de cada neurona para lograr la minimización del error entre la salida que produce la red y la salida deseada o de referencia. Para calcular dicho error existen distintas fórmulas:

- **Error cuadrático medio**. En la ecuación y_i representa cada valor de salida que produce la red y x_i cada valor de salida de referencia.

$$ECM = \frac{1}{n}\sum_{i=1}^{n}(y_i - x_i)^2$$

- **Error medio absoluto**. En la ecuación y_i representa cada valor de salida que produce la red y x_i cada valor de salida de referencia.

$$EAM = \frac{1}{n}\sum_{i=1}^{n}|y_i - x_i|$$

- **Entropía cruzada**. En este caso para varías etiquetas x se calcula la probabilidad de pertenecer a cada una de ellas la entrada, y para calcular el error se compara la probabilidad ideal (que será 1 o 0) con la que se ha designado para esa misma entrada. El error total será el sumatorio del error de cada una de las etiquetas (x) para cada uno de los píxeles. En la ecuación q(x) representa la probabilidad para una etiqueta estimada y p(x) la probabilidad ideal [57].

$$H(p,q) = -\sum_{i,j}\sum_{x} p_{ij}(x)\log\left(q_{ij}(x)\right)$$

Obtener los pesos para que la solución tenga un error nulo de forma analítica no suele ser posible. El objetivo es conseguir que sea el menor posible y para ello se pueden emplear ciertos algoritmos iterativos de optimización. A pesar de que existan opciones muy variadas, presenta una base en común que consiste en tomar la variación del error con respecto a la variación de los pesos como en un descenso de gradiente. Algunos de los algoritmos de optimización más utilizados son el descenso de gradiente o el método de Newton.

En las siguientes funciones w representa el conjunto de pesos, γ es el parámetro de aprendizaje y E el error correspondiente del resultado calculado:

- **Descenso de gradiente:**

$$\Delta w[t] = -\gamma\frac{\partial E_n}{\partial w[t]}$$

- **Método de Newton:**

$$\Delta w[t] = -\gamma\left(\frac{\partial^2 E_n}{\partial w[t]^2}\right)^{-1}\frac{\partial E_n}{\partial w[t]}$$

A lo largo de los siguientes apartados se van a utilizar modelos de redes neuronales ya entrenados, por ello se explica su estructura a continuación.

Modelo detección facial:

Para la detección de caras se ha optado por usar un modelo entrenado mediante la arquitectura *ResNet SSD*. *ResNet* proviene de *Residual Neural Network* y las siglas SSD significan *Single Show Multibox Detector*. Es una red neuronal que devuelve los vértices de un rectángulo que contendría la cara (*bounding box*) y la probabilidad de que ese recuadro contenga un rostro [59].

Lo que propone la arquitectura *ResNet* es incorporar saltos en las conexiones entre capas, es decir, permitir saltos en las conexiones de la red como se puede ver en la Fig. 39.

Con respecto al funcionamiento completo de la SSD, dada una imagen de entrada y un conjunto de etiquetas para un *dataset* o conjunto de entrada actúa de esta manera:

1) Pasa la imagen a través de una serie de capas convolucionales, produciendo mapas de las características a diferentes escalas, por ejemplo, 10×10, luego 6×6 y nuevamente 3×3, etc.

2) Por cada posición en cada uno de estos mapas de características, se usa un filtro convolucional de 3×3 para evaluar un pequeño conjunto de celdas delimitadoras. Éstas actúan de centro de un conjunto de rectángulos de distinta relación de aspecto que pueden contener un rostro.

3) En cada rectángulo se establece al mismo tiempo, la probabilidad de pertenecer a cada categoría.

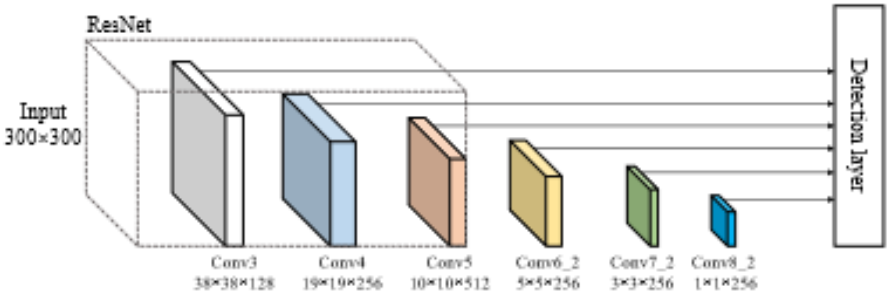

Fig. 38 Arquitectura *ResNet SSD*

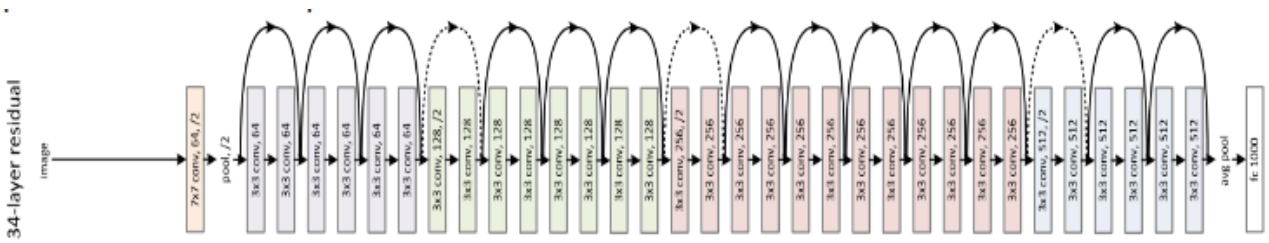

Fig. 39 Esquema arquitectura de la red *ResNet*

Modelo reconocimiento facial:

Para el reconocimiento facial se ha usado el modelo *FaceNet* cuya arquitectura es *Inception-ResNet v1 (ver* Fig. 40*)*. Es una red neuronal que aprende un mapeo de imágenes faciales en un espacio euclidiano compacto donde las distancias corresponden a una medida de similitud de caras. Es decir, cuanto más similares sean las dos imágenes faciales, menor será la distancia entre ellas [60].

La arquitectura *Inception* consiste en concatenar los resultados obtenidos de aplicar diferentes filtros de convolución (con características distintas, como el tamaño, entre otros,) y ventanas de *pooling* a una misma entrada como se puede ver en la Fig. 41. Esto permite al modelo beneficiarse de la extracción de características a múltiples niveles en un único paso.

FaceNet usa un método de pérdida denominado *Triplet Loss*. Éste minimiza la distancia entre el rostro a reconocer y un positivo, imágenes que contienen la misma identidad y maximiza la distancia entre un negativo, imágenes que contienen diferentes identidades [60].

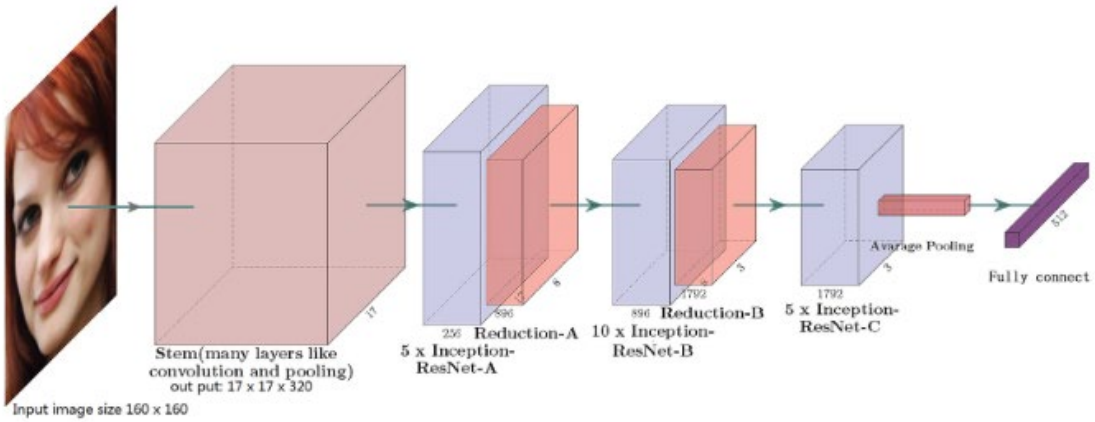

Fig. 40 Arquitectura *Inception-ResNet v1*

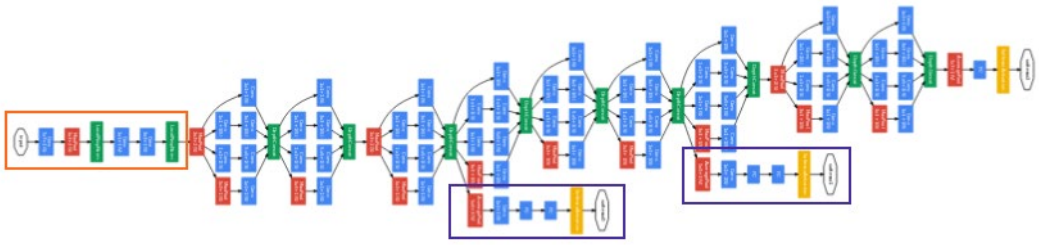

Fig. 41 Esquema de arquitectura la red *Inception*

Modelo reconocimiento de voz:

Por último, para el reconocimiento de voz se ha utilizado *DeepSpeech* que utiliza un sistema de aprendizaje automático bien optimizado basado en una red neuronal recurrente (RNN) tal y como se muestra en la Fig. 42.

Una red neuronal recurrente no tiene una estructura de capas definida, sino que permite conexiones arbitrarias entre las neuronas, incluso pudiendo crear ciclos, con esto se consigue crear la temporalidad, permitiendo que la red tenga memoria [61]. Las redes neuronales recurrentes convencionales presentan problemas en su entrenamiento debido a que el descenso del gradiente tiende a crecer enormemente o a desvanecerse con el tiempo debido a que el gradiente depende, no sólo del error presente, sino también de los errores pasados. La acumulación de errores provoca dificultades para memorizar dependencias a largo plazo. Estos problemas son solventados por las redes LSTM (*Long Short Term Memory*), para ello incorporan una serie de pasos para decidir qué información va a ser almacenada y cual borrada.

DeepSpeech (ver Fig. 42) consta de dos subsistemas: un modelo acústico y un decodificador. El modelo acústico utiliza métodos de aprendizaje automático profundo para calcular la probabilidad de la presencia de ciertos caracteres en el sonido de entrada. El decodificador usa un algoritmo de búsqueda que transforma las probabilidades de los caracteres en transcripciones textuales [62].

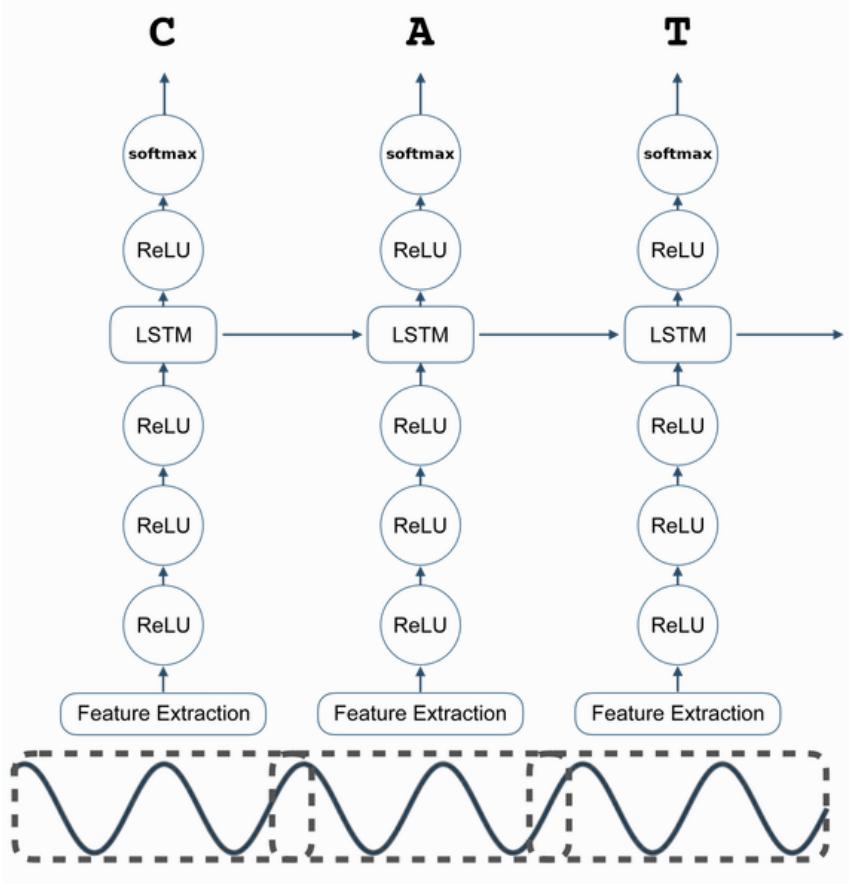

Fig. 42 Arquitectura *RNN DeepSpeech*

2) TensorFlow.

TensorFlow es una plataforma de código abierto para el Aprendizaje Automático [63]. Cuenta con un ecosistema integral y flexible de herramientas, bibliotecas y recursos de la comunidad, por ello se ha posicionado como la herramienta líder en el sector de *Deep Learning.*

Más concretamente, *TensorFlow* es una biblioteca de software de código abierto para computación numérica, que usa gráficos de flujo de datos. Los nodos en las gráficas representan operaciones matemáticas, mientras que los bordes de las gráficas representan las matrices de datos multidimensionales (tensores) comunicadas entre ellos. Es una gran plataforma para construir y entrenar redes neuronales, que permite detectar y descifrar patrones y correlaciones, análogos al aprendizaje y razonamiento usados por los humanos [64].

TensorFlow se construyó pensando en el código abierto y en la facilidad de ejecución y escalabilidad. Permite ser ejecutado en la Nube, pero también en modo local. La idea es que cualquiera pueda ejecutarlo. Se puede ver a personas que lo ejecutan en una sola máquina, un único dispositivo o en grandes clústeres. Y si realmente se desea escalar, la Nube es un gran lugar para hacerlo [64].

Por otra parte, además de *TensorFlow* se encuentra disponible *TensorFlow Lite*, la versión ligera de *TensorFlow* para dispositivos móviles, aplicaciones web o sistemas embebidos. El conversor de *TensorFlow Lite* toma un modelo de *TensorFlow* y genera un modelo de *TensorFlow Lite* (ver Fig. 43).

Los dispositivos embebidos a menudo tienen memoria o potencia computacional limitada. Se pueden aplicar varias optimizaciones a los modelos para que se puedan ejecutar dentro de estas restricciones. La cuantificación posterior al entrenamiento es una técnica de conversión que puede reducir el tamaño del modelo y al mismo tiempo mejorar la latencia del acelerador de CPU y hardware con poca degradación en la precisión del modelo [65].

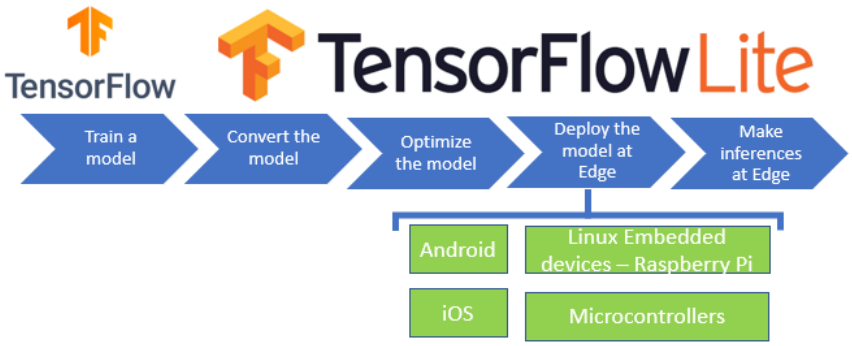

Fig. 43 Proceso de conversión a *TensorFlow Lite*.

El modelo para el reconocimiento facial no se encontraba en el formato de *TensorFlow Lite,* extensión *".tflite"* por lo que se ha transformado y se ha utilizado una cuantificación de rango dinámico que ha permitido reducir el modelo en cuatro veces su peso. Se ha usado el código siguiente:

```
import tensorflow as tf

saved_model_dir = '/content/facenet_keras.h5'
model = tf.keras.models.load_model(saved_model_dir)

# Convert the model
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

# Save the model.
with open('facenet.tflite', 'wb') as f:
  f.write(tflite_model)
```

Fragmento de código 1 Transformación de *TensorFlow a TensorFlow Lite*

3.3.3 Componente para reconocimiento facial.

Para el reconocimiento facial, además de TensorFlow se ha utilizado la librería *OpenCV* que es una biblioteca de código abierto de visión por computador, análisis de imagen y aprendizaje automático y dispone de una gran cantidad de algoritmos [66]. Se ha optado por usar esta librería porque existe un paquete en *ROS (vision_opencv*) que permite convertir las imágenes de los mensaje de *ROS* a *OpenCV* y viceversa [67].

Además, *OpenCV* permite trabajar con redes neuronales, por lo que en la detección de caras se ha usado directamente este modo. Se ha utilizado un modelo *Caffe* que también es un *framework* de aprendizaje profundo muy utilizado en Visión Artificial [68]. Los métodos de la librería que se han utilizado se describen en la Tabla 4.

Tabla 4 Métodos utilizados de la librería OpenCV

`cv2.dnn.readNetFromCaffe(config_path, model_path)`	➔ Se usa para configurar la red. Requiere un archivo *.caffemodel* donde están los pesos de las diferentes neuronas, y un archivo *.prototxt* que describe la arquitectura de la red.
`cv2.dnn.blobFromImage()`	➔ Se usa para preprocesar una imagen.
`setInput(image)`	➔ Transfiere la imagen al inicio de la red.
`forward()`	➔ Permite realizar la inferencia.
`cv2.resize()`	➔ Se usa para cambiar el tamaño de una imagen.
`cv2.imshow()`	➔ Se utiliza para mostrar imágenes por pantalla.

Por otro lado, se ha usado la librería de *TensorFlow* para poder realizar la inferencia del modelo *FaceNet* que se convirtió a *TFLite,* en particular los método definidos en la Tabla 5.

Tabla 5 Métodos utilizados de la librería *TensorFlow*

`tf.lite.Interpreter(model_path)`	➔ Se usa para para cargar el modelo de *TfLite.*
`allocate_tensors()`	➔ Asigna los tensores.
`get_input_details()`	➔ Devuelve los detalles del tensor de la capa de entrada de la red.
`get_output_details()`	➔ Devuelve los detalles del tensor de la capa de salida de la red.
`set_tensor()`	➔ Transfiere el tensor a la capa indicada. Se usará para pasar la imagen al inicio de la red.
`invoke()`	➔ Inicia el intérprete, para calcular la predicción.
`get_tensor()`	➔ Permite obtener el tensor de la capa indicada. Se usará para obtener la predicción.

El proceso de reconocimiento facial se ha dividido en dos partes, la detección facial y el reconocimiento como tal.

1) Detección facial

Se debe preprocesar la imagen que proporciona la cámara dado que la entrada del modelo de aprendizaje profundo utiliza tensores 4-D, con una dimensión 300x300 como muestra la Fig. 38. Además, se aconseja realizar la resta media de los valores (104, 177, 123) en los canales azul, verde y rojo de la imagen respectivamente para obtener mejores resultados [69].

Ahora ya se puede calcular la inferencia sobre la imagen, esta devolverá los vértices de los diferentes rectángulos y la probabilidad de que esos rectángulos contengan una cara. Se ha seleccionado un umbral de probabilidad de 0.9 (el máximo es 1) y se recortará la imagen que sea mayor entre todas las que identifique y que supere un cierto tamaño. Éste dependerá de la cámara y lo que se busca es que seleccione la cara a reconocer cuando se esté en frente del robot.

```
self.__net = cv2.dnn.readNetFromCaffe(config_path, model_path)


def preprocess_image(self, image):
     self.__height = image.shape[0]
     self.__width = image.shape[1]
     blob = cv2.dnn.blobFromImage(cv2.resize(image, (300, 300)), 1.0,
(300, 300), (104.0, 177.0, 123.0))


     return blob


def run_inference(self, blob):
     self.__net.setInput(blob)
     inference = self.__net.forward()


     return inference
```

Fragmento de código 2 Librería detección facial

2) Reconocimiento facial

Una vez que se ha obtenido la cara recortada comienza el paso de reconocimiento facial. De nuevo, hay que preprocesar la imagen ya que el modelo presenta una estructura diferente. En primer lugar, se ha cambiado el tamaño de la imagen porque la primera capa requiere tensores 4-D con una dimensión de 160x160 como se observa en la Fig. 41 Esquema de arquitectura la red *Inception*. Y en segundo lugar, se ha normalizado restando la media y dividiendo entre la desviación típica de la imagen.

Una vez preprocesada la imagen se puede proceder a realizar la inferencia. En este caso el resultado que se obtiene es un vector de características (*embeddin*g), de 512 elementos. El siguiente paso es calcular la distancia (se ha utilizado la definición euclídea) entre la inferencia de las imágenes de referencia con respecto a la inferencia de la imagen a reconocer. El umbral de la distancia para dar una cara como reconocida se toma en 1.24 tal y como se indica en [60].

```
self.__interpreter = tf.lite.Interpreter(model_path=model_path)
self.__interpreter.allocate_tensors()
self.__input_details = self.__interpreter.get_input_details()
self.__output_details = self.__interpreter.get_output_details()

def preprocess_image(self, image, required_size=(160, 160)):
    ret = cv2.resize(image, required_size)
    ret = ret.astype('float32')
    mean, std = ret.mean(), ret.std()
    ret = (ret - mean) / std

    return ret

def run_inference(self, image):
    input_data = np.expand_dims(image, axis=0)
    self.__interpreter.set_tensor(self.__input_details[0]['index'],
input_data)
    self.__interpreter.invoke()
    predictions =
self.__interpreter.get_tensor(self.__output_details[0]['index'])[0]

    return predictions

def faces_distance(self, emb_ref, emb_desc):
    distancia = np.linalg.norm(emb_ref - emb_desc)

    return distancia
```

Fragmento de código 3 Librería reconocimiento facial

3.3.4 Componente para reconocimiento de voz.

Para el reconocimiento de voz se ha usado directamente la librería de *DeepSpeech* que permite el uso de modelos de TensorFlow Lite. El modelo que incorpora la librería está en inglés, sin embargo es posible encontrar modelos en otros idiomas aunque no sean tan precisos. En concreto existe un repositorio denominado

DeepSpeech-Polyglot de donde se ha obtenido el modelo en español el cual tiene 660 horas de entrenamiento [71].

Los métodos de la librería que se han utilizado se describen en la Tabla 6.

Tabla 6 Métodos utilizados de la librería *DeepSpeech*

`Model(model_path)`	➔ Se usa para cargar el modelo
`createStream()`	➔ Permite comenzar la inferencia del audio.
`feedAudioContent(audio_buffer)`	➔ Proporciona el audio para realizar la inferencia, la señal de audio deber de ser de 16 bits mono raw.
`finishStream()`	➔ Calcula la decodificación final de una inferencia de transmisión continua y devuelve el resultado.

Ahora bien, para poder pasar de audio a texto primero hay que capturar el audio. Para ello se ha utilizado otra librería denominada *PyAudio*, en particular los métodos descritos en la Tabla 7.

Tabla 7 Métodos utilizados de la librería *PyAudio*

`PyAudio()`	➔ Se usa para crear un objeto de la clase *PyAudio*.
`open()`	➔ Permite configurar la captura del audio mediante una serie de parámetros: la tasa de muestreo, el número de canales, subrutinas, etc.
`start_stream()`	➔ Inicia la captura de audio.
`is_active()`	➔ Devuelve si la captura de audio sigue activa.
`stop_stream()`	➔ Para la captura de audio.
`close()`	➔ Cierra la captura de audio.
`terminate()`	➔ Se usa para eliminar un objeto de la clase PyAudio.

El proceso de transcripción de voz a texto se da a lugar una vez que se ha reconocido a una persona. El primer paso consiste en iniciar la captura de audio que estará ya configurada con los parámetros correspondientes. Además, dispondrá de una subrutina que permita transformar la señal de audio en el formato que necesita el modelo de aprendizaje profundo.

Conforme se va capturando el audio, se va realizando la inferencia y, por tanto, la transcripción se va a producir de forma continua, hasta que se finalice dicha captura.

```
self.__model = deepspeech.Model(model_path)

def process_audio(self,in_data, frame_count, time_info, status):
     data16 = np.frombuffer(in_data, dtype=np.int16)
     self.__context.feedAudioContent(data16)

     return (in_data, pyaudio.paContinue)

def run_inference(self):
     while self.__stream.is_active():
          time.sleep(0.1)
          a = input()
          if a == 's':
               self.__stream.stop_stream()
               self.__stream.close()
               self.__audio.terminate()
               print('Finished recording.')
               text = self.__context.finishStream()
               print('Final text = {}'.format(text))
               break

     return text
```

Fragmento de código 4 Librería transcripción de voz a texto

3.3.5 Componente para síntesis de voz.

Para la síntesis de voz se ha utilizado una librería llamada pyttsx3. Utiliza para su motor los paquetes de idiomas que están instalados en el sistema operativo, por lo que tiene un sonido natural, a diferencia de otros sintetizadores de voz. Además, se ha complementado con otra librería denominada talkey que reduce la programación a un par de líneas tal y como se muestra en el siguiente código:

```
tts = talkey.Talkey()
tts.say("No le reconozco, pruebe de nuevo", 'es')
```

Fragmento de código 5 Síntesis de voz a partir de texto

3.3.6 Componente de reconocimiento de lenguaje natural.

Una vez realizada la transcripción de voz a texto, ya es posible analizar la frase. Para ello se ha utilizado la librería *spacy-udpipe*, que permite usar los modelos de NLP de *UDPipe* que son más rápidos y eficientes en el pipeline de procesamiento de *Spacy*. Al querer analizar frases en español, se debe cargar uno de los modelos de ese idioma, en concreto se ha cargado *spanish-gsd-ud-2.5-191206.udpipe* (*es-gsd*). En la Tabla 8 se muestran los métodos de la librería que se han empleado:

Tabla 8 Métodos utilizados de la librería *spacy-udpipe*

`spacy_udpipe.load(model name)`	➔ Se usa para crear un objeto de la clase spacy del idioma seleccionado como parámetro.
`token.pos_`	➔ Permite acceder al atributo del etiquetado gramatical de una palabra
`token.lemma_`	➔ Permite obtener la raíz semántica de una palabra.

El procedimiento para analizar una frase es simple, tan solo hay que crear un objeto de la clase *Spacy* y pasarle el texto que se desee interpretar. Lo que hace este objeto es descomponer la frase en palabras (*tokens)* los cuales tienen una serie de atributos que se usarán para definir una serie de reglas lógicas.

En el contexto de este trabajo, las tareas a realizar por el robot están relacionadas con el ámbito de la navegación autónoma, por tanto, antes de definir dichas tareas, es necesario analizar una serie de frases comunes usadas para expresar movimiento y localización.

¿Dónde está el despacho?

Dónde → Adverbio Está → Verbo El → Determinante Despacho → Nombre

Lleva el paquete al despacho de Juan.

Lleva → Verbo El → Determinante Paquete → Nombre Al (a + el) → Preposición + determinante

Despacho → Nombre De → Preposición Juan → Nombre propio

Quiero que traigas el libro de secretaría.

Quiero → Verbo Que → Conjunción Traigas → Verbo El → Determinante Libro → Nombre

De → Preposición Secretaría → Nombre

De estas tres frases se puede observar que:

- Si la oración empieza por adverbio, el destino vendrá precedido de un determinante.

- En caso contrario, el destino vendrá precedido de una preposición o de una preposición junto a un artículo.
- En caso de que haya un complemento directo, se identificará porque va precedido de un determinante al que no acompaña una preposición.

Mediante esas tres proposiciones ya se pueden deducir una serie de reglas lógicas que permiten obtener el destino, el objeto y el verbo. Este último en forma infinitiva mediante el atributo que devuelve la raíz semántica. Por último hay que tener en cuenta que hay verbos que implican ir a un lugar y volver y otros que indican que hay que ir solamente.

```
self.__model = spacy_udpipe.load("es-gsd")


def analyze_text(self, text):
      verbo = objeto = lugar = ""
      dic = {}
      doc = self.__model(text)
      for token in doc:
            dic[token] = token.pos_
      for i in range (0, len(dic)):
            if i == 0:
                  if list(dic.values())[i] == 'ADV': donde = True
                  else: donde = False
            if list(dic.values())[i] == 'VERB':
                  verbo = str(list(dic.keys())[i].lemma_)
            if list(dic.values())[i] == 'PROPN':
                  lugar += str(list(dic.keys())[i])
            if list(dic.values())[i] == 'NOUN' and donde == True:
                  lugar += str(list(dic.keys())[i])
            elif list(dic.values())[i] == 'NOUN' and donde == False:
                  if list(dic.values())[i-2] == 'ADP' or
list(dic.values())[i-1] == 'ADP':
                        lugar = str(list(dic.keys())[i])
                  else: objeto = str(list(dic.keys())[i])


      return verbo, objeto, lugar
```

Fragmento de código 6 Librería procesamiento de lenguaje natural

3.3.7 Componente para navegación.

Para la navegación se ha usado el paquete que proporciona *ROS*. En su conjunto, se trata de una serie de algoritmos que gracias a la información que proporcionan los sensores del robot y su odometría (uso de datos de sensores de movimiento para estimar la posición relativa de un robot móvil) consiguen controlar el robot mediante mensajes estándares. El paquete puede mover el robot evitando problemas como colisiones, quedarse atascado o desorientarse de su posición [72]. Sin embargo son necesarios unos requerimientos para poder usarlo:

- Este paquete sólamente funciona con robots con tracción diferencial y holomónicos.
- Es necesario que el robot tenga montado un láser en su base. Como alternativa se puede usar otros sensores equivalentes, por ejemplo un sónar.
- La precisión en la navegación es mejor en aquellos robots cuya estructura es circular o cuadrada.

En la Fig. 44 se puede ver la organización del paquete de navegación. Se observan tres tipos de color de recuadros, blanco, azul y gris. Los recuadros blancos son todos los nodos que proporciona *ROS* y hacen al robot autónomo, los azules son los nodos del robot que lo dotan de movimiento y extraen información del entorno y los grises son nodos adicionales para el funcionamiento.

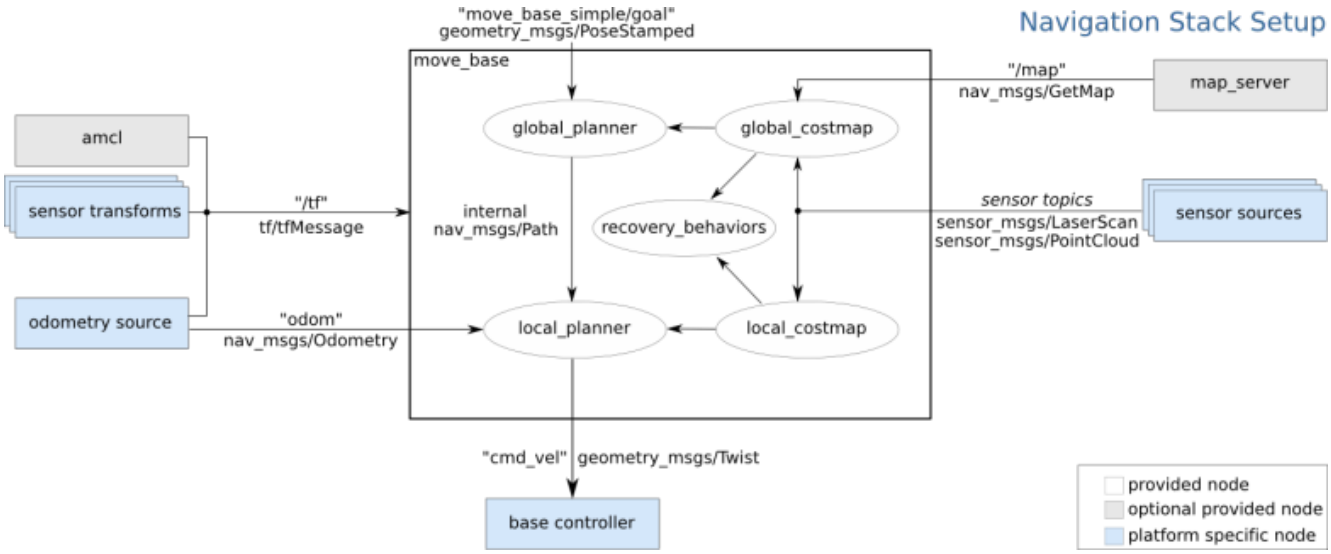

Fig. 44 Paquete de navegación de ROS

A continuación se describirán brevemente los distintos nodos:

1) Transformación de medidas.

Se encarga de que el robot publique la información de los sensores respecto un mismo marco de referencia del robot usando el paquete */tf*. Los marcos de referencia de los robots normalmente se llaman */base_link* o *base_footprint*, dependiendo de si el punto de origen de sus ejes está en el centro de masas de la base móvil o su proyección en el suelo, respectivamente [73].

2) Información de los sensores.

Es utilizada para evitar obstáculos móviles en el entorno de trabajo. Para ello los sensores deben publicar la información en mensajes del tipo sensor *msgs/LaserScan* o sensor *msgs/PointCloud2*, para representar datos de escáneres 2D o nubes de puntos 3D, respectivamente. Hay muchos drivers de ROS para distintos tipos de sensores que adaptan su toma de datos a este tipo de mensajes. En caso de que no haya un *driver* adecuado se debe implementar un nodo que reciba esa información y la transforme [73].

3) Información de odometría.

Es requerida para que sea publicada usando tf y los mensajes tipo *nav_msgs/Odometry*. De esta manera se puede conocer de forma estimada la posición y la velocidad del robot. Esta información se obtiene a partir de los encoders de las ruedas que permiten calcular cuánto ha avanzado el robot en función de su giro y a qué velocidad están girando, y a través del giroscopio se puede saber cuánto ha girado el robot desde su arranque [73].

4) Controlador de la base del robot.

Puede enviar comandos de velocidad usando el mensaje *geometry_msgs/Twist* respecto la referencia de la base del robot. El *topic* más común donde se publica la velocidad es *cmd_vel* aunque dependiendo del robot puede cambiar. Para traducir los valores de velocidad angular y lineal a comandos para los motores del robot es necesario estar suscrito a ese *topic.* [73].

5) Servidor del mapa.

No es obligatorio pero siempre se asume que se tiene uno. Permite conocer las limitaciones del entorno y los obstáculos fijos. La implementación actual del servidor del mapa convierte los valores de color de la imagen del mapa en valores de ocupación ternarios: libre (0), ocupado (100) y desconocido (-1) [73].

6) Paquete amcl.

AMCL (del inglés, *Adaptative Monte Carlo Localization)* es un algoritmo de localización denominado localización adaptativa de Monte Carlo. Necesita un mapa escaneado, los datos en tiempo real de los sensores y los mensajes de transformación para dar como salida la posición estimada del robot. Al ejecutar el algoritmo se iniciará un filtro de partículas en donde cada partícula representa una posible posición del robot en función de su probabilidad [73].

7) Configuración común.

El paquete de navegación utiliza mapas de costeo (*costmaps)* para almacenar información de los obstáculos en el escenario. Con el fin de hacer esto apropiadamente es necesario guardar los *costmaps* desde los *topics* de los que se está escuchando para que vayan siendo actualizados. Estos parámetros son comunes en el *global costmap* y en el *local costmap*, y el archivo que contiene esta información es el *costmap_common_params.yaml*. En éste se configuran parámetros como la geometría del robot, la longitud de detección de obstáculos y las fuentes de detección de los obstáculos [73].

8) Configuración global.

Esta configuración se utiliza en el mapa de coste de la navegación global. Crea planes de trayectoria a largo plazo sobre el entorno completo. Aquí se configura el *topic* donde se obtiene el mapa del entorno, la frecuencia a la que se actualizará el mapa y hacia que marco de referencia se transformarán las coordenadas del mapa. El archivo de configuración es *global_costmap_params.yaml* [73].

9) Configuración local.

En esta configuración se establecen los parámetros del mapa de coste utilizado en la navegación local, encargada de evitar obstáculos móviles que estén alrededor del robot. En este archivo se configura el *topic* donde se obtiene la posición estimada del robot en el entorno, la frecuencia a la que se actualizará el mapa y hacia que marco de referencia se transformarán las coordenadas del mapa, las dimensiones del mapa local y su resolución. El archivo de configuración se denomina *local_costmap_params.yaml* [73].

10) Planificador local.

Es el encargado de calcular comandos de velocidad para enviárselos a la base móvil del robot. Por defecto utiliza el algoritmo DWA, explicado en la sección 2.2.1. El archivo de configuración es *base_local_planner params.yam*l donde se pueden modificar las velocidades y aceleraciones máximas que tendrá el robot [73].

11) Planificador global.

Se encarga de trazar la trayectoria que seguirá el robot desde su posición actual hasta su objetivo. El planificador tiene implementados el algoritmo de Dijkstra y el A*, pero por defecto usa el algoritmo de Dijkstra ya que se obtienen mejores resultados [73].

12) Estrategias de recuperación.

En esta configuración se establen los parámetros de las estrategias relacionados con la recuperación del robot en caso de que no pueda alcanzar su objetico. Algunos de las estrategias son rotar sobre sí mismo, limpiar el mapa local de los obstáculos detectados, entre otros. El archivo se denomina *recovery_params.yaml* [73].

3.4 Integración de todos los componentes.

Una vez conocido el funcionamiento de los distintos sistemas, es necesario integrarlos. Para ello es clave *ROS*, cuyo punto fuerte es la estandarización y *Gazebo* que permite simular el comportamiento de un robot en un entorno 3D tal como se ha visto en el sección 3.3.1.

Dado que debe existir una sincronización entre los diferentes componentes, se ha optado por implantar el modo de comunicación cliente-servidor entre los nodos de ROS, cuyo funcionamiento se puede ver en el flujograma de la Fig. 45.

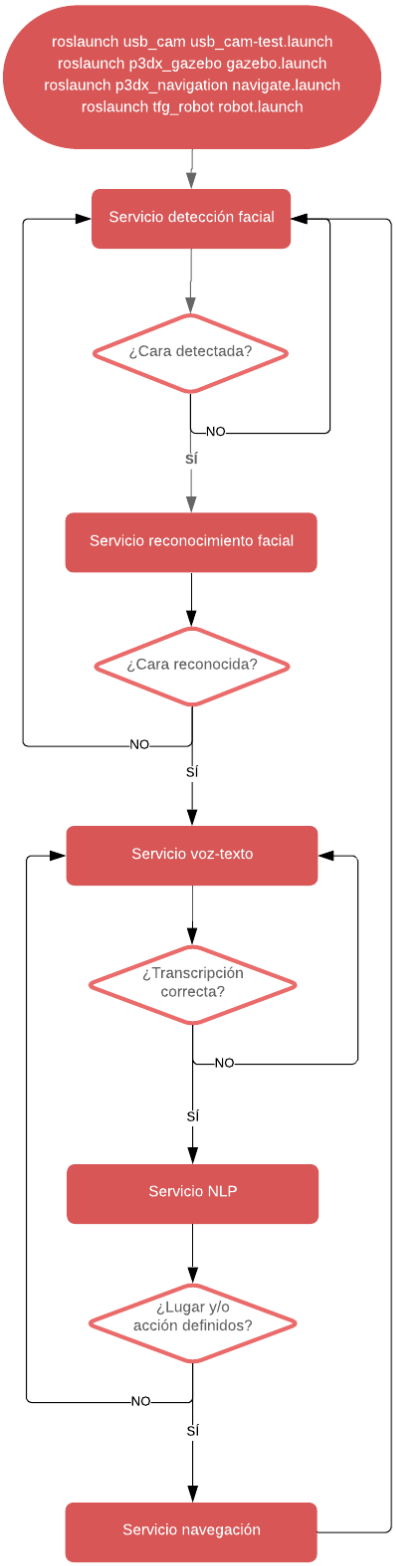

Fig. 45 Flujograma de funcionamiento del sistema

Los archivos que se deben ejecutar para el funcionamiento del sistema son los que se describen en la Tabla 9.

Tabla 9 Comandos para la puesta en marcha del sistema.

`roslaunch usb_cam usb_cam-test.launch`	➔ Inicia la comunicación con la cámara.
`roslaunch p3dx_gazebo gazebo.launch`	➔ Inicia la simulación en gazebo.
`roslaunch p3dx_navigation navigate.launch`	➔ Inicia las configuraciones del paquete de navegación
`roslaunch tfg_robot robot.launch`	➔ Inicia todos los nodos que se han creado para integrar el sistema.

A continuación se van a describir brevemente el funcionamiento de los diferentes nodos:

1. Nodo cliente.

Se podría considerar el nodo de sincronización. Se encarga de iniciar la comunicación con los distintos nodos servidores.

```
rospy.Subscriber('face/detect', Bool, self.detection_callback)


def detection_callback(self,msg):
      self.detected = msg.data
      if self.detected == True:
            self.id = self.start_client_face()
            if self.id != 'Unknow':
                  while self.flag == False:
                        sentence = self.start_client_listen())
                        resp2 = self.start_client_analyze(sentence)
                        if resp2[0] in self.places and resp2[1] in
self.actions:
                              self.flag = True
                        else:
                              self.speaker(self.id)
                  destino = resp2[0]
                  volver = resp2[2]
                  self.start_client_goal(destino, volver)
                  self.flag = False
```

Fragmento de código 7 Nodo cliente

2. Nodo detección facial.

Como se ha indicado en el aparatado 3.2, se ha trabajado con hardware doméstico, luego para la detección facial se ha utilizado la cámara web ya descrita. Se ha usado un nodo que publica las imágenes en el siguiente *topic usb_cam/image_raw* en el cual está suscrito el nodo de detección facial.

Para poder realizar el proceso de detección facial es necesario realizar la conversión del mensaje publicado en ese *topic* al formato de imágenes *OpenCV*, para ello se utiliza la librería *cv_bridge*. Una vez realizado el proceso de detección (ver sección 3.3.3 1)), se pueden dar dos opciones:

- Que no haya detectado ningún rostro, en tal caso se publicará en el *topic face/detect* un *False*.
- Que sí se detecte una cara, luego se publicará en el *topic face/detect* un *True* y se transformará la imagen de la cara recortada al formato ROS para poder publicarla en el *topic face/cropped*.

```
self.__sub = rospy.Subscriber('usb_cam/image_raw', Image,
self.ImageCallback)


def ImageCallback(self,msg):
     try:
          bridge = CvBridge()
          img = bridge.imgmsg_to_cv2(msg, "bgr8")
          img_preprocessed = self.__ofd.preprocess_image(img)
          inference = self.__ofd.run_inference(img_preprocessed)
          face = self.__ofd.cropp__front_face(inference, img)
          if face.size != 0:
               imgMsg = bridge.cv2_to_imgmsg(face, "bgr8")
               self.__result_pub.publish(imgMsg)
               self.__result_pub2.publish(True)
          else:
               self.__result_pub2.publish(False)
     except Exception as err:
          print err
```

Fragmento de código 8 Nodo detección facial

3. Nodo servidor de reconocimiento facial.

En el caso de que se haya detecto un rostro, se inicia el nodo de reconocimiento facial que es cliente del topic face/cropped. De nuevo se debe realizar la conversión de las imágenes para poder preprocesarlas.

Una vez realizado el proceso de reconocimiento facial (ver sección 3.3.3 2)), pueden ocurrir dos casos:

- No se reconozca la persona, en tal caso mediante la función que sísntesis de voz, se reproducirá que no reconoce la persona. El nodo se quedará a la espera de recibir otra imagen.
- Sí se reconozca a la persona, por lo que se reproducirá que se ha reconocido la persona y se devolverá su nombre como respuesta al servicio.

```
rospy.Service('face_recognition', Name, self.ImageCallback)


def ImageCallback(self, req):
    print('ImageCallback')
    msg = rospy.wait_for_message('face/cropped', Image)
    try:
        bridge = CvBridge()
        img = bridge.imgmsg_to_cv2(msg, "bgr8")
        img_preprocessed = self.__ofr.preprocess_image(img)
        inference = self.__ofr.run_inference(img_preprocessed)
        id = self.__ofr.compare_faces(inference)
        self.speaker(id)
     except Exception as err:
         print err
     return NameResponse(str(id))
```

Fragmento de código 9 Nodo reconocimiento facial

4. Nodo servidor de voz a texto.

Su funcionamiento es simple, se encarga de capturar el audio hasta que se presiones la tecla s. A continuación se realizará la inferencia (ver sección 3.3.4) que devolverá el resultado y se reproducirá. En el caso de que la reproducción no coincida con la comunicación, se volverá a capturar el audio hasta que se confirme que es correcto. Una vez confirmado devolverá la frase transcrita como respuesta al servicio.

```
rospy.Service('listening', Name, self.listen)


def listen(self, req):
    print('ListenCallback')
    a = 'n'
    while a == 'n':
          #Listen audio
          self.__ol.listen_audio()
          #Run interference
          text = self.__ol.run_inference()
          self.speaker(text)
          a = input()
    return NameResponse(text)
```

Fragmento de código 10 Nodo transcripción de voz a texto

5. Nodo de procesamiento de lenguaje natural.

Su objetivo es realizar el análisis (ver sección 3.3.6) de la frase obtenida del nodo anterior. Una vez analizada, devolverá el destino, la acción a realizar y si tiene que volver como respuesta al servicio.

```
rospy.Service('analyzing', Sentence, self.AnalyzeText)


def AnalyzeText(self,req):
    print('AnalyzeCallback')
    text = req.lis
    text = text + '.'
    verbo, objeto, lugar = self.__oa.analyze_text(text)
    destino, accion, volver = self.__oa.assigment(verbo, objeto, lugar)
    return SentenceResponse(unidecode.unidecode(destino),
unidecode.unidecode(accion), volver)
```

Fragmento de código 11 Nodo procesamiento de lenguaje natural

6. Nodo de navegación.

Gracias a la configuración previa del paquete de navegación de *ROS* (ver sección 3.3.7) es posible indicarle al robot la posición del objetivo que se quiere alcanzar. Los propios nodos del paquete se encargarán de mandarle las consignas de velocidad y las estrategias para evitar los posibles obstáculos que aparezcan en la trayectoria hacia dicho objetivo.

Los destinos que proporciona el nodo anterior están previamente guardados en este nodo, según sus coordenadas en el mapa, en formato de cuaterniones. En el caso de que tenga que realizar un movimiento de ida y vuelta, debe guardar la posición inicial.

```
rospy.Service('sending_goal', Goal, self.movebase_callback)


def movebase_callback(self, req):
     goal = req.dest
     volver = req.volv
     if volver == True:
          current_x, current_y, current_z, current_w = self.get_pose()
     self.identify_goal(goal)
     self.movebase_client(self.__coordinate_x, self.__coordinate_y,
self.__orientation_z, self.__orientation_w)
     if volver == True:
          self.movebase_client(current_x, current_y, current_z,
current_w)
     return GoalResponse()
```

Fragmento de código 12 Nodo de navegación

4 Análisis de resultados.

En el desarrollo de este capítulo se mostrará en qué consisten las pruebas que se han llevado a cabo y los resultados que se han obtenido. Como se ha dividido el trabajo en diferentes módulos, se seguirá el orden del flujograma de la Fig. 45. Además se incluirá una pequeña imagen del funcionamiento completo del sistema.

4.1 Pruebas diseñadas.

Para realizar las pruebas se ha configurado un entorno virtual para *Python* en el que es posible instalar paquetes sin interferir con el sistema. De esta forma, se pueden realizar los ensayos y una vez que se han obtenido buenos resultados, proceder a instalar los paquetes en el sistema. Además, al haber creado librerías propias, se pueden usar tanto en el entorno virtual como en la integración.

Por otro lado se ha utilizado *Colab*, que es un servicio *Cloud* de *Google*, basado en los notebooks de J*upyter*. Permite ejecutar y programar en Python desde el navegador [74], [75] con las siguientes ventajas:

- No requiere configuración.
- Da acceso gratuito a *GPU*s.
- Permite compartir contenido fácilmente.

A continuación se van a explicar las pruebas diseñadas para cada módulo y los resultados obtenidos.

4.1.1 Pruebas de detección facial y resultados.

La primera prueba consiste en detectar varias caras en una imagen y recortarlas, el resultado se muestra en la Fig. 46 y en la Fig. 47.

Fig. 46 Detección varios rostros

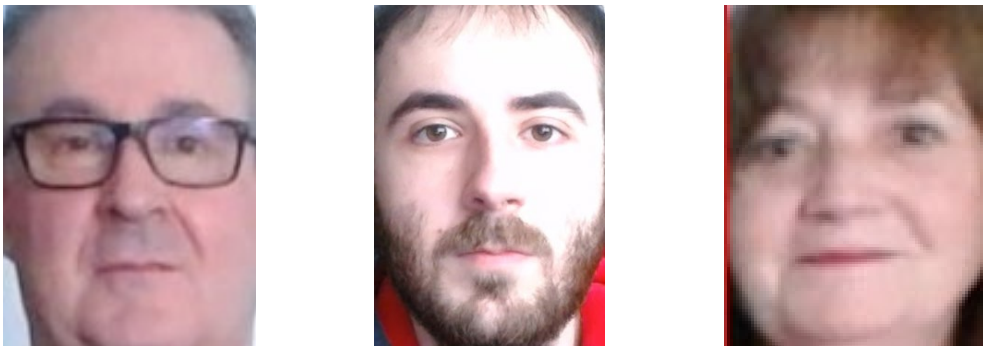

Fig. 47 Recorte varias caras

La segunda prueba busca que sólo se detecte el rostro más cercano a la cámara, a continuación la Fig. 48 muestra el resultado.

Fig. 48 Detección de rostro más cercano

La última prueba se basa en detectar la cara cambiando la iluminación y con una mascarillas, los resultados se muestran en la Fig. 49 y en la Fig. 50.

Fig. 49 Detección de rostro con mascarilla

Fig. 50 Detección de rostro con cambios de iluminación

4.1.2 Pruebas de reconocimiento facial y resultados.

Respecto a las pruebas de reconocimiento facial, se quiere comprobar que el umbral de reconocimiento es 1.24 como indica el artículo [60]. Para ello se utilizarán tres imágenes patrón, y las imágenes anteriores. Los resultados se pueden ver en la Fig. 51 y en la Fig. 52.

Fig. 51 Imágenes patrón para el reconocimiento facial

Fig. 52 Imágenes de prueba para el reconocimiento facial

La Fig. 53 muestra, en detalle, las distancias calculadas.

```
Distancia Roberto_normal vs Roberto 0.8587091
Distancia Roberto_mascarilla vs Roberto 0.91425574
Distancia Roberto_oscura vs Roberto 0.90125954
Distancia Roberto_normal vs Manuel 1.2731004
Distancia Roberto_mascarilla vs Manuel 1.4313129
Distancia Roberto_oscura vs Manuel 1.531685996055603
Distancia Roberto_normal vs Nieves 1.2812655
Distancia Roberto_mascarilla vs Nieves 1.2896627
Distancia Roberto_oscura vs Nieves 1.3814552
```

Fig. 53 Resultado distancias para el reconocimiento facial

4.1.3 Pruebas de reconocimiento de voz y resultados.

Aquí se han realizado pruebas de transcripción de voz a texto en condiciones normales y en presencia de ruido, los resultados se muestran en la Fig. 54, en la Fig. 55, en la Fig. 56 y en la Fig. 57.

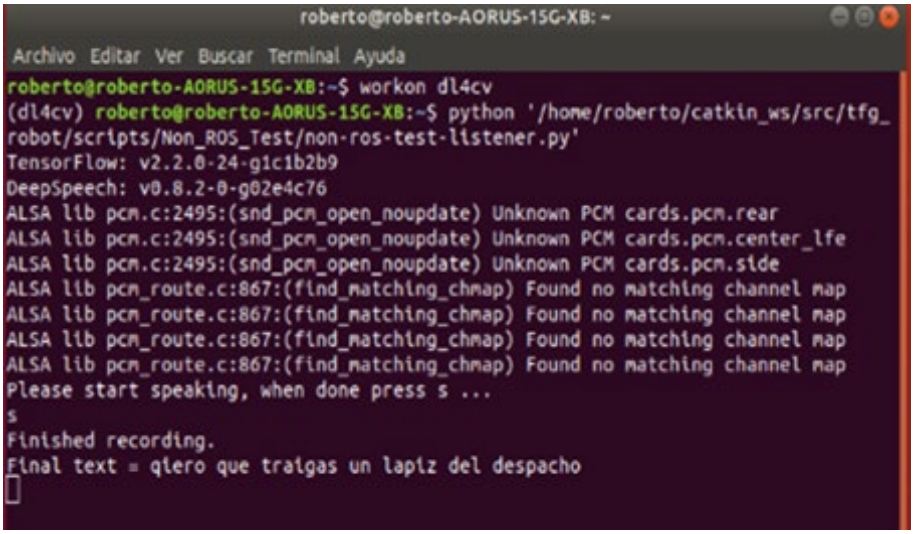

Fig. 54 Resultado transcripción de voz a texto Deepspeech (I)

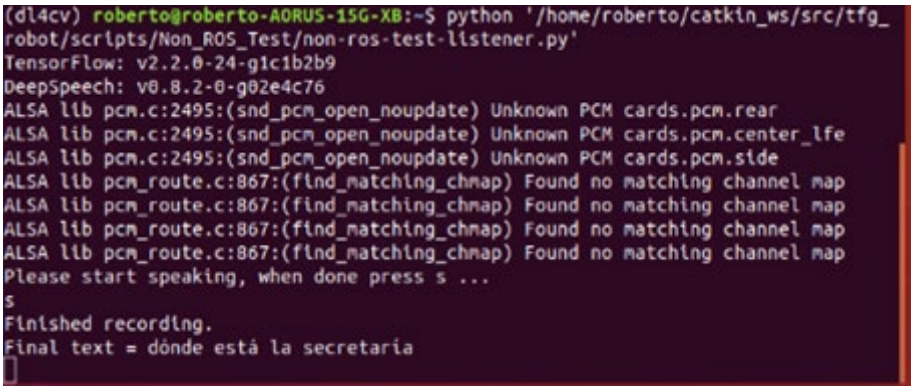

Fig. 55 Resultado transcripción de voz a texto Deepspeech (II)

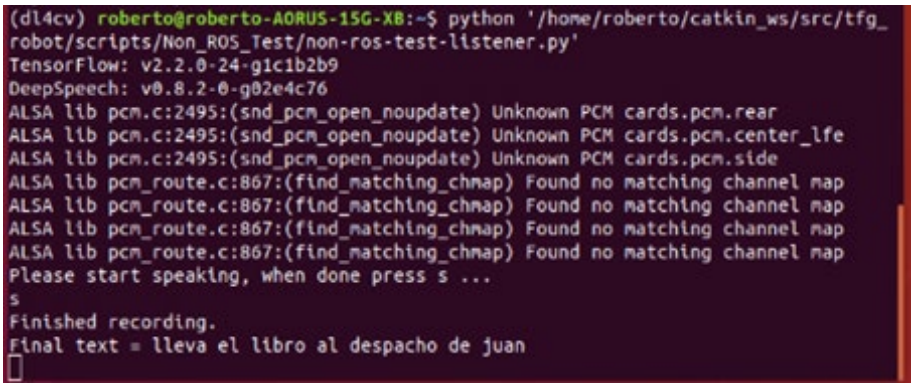

Fig. 56 Resultado transcripción de voz a texto Deepspeech (III)

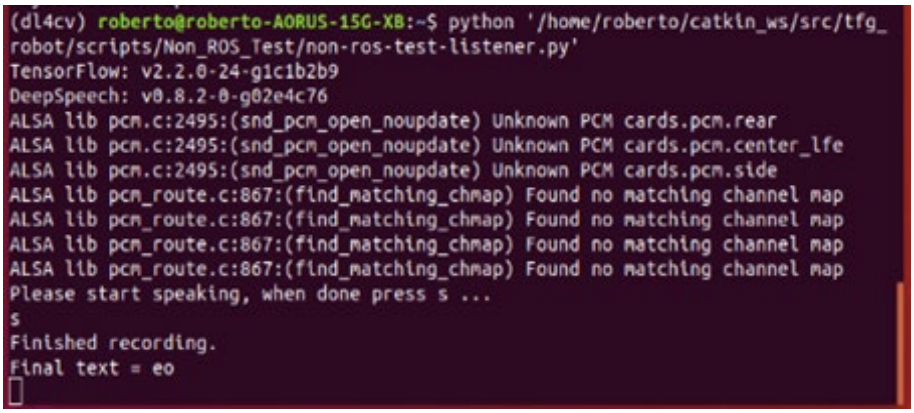

Fig. 57 Resultado transcripción de voz a texto Deepspeech ruido

También se ha utilizado la *API* que proporciona *Google* denominada *Speech-to-Text*. Para facilitar el proceso se ha trabajado directamente desde la propia web [76] aunque se podría importar la librería y utilizar sus métodos siempre y cuando se active el servicio y se lleve a cabo la autentificación con la clave de la API. Los resultados se muestran en la Fig. 58.

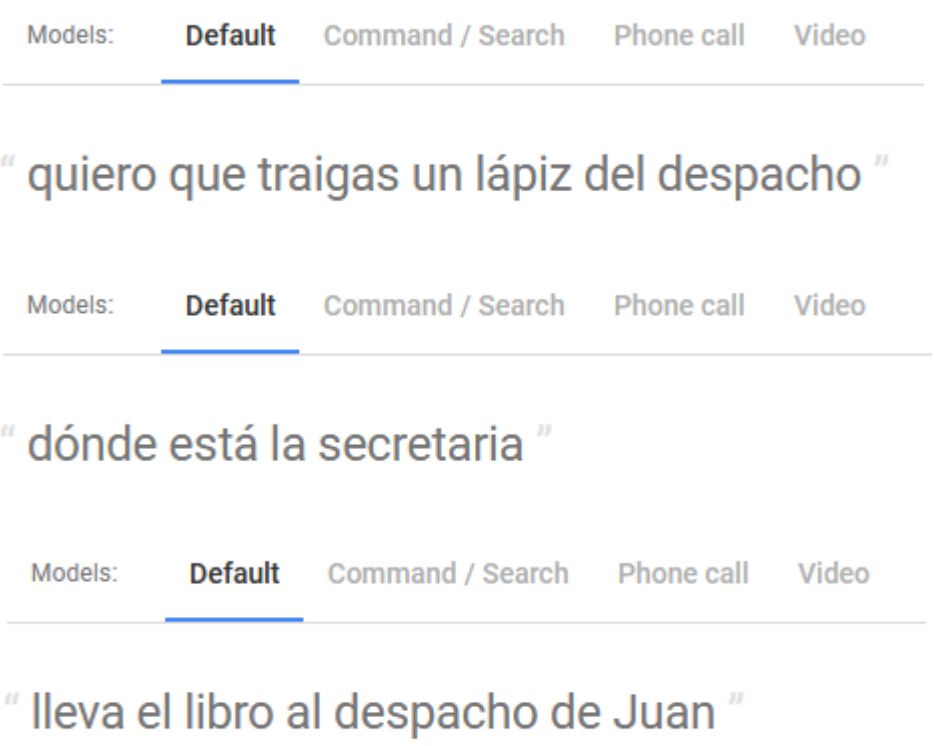

Fig. 58 Resultado transcripción de voz a texto *API Speech-to-Text* de *Google*

4.1.4 Pruebas de síntesis de voz y resultados.

Aquí tan solo se ha comprobado que el tono de voz que se ha usado para la síntesis de voz (propio del sistema) suene natural. Los resultados han sido satisfactorios. También se ha probado con otros sintetizadores *como Festival, flite* y *SVOX Pico* con peores resultados ya que se obtiene un tono de voz demasiado artificial.

4.1.5 Pruebas de reconocimiento de lenguaje natural y resultados.

En el caso de *NLP* se han realizado dos pruebas. En la primera se realiza el análisis gramatical y la obtención de la raíz semántica de una serie de frases como se puede observar en la Fig. 59.

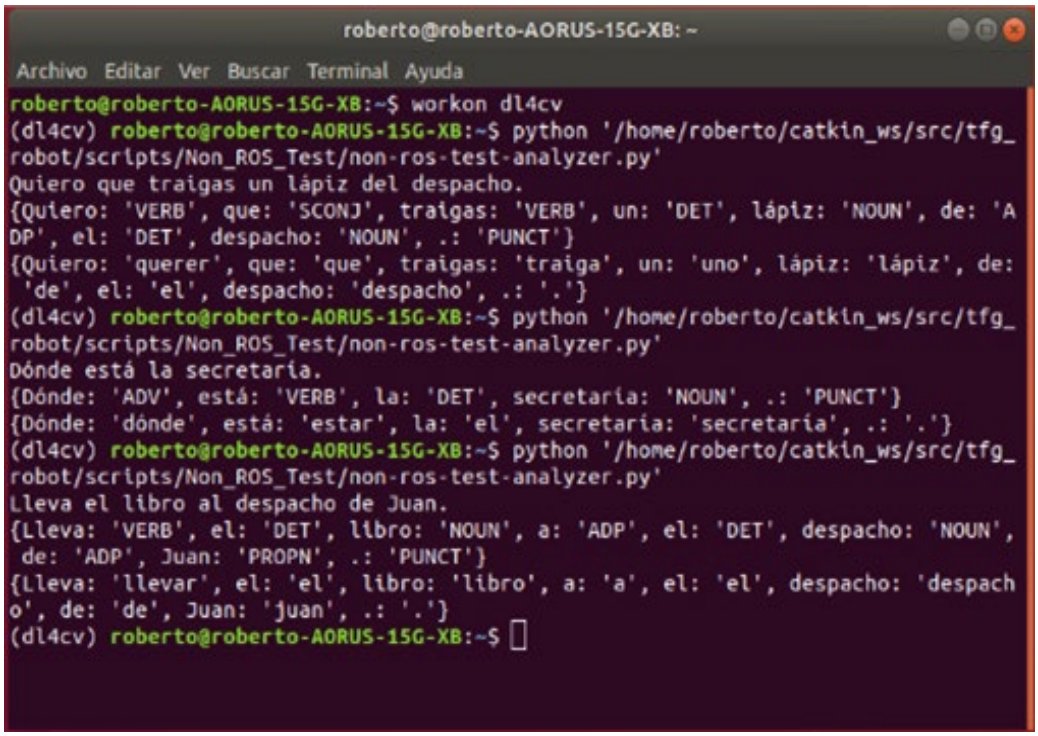

Fig. 59 Análisis gramatical SpaCy-UDPipe

De nuevo se ha trabajado con la *API* que proporciona *Google* denominada *Cloud Natural Language* desde la propia web [77]. Los resultados se muestran en la Fig. 60.

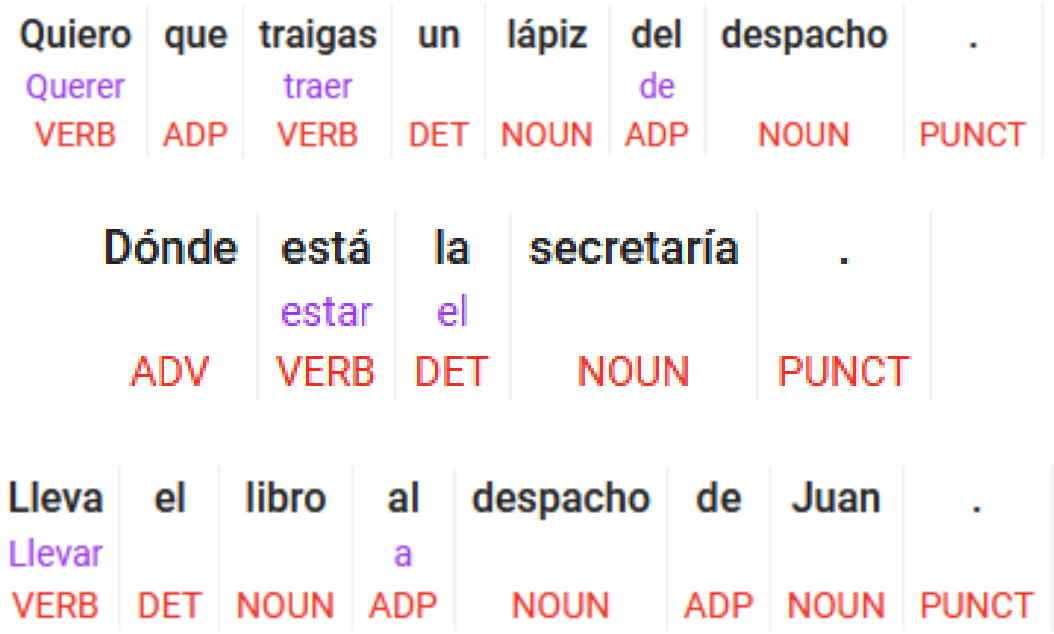

Fig. 60 Análisis gramatical API NLP de Google

En la segunda prueba se ha comprobado el funcionamiento del sistema experto. Está definido como un conjunto de reglas lógicas que extraen el verbo, el objeto y el lugar de las frases como se observa en la Fig. 61.

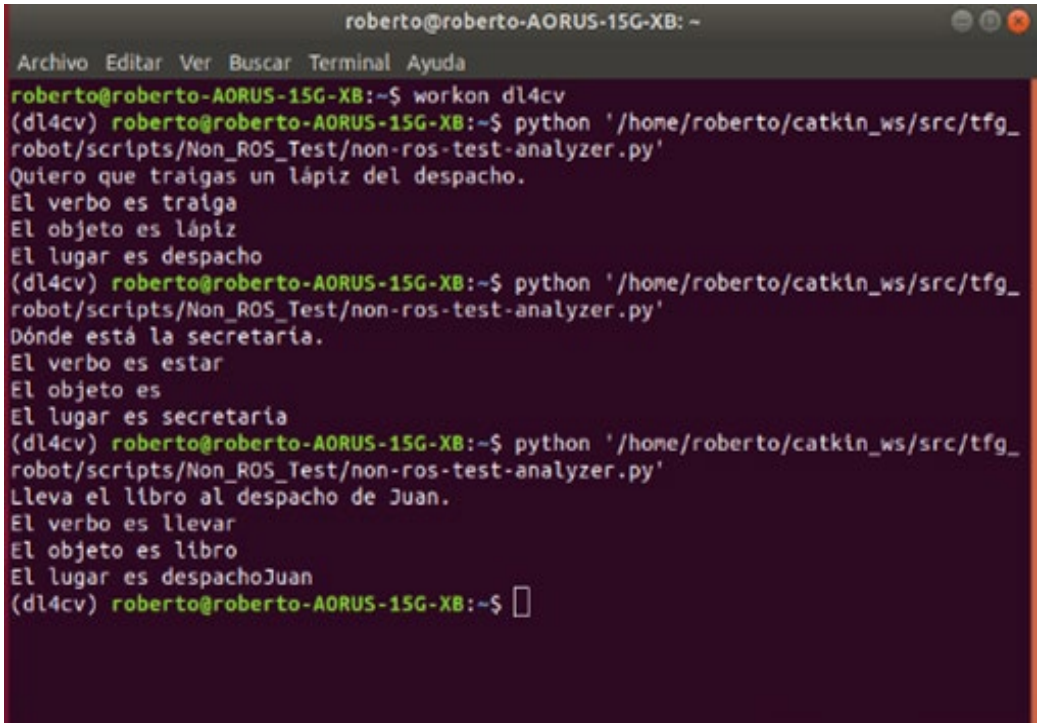

Fig. 61 Resultados del sistema experto NLP

4.1.6 Pruebas de navegación y resultados.

Las pruebas de navegación demuestran que el robot es capaz buscar la ruta para alcanzar un objetivo y puede esquivar obstáculos. Dichas experiencias se muestran en la Fig. 62, en la Fig. 63, en la Fig. 64 y en la Fig. 65.

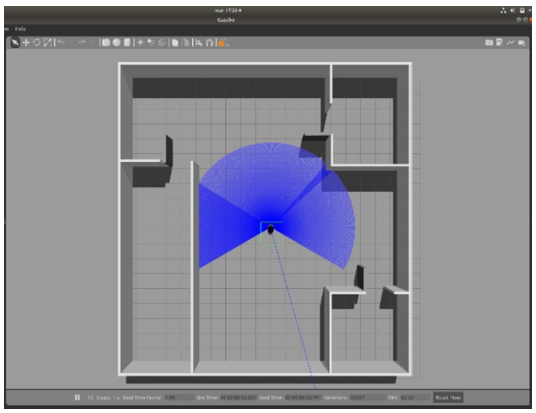

Fig. 62 Entorno 3D en Gazebo

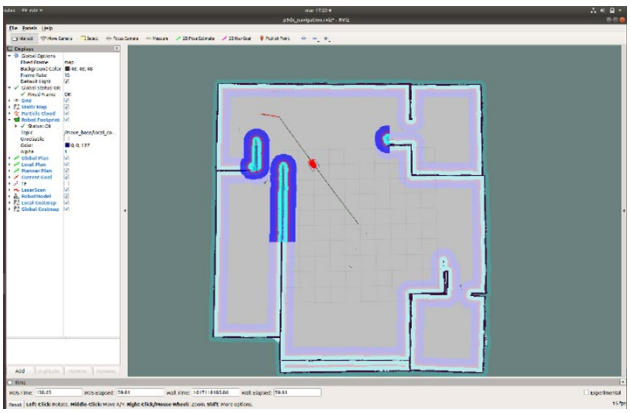

Fig. 63 Planificación de trayectoria

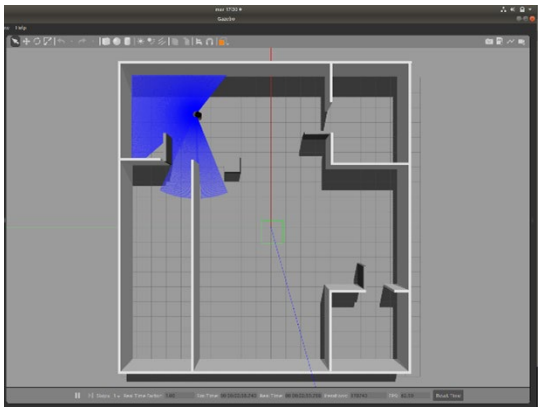

Fig. 64 Entorno 3D en Gazebo con obstáculo

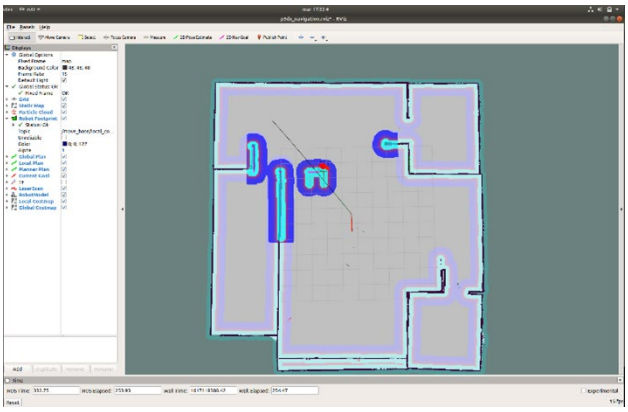

Fig. 65 Estrategia para esquivar obstáculos

4.1.7 Pruebas finales del proceso de integración y resultados.

Es una tarea compleja plasmar las pruebas realizadas para comprobar la integración de los resultados. En primer lugar se va a comparan, por un lado, el gráfico de nodos cuando se lanzan todos los paquetes, y el gráfico de nodos junto a los *topics* activos, como se muestra en la Fig. 66 y en la Fig. 67, respectivamente.

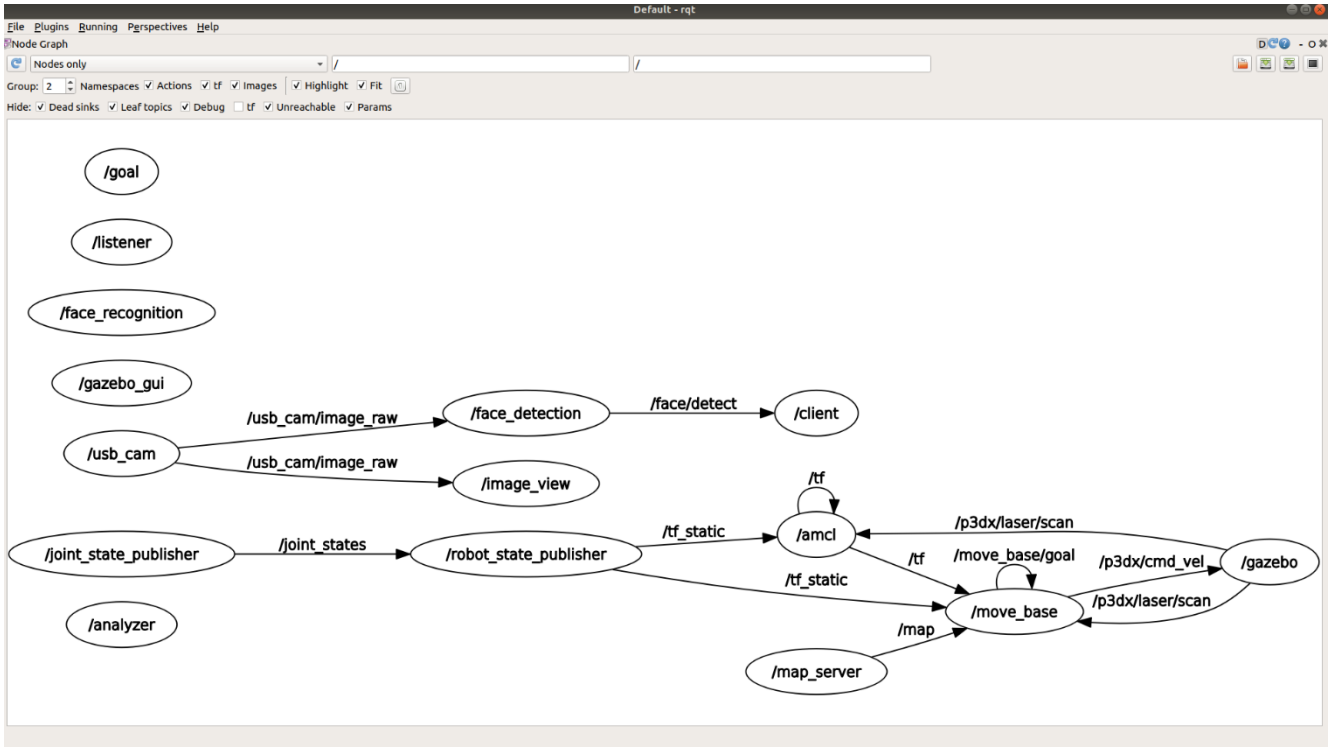

Fig. 66 Esquema gráfico de los nodos

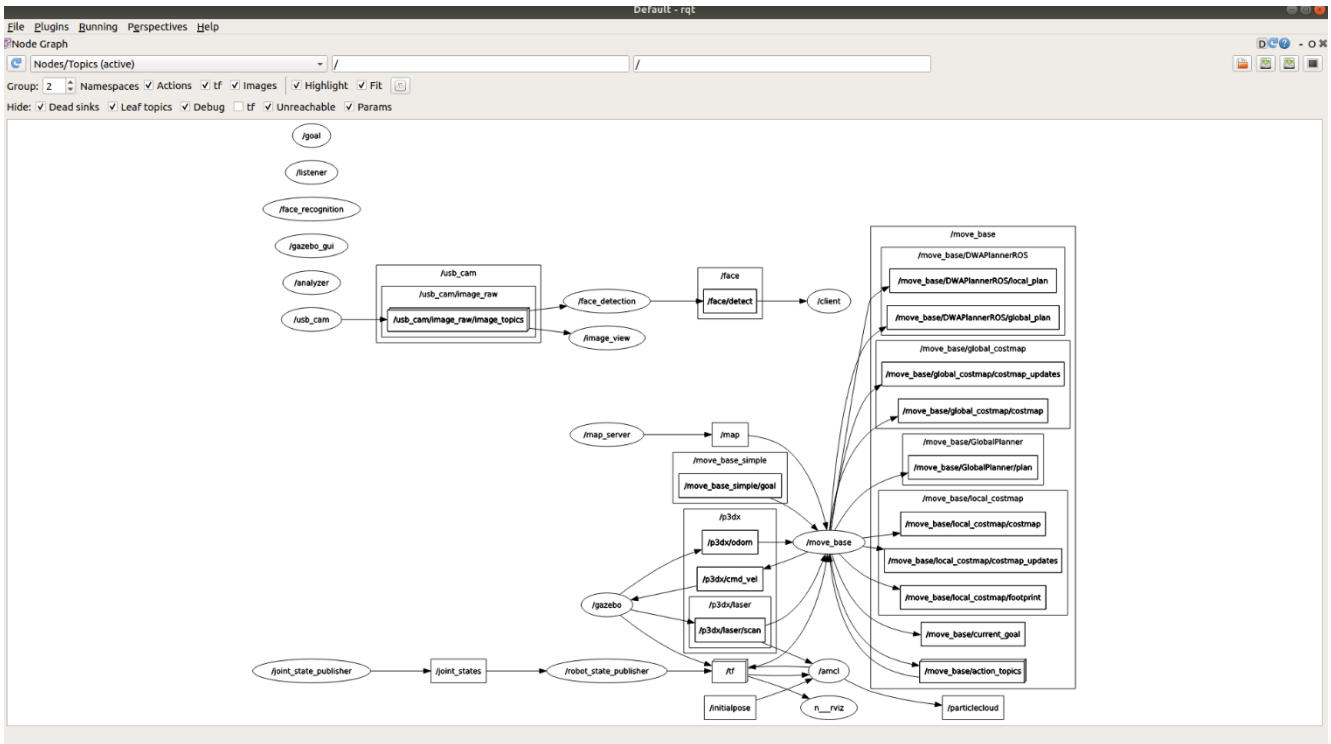

Fig. 67 Esquema gráfico de los nodos y *topics* activos

En segundo lugar se comprueba que todos los componentes del robot simulado estén relacionados. Esto quiere decir que los sistemas de referencia de los diferentes componentes estén ligados como se observa en la Fig. 68.

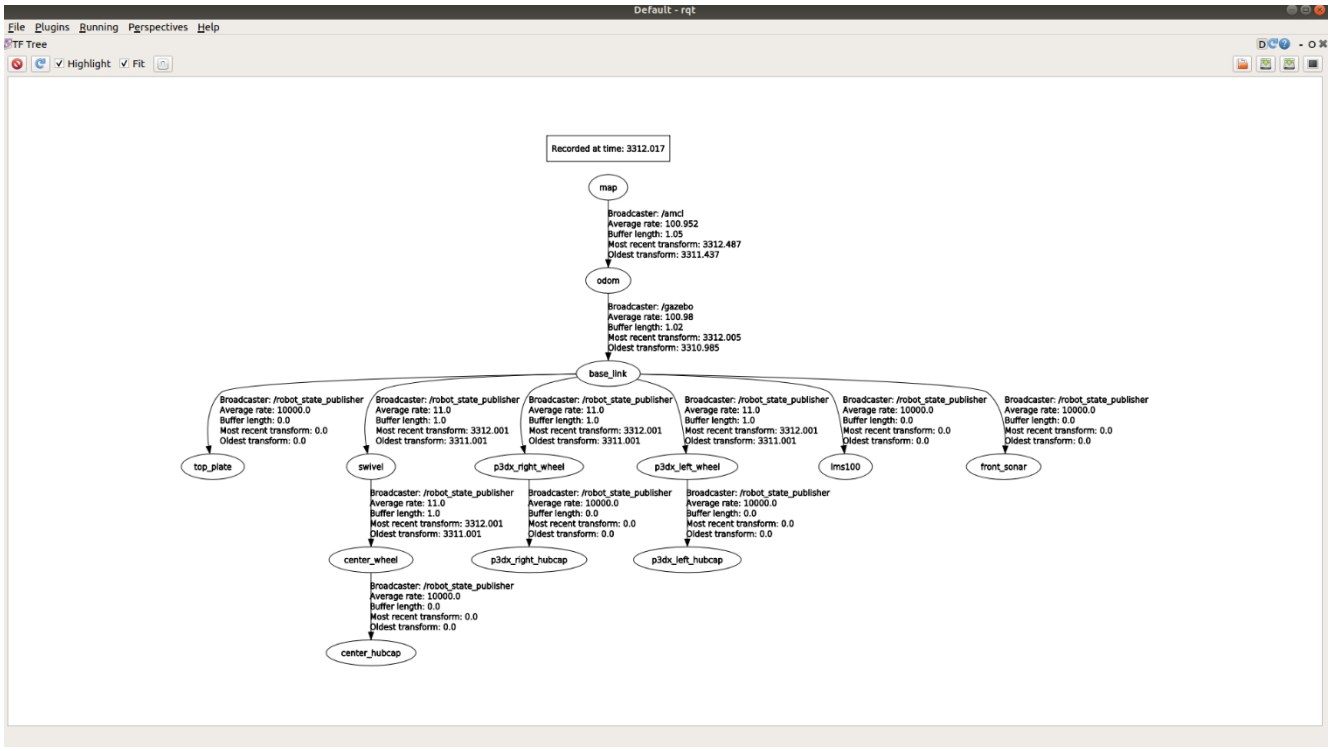

Fig. 68 Esquema gráfico de las transformaciones de los sistemas de referencia del robot

En tercer lugar se comprueba que la estructura de los servicios es tal y como se ha definido. La Fig. 69 muestra el diseño de los distintos servicios.

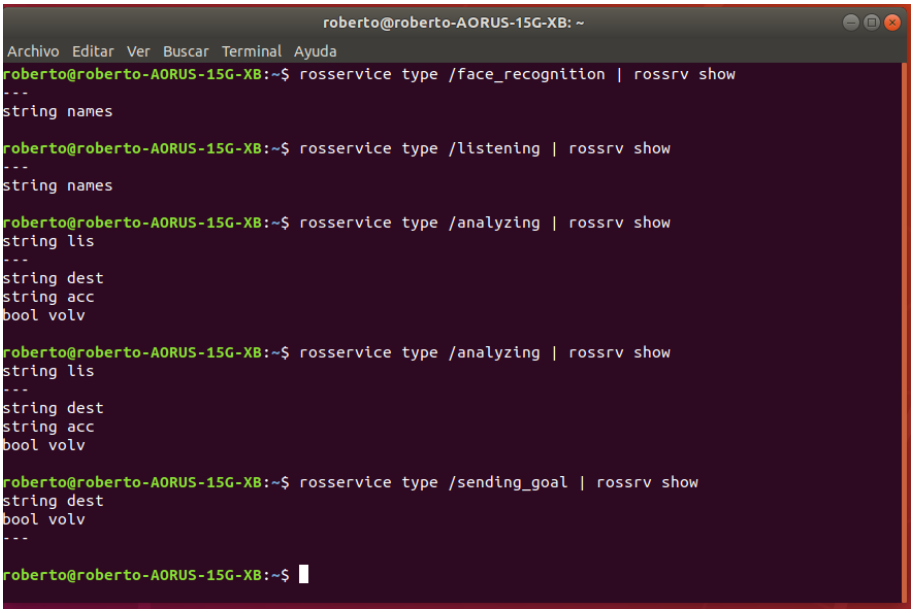

Fig. 69 Estructura de los servicios propuestos

Por último, se muestra una imagen del conjunto del sistema (ver Fig. 70).

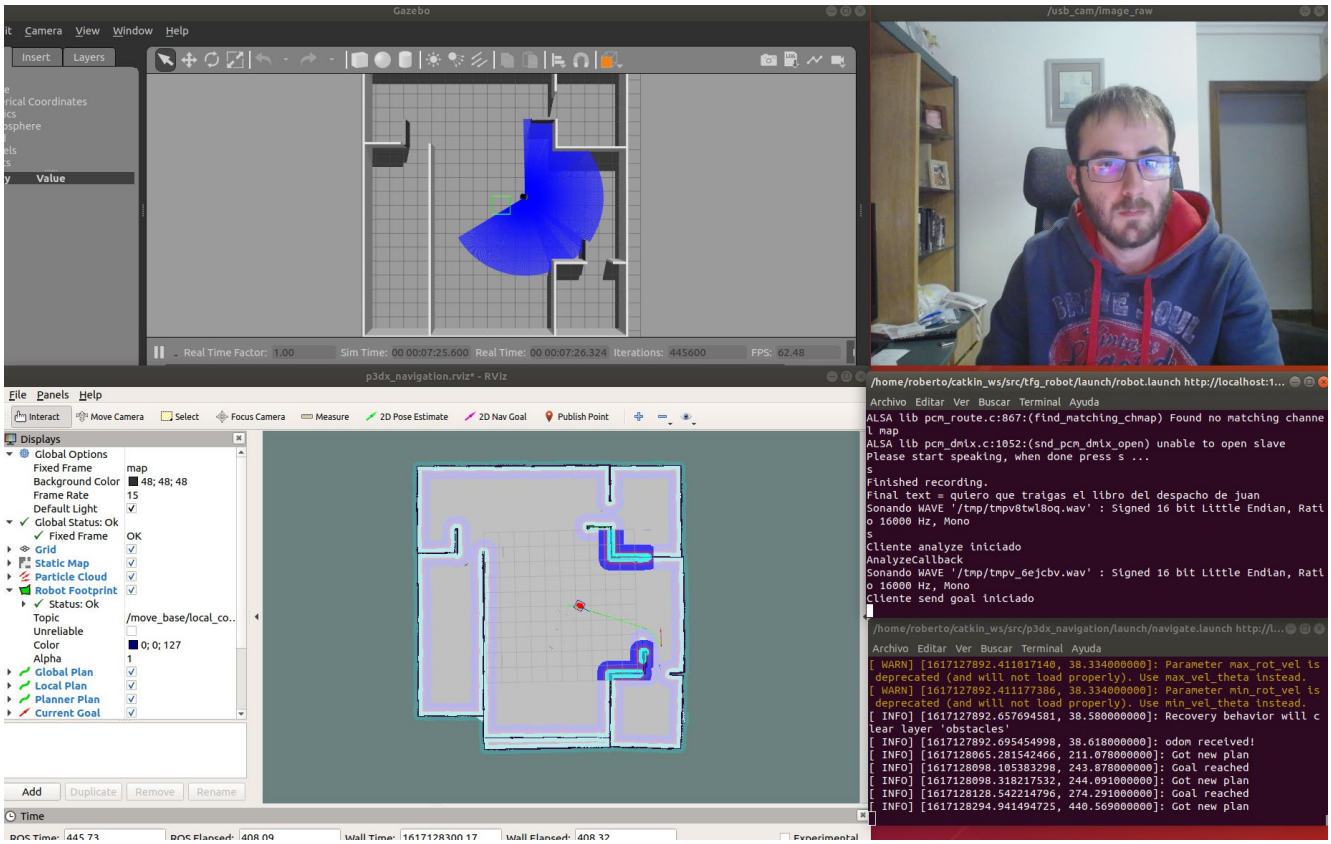

Fig. 70 Ventanas del conjunto del sistema

4.2 Discusión.

Una vez comprobados los resultados de la integración, se puede confirmar que se ha conseguido logar el objetivo de poder comunicarse con un robot de forma natural y que disponga de navegación autónoma.

La metodología de dividir el problema en pequeños módulos ha sido la adecuada porque ha permitido realizar las diferentes pruebas por separado y, por tanto, solucionar problemas de una forma sencilla. Además, la idea de crear librerías ha posibilitado usar las diferentes funciones en un entorno de trabajo virtual, para comprobar su funcionamiento y modificar ciertos aspectos antes de implantarlas en el sistema final.

A continuación se describen los principales beneficios y limitaciones del sistema:

Beneficios:

- Al usar modelos de Inteligencia Artificial (concretamente *Deep Learning*) que no requieren de conexión a internet se consigue una funcionalidad plena en modo fuera de línea o local. Además el uso de modelos ligeros, permite desplegar el sistemas en diferentes dispositivos.

- Los modelos de detección y reconocimiento facial han dado un buen resultado. Este punto es importante porque es la parte que más tiempo está activa y sería la que más gastos produciría en el caso de contratar un servicio en la Nube.
- La librería de síntesis de voz también ha funcionado según lo esperado, porque al usar el paquete de lenguaje del propio sistema operativo se evita que el tono de voz suene artificial.
- El paquete de navegación de ROS es robusto y ha permitido dotar al robot de un sistema de navegación autónoma completo incluyendo técnicas para describir una ruta que permite alcanzar una meta y evitar los obstáculos encontrados en la trayectoria.
- El uso de ROS ha permitido una correcta integración de todos los componentes de una forma sencilla mediante el modo de comunicación cliente-servidor que permite sincronizar los nodos.

Limitaciones:

- El modelo de reconocimiento de voz tiene problemas relacionados con el ruido y en ocasiones ciertas palabras no son reconocidas en el proceso de inferencia. Esto se debe, principalmente, a que se ha usado un modelo en español entrenado por la comunidad y no se ha lanzado de forma oficial. En este caso se podría realizar un entrenamiento exhaustivo del modelo o utilizar un servicio que proporcione la Nube.
- El modelo de Procesamiento de Lenguaje Natural también presenta limitaciones menores en la funcionalidad de obtener la raíz semántica de las palabras. En este aspecto se podrían usar sistemas más pesados como el de *Stanza* o los servicios que proporciona la Nube.
- Por último, es obvio que el sistema experto es cerrado. Debería tener una constante actualización para entender diferentes órdenes, pero se ha diseñado de manera fácilmente escalable.

5 Conclusiones y trabajo futuro.

Para concluir el trabajo, se realizará un pequeño resumen de los objetivos obtenidos tras la realización de éste, cómo se han obtenido y una breve explicación de por qué se han usado las metodologías descritas en apartados anteriores. Por último, se describirán posibles proyectos que se pueden realizar a partir de éste, como serían añadir nuevas funcionalidades o mejor ciertas limitaciones.

5.1 Conclusiones.

A lo largo del desarrollo de este trabajo, se han investigado diferentes técnicas disponibles que se podían aplicar para conseguir definir el sistema, de las cuales se han obtenido varias conclusiones.

En un primer momento se debatió si usar servicios que podían proporcionar las diferentes nubes de las empresas *Google, Microsoft, Amazon*… o trabajar con modelos de aprendizaje profundo guardados en el propio sistema. Se eligió la segunda opción por la versatilidad que proporciona, la independencia de conexión a Internet y el ahorro de coste que suponía.

Ahora bien, existe un gran número de modelos de Inteligencia Artificial. El desafío ha sido elegir aquellos que puedan funcionar en un hardware limitado como puede ser una *Rapberry Pi* cuyas características son similares al equipo donde se han realizado las pruebas. Por supuesto, una vez seleccionados, ha sido necesario estudiar su arquitectura para comprenderlo y aplicarlos.

Sin embargo, han surgido algunos problemas con algunos de los modelos de reconocimiento de voz y *NLP*, que todavía están en un proceso de evolución. Los problemas surgen principalmente con todo lo relacionado con la voz porque suelen estar diseñados para el idioma inglés.

Otro punto importante ha sido la división del trabajo en diferentes módulos porque ha permitido programar la funcionalidad de cada sistema independientemente. De esta forma se ha conseguido comprobar cada parte del sistema de forma particular antes de realizar la integración, con lo que se ha facilitado el trabajo.

Por último, para integrar los diferentes componentes se ha utilizado *ROS* que gracias a un forma de comunicación cliente-servidor ha permitido sincronizar el sistema de forma sencilla. Además, dada la situación de pandemia vivida en el último año se optó por trabajar con un simulador, en concreto *Gazebo*. En todo caso, la programación es compatible con un robot real.

En definitiva, se ha logrado el principal objetivo, diseñar un robot con navegación autónoma que permitiese comunicarse con las personas de forma natural.

5.1.1 Trabajo futuro.

En cuanto a trabajos futuros, como primera opción, se plantea la posibilidad de integrar el sistema en un robot real. Los diferentes módulos de programación son totalmente compatibles, luego habría que centrarse en cómo crear la comunicación de los diferentes componente del robot con *ROS*.

Por otro lado, otros aspectos que se pueden ampliar son la creación de diferentes funciones como por ejemplo: poder guardar una imagen de la cara de una persona directamente interactuando con el robot, para que pueda ser reconocida a posteriori; la creación de una función para que guarde posibles lugares de interés e incluso un módulo que permitiese realizar un mantenimiento predictivo de los elementos del robot.

Por último, como se ha mencionado en el sección 4.2, existen ciertas debilidades con algunos modelos de Aprendizaje Automático. Se podrían dar dos posibles soluciones. La primera consistiría en realizar este mismo trabajo usando los servicios que proporcionan las Nubes a través de proveedores como *Google, Amazon o Microsoft*. La segunda sería crear un servidor propio donde alojar los modelos de aprendizaje profundo más pesados, entonces desde el propio sistema del robot realizar una petición y recibir como respuesta la inferencia. Dado que estas líneas de trabajo implicarían requerir una conexión a Internet, se podría considerar desarrollar un sistema que utilizase los modelos internos en el caso de que no se dispusiese de esta conexión.

6 Bibliografía.

[1] «Professional Service Robots Resources & Education | RIA», *Robotics Online*. https://www.robotics.org/service-robots (accedido ene. 07, 2021).

[2] K. Li, J. Wu, X. Zhao, y M. Tan, «Real-Time Human-Robot Interaction for a Service Robot Based on 3D Human Activity Recognition and Human-Mimicking Decision Mechanism», en *2018 IEEE 8th Annual International Conference on CYBER Technology in Automation, Control, and Intelligent Systems (CYBER)*, Tianjin, China, jul. 2018, pp. 498-503, doi: 10.1109/CYBER.2018.8688272.

[3] IFR, «International Federation of Robotics», *IFR International Federation of Robotics*. https://ifr.org/service-robots/: (accedido ene. 14, 2021).

[4] M. E. Pollack *et al.*, «Pearl: A Mobile Robotic Assistant for the Elderly», p. 7.

[5] «Robot Pepper», *Aliverobots*. https://aliverobots.com/robot-pepper/ (accedido ene. 14, 2021).

[6] K. Shiarlis *et al.*, «TERESA: A Socially Intelligent Semi-autonomous Telepresence System», p. 7.

[7] S. Quinlan y O. Khatib, «Elastic bands: connecting path planning and control», en *[1993] Proceedings IEEE International Conference on Robotics and Automation*, Atlanta, GA, USA, 1993, pp. 802-807, doi: 10.1109/ROBOT.1993.291936.

[8] S. T. Pfister, «Algorithms for Mobile Robot Localization and Mapping, Incorporating Detailed Noise Modeling and Multi-Scale Feature Extraction», p. 200.

[9] L. Jetto, S. Longhi, y G. Venturini, «Development and experimental validation of an adaptive extended Kalman filter for the localization of mobile robots», *IEEE Trans. Robot. Autom.*, vol. 15, n.º 2, pp. 219-229, abr. 1999, doi: 10.1109/70.760343.

[10] D. Fox, W. Burgard, y S. Thrun, «Markov Localization for Reliable Robot Navigation and People Detection», en *Sensor Based Intelligent Robots*, vol. 1724, H. I. Christensen, H. Bunke, y H. Noltemeier, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 1-20.

[11] A. Musa, «Advanced Techniques for Mobile Localization and Tracking», p. 177.

[12] F. Dellaert, D. Fox, W. Burgard, y S. Thrun, «Monte Carlo localization for mobile robots», en *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, Detroit, MI, USA, 1999, vol. 2, pp. 1322-1328, doi: 10.1109/ROBOT.1999.772544.

[13] R. Siegwart y I. R. Nourbakhsh, *Introduction to autonomous mobile robots*. Cambridge, Mass: MIT Press, 2004.

[14] G. Dudek y M. Jenkin, *Computational principles of mobile robotics*, 2nd ed. New York: Cambridge University Press, 2010.

[15] M. A. Javaid, «Understanding Dijkstra Algorithm», *SSRN Electron. J.*, 2013, doi: 10.2139/ssrn.2340905.

[16] B. M. ElHalawany, H. M. Abdel-Kader, A. E. Elsayed, y Z. B. Nossair, «Modified A* Algorithm for Safer Mobile Robot Navigation», p. 6, 2013.

[17] J. C. Montesdeoca, J. M. Toibero, y R. Carelli, «Human-robot interaction, evolution, advances and new challenges», en *2017 XVII Workshop on Information Processing and Control (RPIC)*, Mar del Plata, sep. 2017, pp. 1-4, doi: 10.23919/RPIC.2017.8214308.

[18] R. B. Gonzalo, «Usability in biometric recognition systems», p. 201.

[19] J. N. Pato y L. I. Millett, «Whither Biometrics Committee», *Biom. Recognit.*, p. 183.

[20] A. K. Sharma, A. Raghuwanshi, y V. K. Sharma, «Biometric System- A Review», vol. 6, p. 5, 2015.

[21] «¿Qué es la huella dactilar y para qué sirve?», *Cucorent*, abr. 11, 2019. https://www.cucorent.com/blog/que-es-la-huella-dactilar-sirve/ (accedido ene. 10, 2021).

[22] Y.-Q. Wang, «An Analysis of the Viola-Jones Face Detection Algorithm», *Image Process. Line*, vol. 4, pp. 128-148, jun. 2014, doi: 10.5201/ipol.2014.104.

[23] «Face Recognition Using HOG Feature Extraction and SVM Classifier», *Int. J. Emerg. Trends Eng. Res.*, vol. 8, n.º 9, pp. 6437-6440, sep. 2020, doi: 10.30534/ijeter/2020/244892020.

[24] A. V. Ponkia y J. Chaudhari, «Face Recognition Using PCA Algorithm», vol. 2012, n.º 1, p. 5, 2012.

[25] A. Imran, M. Miah, H. Rahman, A. Bhowmik, y D. Karmaker, «Face Recognition using Eigenfaces», *Int. J. Comput. Appl.*, vol. 118, pp. 12-16, may 2015, doi: 10.5120/20740-3119.

[26] J. Li, Y. Wang, T. Tan, y A. K. Jain, «Live face detection based on the analysis of Fourier spectra», Orlando, FL, ago. 2004, pp. 296-303, doi: 10.1117/12.541955.

[27] T. Ahonen, A. Hadid, y M. Pietikäinen, «Face Recognition with Local Binary Patterns», may 2004, vol. 3021, pp. 469-481, doi: 10.1007/978-3-540-24670-1_36.

[28] E. Sadeghipour y N. Sahragard, «Face Recognition Based on Improved SIFT Algorithm», *Int. J. Adv. Comput. Sci. Appl.*, vol. 7, ene. 2016, doi: 10.14569/IJACSA.2016.070175.

[29] X. Zhu, S. Liao, Z. Lei, R. Liu, y S. Li, «Feature Correlation Filter for Face Recognition», ago. 2007, vol. 4642, pp. 77-86, doi: 10.1007/978-3-540-74549-5_9.

[30] R. K, B. Thilaka, y R. N, «An adaptive K-Means Clustering Algorithm and its Application to Face Recognition», *J. Appl. Comput. Sci. Math.*, vol. 4, ene. 2010.

[31] M. Gibran, E. Nababan, y P. Sihombing, «Analysis of Face Recognition with Fuzzy C-Means Clustering Image Segmentation and Learning Vector Quantization», jun. 2020, pp. 188-193, doi: 10.1109/MECnIT48290.2020.9166649.

[32] A. Wirdiani, P. Hridayami, A. Widiari, K. Rismawan, P. Candradinata, y I. Jayantha, «Face Identification Based on K-Nearest Neighbor», *Sci. J. Inform.*, vol. 6, pp. 150-159, nov. 2019, doi: 10.15294/sji.v6i2.19503.

[33] S. Lawrence, C. Giles, A. Tsoi, y A. Back, «Face Recognition: A Convolutional Neural Network Approach», *Neural Netw. IEEE Trans. On*, vol. 8, pp. 98-113, feb. 1997, doi: 10.1109/72.554195.

[34] «Reconocimiento de voz», *HiSoUR Arte Cultura Historia*, nov. 05, 2018. https://www.hisour.com/es/speech-recognition-42804/ (accedido ene. 12, 2021).

[35] «Cognitive Services: API para desarrolladores de inteligencia artificial | Microsoft Azure». https://azure.microsoft.com/es-es/services/cognitive-services/ (accedido ene. 13, 2021).

[36] «Text-to-Speech: síntesis de voz natural | Cloud Text-to-Speech», *Google Cloud*. https://cloud.google.com/text-to-speech?hl=es (accedido ene. 13, 2021).

[37] «Amazon Polly», *Amazon Web Services, Inc.* https://aws.amazon.com/es/polly/ (accedido ene. 13, 2021).

[38] «eSpeak: Speech Synthesizer». http://espeak.sourceforge.net/ (accedido ene. 13, 2021).

[39] «Festvox: Festival». http://festvox.org/festival/ (accedido ene. 13, 2021).

[40] «Difono», *Wikipedia, la enciclopedia libre*. jul. 11, 2019, Accedido: abr. 05, 2021. [En línea]. Disponible en: https://es.wikipedia.org/w/index.php?title=Difono&oldid=117350973.

[41] «pyttsx3 - Text-to-speech x-platform — pyttsx3 2.6 documentation». https://pyttsx3.readthedocs.io/en/latest/ (accedido abr. 05, 2021).

[42] «Modulo Pyttsx: Reproduciendo texto desde Python», may 14, 2018. https://carlosjuliopardoblog.wordpress.com/2018/05/14/motores-de-voz/ (accedido abr. 05, 2021).

[43] J. I. B. Fernández, «Elaboración de un modelo NER para su uso en aplicaciones NLP», p. 197.

[44] «Natural Language Toolkit — NLTK 3.5 documentation». https://www.nltk.org/ (accedido ene. 14, 2021).

[45] «spaCy · Industrial-strength Natural Language Processing in Python». https://spacy.io/ (accedido ene. 14, 2021).

[46] «UDPipe Natural Language Processing - Text Annotation». https://cran.r-project.org/web/packages/udpipe/vignettes/udpipe-annotation.html (accedido ene. 14, 2021).

[47] «Overview», *Stanza*. https://stanfordnlp.github.io/stanza/ (accedido ene. 14, 2021).

[48] «Azure Kinect DK: desarrollo de modelos de IA | Microsoft Azure». https://azure.microsoft.com/es-es/services/kinect-dk/ (accedido mar. 26, 2021).

[49] «Gazebo». http://gazebosim.org/ (accedido abr. 05, 2021).

[50] «ROS/Introduction - ROS Wiki». Accedido: mar. 25, 2021. [En línea]. Disponible en: http://wiki.ros.org/ROS/Introduction.

[51] C. C. Cuevas Castañeda, «Ros-gazebo. una valiosa Herramienta de Vanguardia para el Desarrollo de la Robótica», *Publicaciones E Investig.*, vol. 10, p. 145, mar. 2016, doi: 10.22490/25394088.1593.

[52] A. Fernández, «Qué es la Inteligencia Artificial • Definición, ejemplos y casos de uso.», *AuraPortal*, jun. 19, 2019. https://www.auraquantic.com/es/que-es-la-inteligencia-artificial/ (accedido mar. 26, 2021).

[53] «Expert Systems in Artificial Intelligence - Javatpoint», *www.javatpoint.com*. https://www.javatpoint.com/expert-systems-in-artificial-intelligence (accedido mar. 25, 2021).

[54] «MorphoDiTa User's Manual | ÚFAL». https://ufal.mff.cuni.cz/morphodita/users-manual (accedido mar. 26, 2021).

[55] D. «johanka» Spoustová, J. Hajič, J. Raab, y M. Spousta, «Semi-supervised training for the averaged perceptron POS tagger», en *Proceedings of the 12th Conference of the European Chapter of the Association for Computational Linguistics on - EACL '09*, Athens, Greece, 2009, pp. 763-771, doi: 10.3115/1609067.1609152.

[56] FrancescaLazzeri, «Aprendizaje profundo frente a aprendizaje automático - Azure Machine Learning». https://docs.microsoft.com/es-es/azure/machine-learning/concept-deep-learning-vs-machine-learning (accedido mar. 25, 2021).

[57] L. M. N. Vivero, «SEGMENTACIÓN DE IMÁGENES DE RESONANCIA MAGNÉTICA CEREBRAL MEDIANTE REDES NEURONALES ARTIFICIALES CONVOLUCIONALES», p. 46.

[58] «Convolutional Neural Networks: La Teoría explicada en Español | Aprende Machine Learning». https://www.aprendemachinelearning.com/como-funcionan-las-convolutional-neural-networks-vision-por-ordenador/ (accedido mar. 26, 2021).

[59] J.-S. Lim, M. Astrid, H.-J. Yoon, y S.-I. Lee, «Small Object Detection using Context and Attention», *ArXiv191206319 Cs*, dic. 2019, Accedido: mar. 27, 2021. [En línea]. Disponible en: http://arxiv.org/abs/1912.06319.

[60] F. Schroff, D. Kalenichenko, y J. Philbin, «FaceNet: A Unified Embedding for Face Recognition and Clustering», *2015 IEEE Conf. Comput. Vis. Pattern Recognit. CVPR*, pp. 815-823, jun. 2015, doi: 10.1109/CVPR.2015.7298682.

[61] D. Calvo, «Red Neuronal Recurrente - RNN», *Diego Calvo*, dic. 09, 2018. https://www.diegocalvo.es/red-neuronal-recurrente/ (accedido mar. 26, 2021).

[62] Darkcrizt, «DeepSpeech: el motor de reconocimiento de voz de Mozilla», *Desde Linux*, dic. 11, 2019. https://blog.desdelinux.net/deepspeech-el-motor-de-reconocimiento-de-voz-de-mozilla/ (accedido mar. 26, 2021).

[63] «TensorFlow». https://www.tensorflow.org/?hl=es-419 (accedido mar. 26, 2021).

[64] «Todo lo que necesitas saber sobre TensorFlow, la plataforma para Inteligencia Artificial de Google – Puentes Digitales». https://puentesdigitales.com/2018/02/14/todo-lo-que-necesitas-saber-sobre-tensorflow-la-plataforma-para-inteligencia-artificial-de-google/ (accedido mar. 26, 2021).

[65] «Cuantización posterior al entrenamiento | TensorFlow Lite». https://www.tensorflow.org/lite/performance/post_training_quantization?hl=es (accedido mar. 26, 2021).

[66] «OpenCV», *OpenCV*. https://opencv.org/ (accedido mar. 27, 2021).

[67] «cv_bridge - ROS Wiki». http://wiki.ros.org/cv_bridge (accedido mar. 27, 2021).

[68] «Caffe | Deep Learning Framework». https://caffe.berkeleyvision.org/ (accedido mar. 27, 2021).

[69] «opencv/opencv», *GitHub*. https://github.com/opencv/opencv (accedido mar. 27, 2021).

[70] F. Schroff, D. Kalenichenko, y J. Philbin, «FaceNet: A Unified Embedding for Face Recognition and Clustering», *2015 IEEE Conf. Comput. Vis. Pattern Recognit. CVPR*, pp. 815-823, jun. 2015, doi: 10.1109/CVPR.2015.7298682.

[71] «Jaco-Assistant / DeepSpeech-Polyglot», *GitLab*. https://gitlab.com/Jaco-Assistant/deepspeech-polyglot (accedido mar. 27, 2021).

[72] «navigation - ROS Wiki». http://wiki.ros.org/navigation (accedido mar. 28, 2021).

[73] «navigation/Tutorials/RobotSetup - ROS Wiki». http://wiki.ros.org/navigation/Tutorials/RobotSetup (accedido mar. 28, 2021).

[74] «Google Colaboratory». https://colab.research.google.com/notebooks/welcome.ipynb?hl=es (accedido mar. 30, 2021).

[75] M. Roman, «INTRODUCCIÓN A GOOGLE COLAB PARA DATA SCIENCE», *Datahack*, jun. 17, 2019. https://www.datahack.es/blog/big-data/google-colab-para-data-science/ (accedido mar. 30, 2021).

[76] «Speech-to-Text: reconocimiento de voz automático», *Google Cloud*. https://cloud.google.com/speech-to-text?hl=es (accedido mar. 30, 2021).

[77] «Cloud Natural Language | Cloud Natural Language | Google Cloud». https://cloud.google.com/natural-language?hl=es (accedido mar. 30, 2021).

[78] «melodic/Installation/Ubuntu - ROS Wiki». http://wiki.ros.org/melodic/Installation/Ubuntu (accedido mar. 31, 2021).

[79] «Ubuntu 18.04: Install TensorFlow and Keras for Deep Learning», *PyImageSearch*, ene. 30, 2019. https://www.pyimagesearch.com/2019/01/30/ubuntu-18-04-install-tensorflow-and-keras-for-deep-learning/ (accedido mar. 31, 2021).

Anexo I Instalación de ROS y configuración.

A continuación se va a realizar una guía de cómo se instala *ROS* en un equipo. En este caso se disponía de Ubuntu ya instalado, en caso contrario se debería realizar una partición en el disco duro o utilizar una máquina virtual para poder instalarlo. Como se dispone de la versión 18.04 esta guía en concreto servirá para instalar la versión *ROS Melodic Morenia* [78].

En primer lugar hay que configurar el dispositivo para que pueda recibir los paquetes de software procedentes de *packages.ros.org*. Desde una terminal hay que escribir lo siguiente:

```
sudo sh -c 'echo "deb http://packages.ros.org/ros/ubuntu $(lsb_release -
sc) main" > /etc/apt/sources.list.d/ros-latest.list'
```

Seguidamente, hay que establecer una clave, para evitar se descargue un software incorrecto:

```
sudo apt-key adv --keyserver 'hkp://keyserver.ubuntu.com:80' --recv-key
C1CF6E31E6BADE8868B172B4F42ED6FBAB17C654
```

Ahora hay que asegurarse de que la lista de paquetes *Debian* esté actualizada:

```
sudo apt update
```

Hay muchas herramientas y bibliotecas de ROS, pero se recomienda instalar la versión completa que incluye *ROS, rqt , rviz* , bibliotecas genéricas de robots, simuladores 2D / 3D y percepción 2D / 3D. También se podrían instalar los paquetes individuales . El siguiente comando instalaría la versión completa:

```
sudo apt install ros-melodic-desktop-full
```

A continuación, se deben establecer las variables de entorno para que no haga falta hacerlo cada vez que se abra una nueva terminal:

```
echo "source /opt/ros/melodic/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Ya se ha instalado lo necesario para ejecutar los paquetes principales de ROS. Para crear y administrar espacios de trabajos propios en ROS, existen varias herramientas y requisitos que se distribuyen por separado.

Por ejemplo*, rosinstall* es una herramienta de línea de comandos que permite descargar fácilmente árboles de origen para paquetes ROS con un solo comando.

Para instalar esta herramienta y otras dependencias para construir paquetes ROS se debe ejecutar lo siguiente:

```
sudo apt install python-rosdep python-rosinstall python-rosinstall-
generator python-wstool build-essential
```

Por último, antes de poder utilizar muchas herramientas de ROS, se debe iniciar *rosdep* y actualizarlo. Permite instalar fácilmente las dependencias del sistema para la fuente que se desea compilar y es necesario para ejecutar algunos componentes centrales de ROS. Se instala con el siguiente comando y se debe iniciar y actualizar:

```
sudo apt install python-rosdep
sudo rosdep init
rosdep update
```

Si todo se ha instalado correctamente, con el siguiente comando se iniciaría ROS (con ctrl+c se para):

```
roscore
```

Una vez comprobado el correcto funcionamiento se pueden instalar paquetes de ROS, por ejemplo el paquete de navegación, el paquete de *gmapping* y el de tele operación.

```
sudo apt-get install ros-melodic-navigation
sudo apt-get install ros-melodic-gmapping
sudo apt-get install ros-melodic-teleop-twist-keyboard
```

Por otro lado, es recomendable crear un espacio de trabajo. Para ello en una terminal se ejecuta la siguiente instrucción:

```
mkdir -p catkin_ws/src
```

Luego, hay que dirigirse al directorio que se ha creado para e iniciar del espacio de trabajo:

```
cd ~/catkin_ws/src
catkin_init_workspace
```

Una vez iniciado es necesario crear los paquetes dentro del espacio de trabajo, mediante la siguiente instrucción aunque aún no haya paquetes. La primera vez se crearán dos nuevas carpetas denominadas *build y devel*:

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

A continuación, se deben establecer las variables del espacio de trabajo para que no haga falta hacerlo cada vez que se abra una nueva terminal:

```
echo "source ~/catkin_ws/devel/setup.bash" >> ~/.bashrc
source ~/.bashrc
```

Anexo II Instalación de TensorFlow y creación de un entorno virtual de Python.

Esta guía de instalación se ha realizado para la versión de Ubuntu 18.04. Para comprobar primero el funcionamiento se instalará primero en un entorno virtual y una vez realizadas las pruebas en el propio sistema [79].

En primer lugar hay que comprobar que la lista de paquetes *Debian* esté actualizada ejecutando los siguientes comandos en una terminal:

```
sudo apt-get update
sudo apt-get upgrade
```

A continuación, se deben instalar ciertas herramientas de desarrollo, librerías para de I/O de imágenes y video, librerías de optimización y otros paquetes:

```
sudo apt-get install build-essential cmake unzip pkg-config
sudo apt-get install libxmu-dev libxi-dev libglu1-mesa libglu1-mesa-dev
sudo apt-get install libjpeg-dev libpng-dev libtiff-dev
sudo apt-get install libavcodec-dev libavformat-dev libswscale-dev
libv4l-dev
sudo apt-get install libxvidcore-dev libx264-dev
sudo apt-get install libgtk-3-dev
sudo apt-get install libopenblas-dev libatlas-base-dev liblapack-dev
gfortran
sudo apt-get install libhdf5-serial-dev
sudo apt-get install python3-dev python3-tk python-imaging-tk
```

En el caso de que se quiera utilizar la *GPU* con *TensroFlow*, habría que seguir los pasos mostrados en [79]. En este caso se va a usar la *CPU*, luego el siguiente paso sería crear el entorno virtual. Para ello primero se debe instalar *pip* que es una herramienta para administrar los paquetes de *Python*:

```
wget https://bootstrap.pypa.io/get-pip.py
sudo python3 get-pip.py
```

Seguidamente se instalarán dos herramientas que permite crear un entorno virtual:

```
sudo pip install virtualenv virtualenvwrapper
sudo rm -rf ~/get-pip.py ~/.cache/pip
```

A continuación, se deben estableces las variables de entorno para conseguir que esas dos herramientas trabajen de forma conjunta. Para ello hay que añadir abrir el fichero *~/.bashrc* y añadir al final las siguientes líneas:

```
# virtualenv and virtualenvwrapper
export WORKON_HOME=$HOME/.virtualenvs
export VIRTUALENVWRAPPER_PYTHON=/usr/bin/python3
source /usr/local/bin/virtualenvwrapper.sh
```

Ahora ya se puede crear el entorno virtual, ejecutando en el terminal lo siguiente:

```
mkvirtualenv dl4cv -p python3
```

Para activar el entorno se usa el siguiente comando:

```
workon dl4cv
```

Una vez activado ya se pueden instalar librería, por ejemplo:

```
pip install numpy
pip install opencv-contrib-python
pip install scipy matplotlib pillow
```

Posteriormente se instala *TernsorFlow y Keras*:

```
pip install tensorflow →(CPU)
pip install tensorflow-gpu==1.12.0 →(GPU)
pip install keras
```

Para comprobar el funcionamiento dentro del entorno se puede ejecutar lo siguiente, no debería dar ningún error:

```
python
import tensorflow
import keras
```

Por último para salir de la consola de Python y del entorno virtual, se debe ejecutar:

```
exit()
deactivate
```

Una vez que se hayan realizado todas las pruebas y se sepa qué librerías se van a utilizar en el trabajo, se deben instalar en el sistema tal y como se ha visto pero sin activar el entorno virtual.

Anexo III Modelización del entorno 3D en Gazebo y generación del mapa.

En este anexo se va a realizar un pequeño tutorial, en primer lugar de como se puede diseñar un entorno 3D con Gazebo. En segundo lugar, la forma de crear un mapa con el paquete de *ROS gmapping* que sirve tanto para simulaciones como para entornos reales. Los mapas generados con este paquete se pueden utilizar en el paquete de navegación.

Creación del entorno 3D en Gazebo.

En primer lugar se debe iniciar *Gazebo* mediante el siguiente comando en una terminal:

```
roslaunch p3dx_gazebo gazebo.launch
```

Para crear el diseño hay que ir *a edit, building editor*. Desde esa ventana se pueden añadir paredes con diferentes texturas, ventanas, etc. Una vez creado, se guardará dándole *click a exit, save and exit* y seleccionando los siguientes directorios:

```
/home/roberto/catkin_ws/src/ua_ros_p3dx/p3dx_gazebo/worlds/
/home/roberto/.gazebo
```

En la versión 18.04 de Ubuntu no se dispone de la carpeta *model*, luego se debe crear usan el terminal con los comandos mostrados a continuación:

```
cd .gazebo
ls -la
mkdir models
```

A continuación se debe copiar el diseño que se ha creado en su interior. El siguiente paso es asociar las carpetas de Gazebo, para ello hay que irse a la siguiente dirección:

```
/usr/share/gazebo-9
```

Y desde esa directorio, abrir una terminal para escribir lo siguiente:

```
sudo cp -r . /home/roberto/catkin_ws/src/ua_ros_p3dx/p3dx_gazebo/
```

Por último hay que ir a:

```
/catkin_ws/src/ua_ros_p3dx/p3dx_gazebo
```

Y modificar el archivo p3dx.world con lo siguiente:

```
<include>
  <uri>model://tfg</uri>
</include>
```

Siendo tfg el nombre del diseño que se ha creado. Ahora ya se puede cerrar Gazebo desde su terminal mediante ctrl+c e iniciarla de nuevo para comprobar que realmente se ha cargado todo de forma correcta.

Creación del mapa.

En primer lugar hay que ejecutar un nodo que permite realizar la transformación entre el sistema de referencia del láser y de la base del robot:

```
rosrun tf_transformacion tf_transformacion_node
```

Para asegurarse que está funciona se puede ejecutar en otra terminal lo siguiente:

```
rostopic echo tf
```

Una vez comprobado el funcionamiento ya se puede parar ese proceso y ejecutar el nodo de mapeo denominado *gmapping*. Hay que pasarle como parámetro el *topic* donde está publicando el láser:

```
rosrun gmapping slam_gmapping scan:=/p3dx/laser/scan
```

Se puede visualizar la creación del mapa con r*viz*, tan solo se debe ejecutar en otra terminal:

```
rviz
```

En la ventana que se abre desde la opción *add* hay que añadir *map* y seleccionar el *topic /map*. De este modo ya se debería ver par porción de mapa que ha generado el láser en la simulación.

A continuación, hay que mover al robot para que vaya creando el mapa. Una forma sencilla es mediante tele operación. Para ello se puede ejecutar el siguiente nodo pasándole como parámetro el *topic* donde se publica la velocidad del robot:

```
rosrun teleop_twist_keyboard teleop_twist_keyboard.py cmd_vel:=/p3dx/cmd_vel
```

En dicha terminal saldrá una ventana explicando como cambiar la velocidad del robot. Ahora tan solo hay que recorrer todo el mapa. Es recomendable realizar varias pasadas para conseguir que el mapa se actualice y se obtenga un mejor resultado.

Una vez generado el mapa hay que guardarlo, para ello se debe de lanzar una terminal desde la carpeta donde se desee guardar y escribir lo siguiente:

```
rosrun map_server map_saver -f 'file'
```

De esta manera ya se ha conseguido generar un mapa que se utilizará para ejecutar la navegación autónoma.