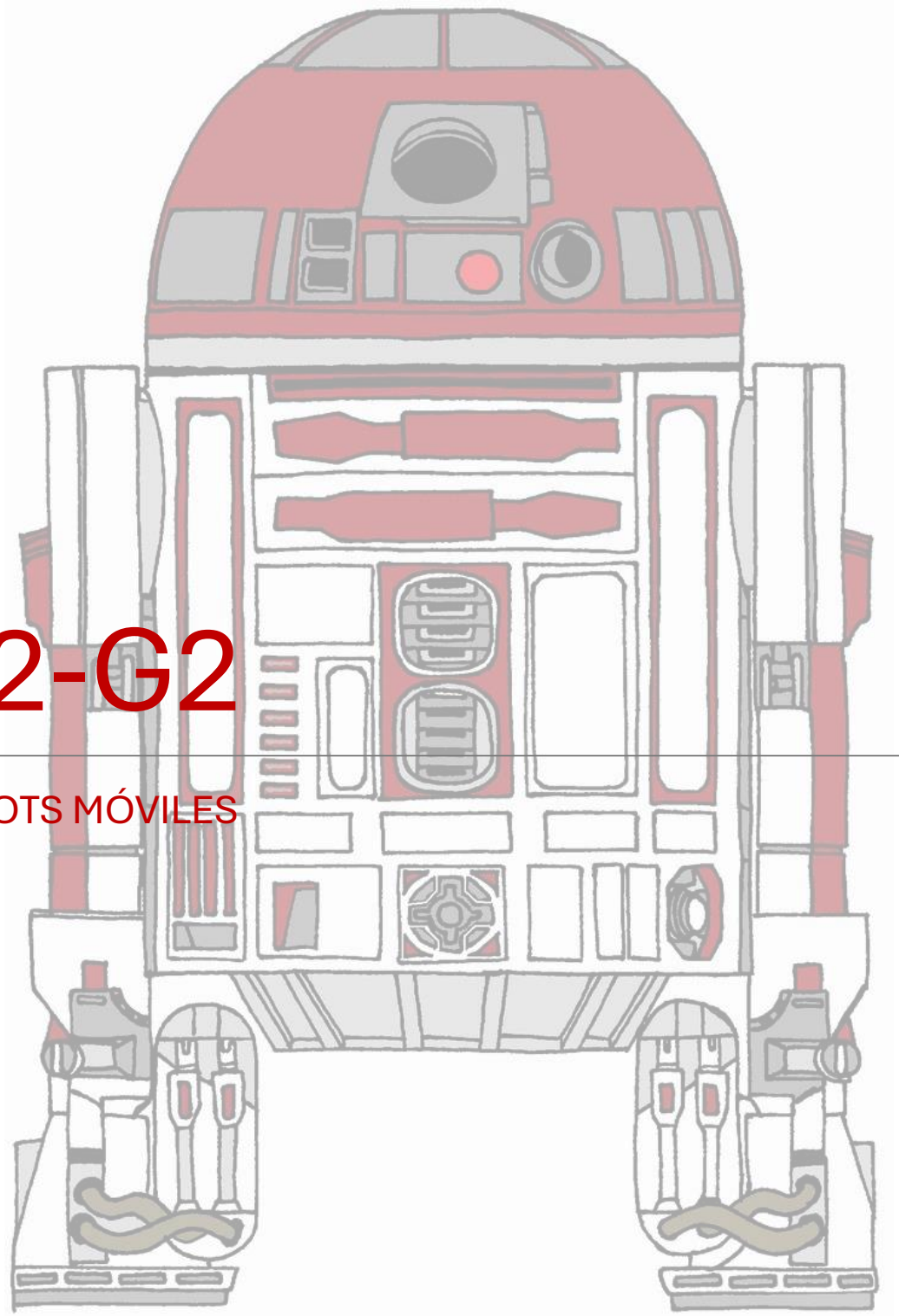


R2-G2

ROBOTS MÓVILES

José Llorens M. y Asahel Hernández T.
UA | INGENIERÍA ROBÓTICA

INDICE

GITHUB DEL PROYECTO

https://github.com/Hispano1919/security_robot

En el repositorio de Github se encuentran los prerrequisitos de instalación y las instrucciones de uso, así como una breve explicación del *workspace* y los códigos fuente.

1. Planteamiento del proyecto e idea inicial

Ilustración 1. Logo del proyecto R2-G2: Robot Guardián

Se plantea un proyecto para la implementación de un robot guardián, llamado R2-G2, en honor a R2-D2 de *StarWars*.

El robot debe ser capaz de navegar por el entorno de manera autónoma, seguir a una persona e interactuar con ella de manera cómoda, patrullar determinadas zonas y explorar entornos nuevos.

Este proyecto podría servir de base para una futura aplicación robótica con más capacidades, como transporte de objetos, integración de un robot manipulador, trabajo colaborativo, etc.

Se pretenden implementar diferentes algoritmos de control al robot móvil Turtlebot 2 físico del laboratorio. Estos algoritmos permitirían al robot seguir diferentes instrucciones de movimiento tanto por comandos de voz como por control visual, además de los controles tradicionales de navegación y algunos métodos nuevos empleados.

Por una parte, el control por voz permite dar determinadas instrucciones al robot, algunas de ellas como: “sígueme” , “ve a la cocina” o “patrulla el área 1”, darían paso a la ejecución de funciones que permiten el seguimiento de dichas instrucciones.

Por otra parte, el control visual sirve de soporte para los comandos por voz, dotando al robot de herramientas como el reconocimiento de gestos por cámara o reconocimiento de QRs, que permiten identificar si se ha llegado al lugar deseado, así como la detección e identificación de personas y el posterior seguimiento de las mismas, si se precisa.

Además, para facilitar la interacción del usuario con el robot, se desarrolla e implementa una interfaz gráfica mediante la cual el usuario puede observar diferentes paneles. Estos muestran una serie de imágenes del mapa del entorno y la imagen que está captando la cámara del robot en cada momento. Además, esta interfaz también permite al usuario enviar instrucciones concisas al robot mediante el uso de botones.

Una versión más simplificada de la interfaz permite, mediante línea de comandos, una interacción similar a la interfaz de voz. En la terminal de comandos el usuario escribe qué desea que el robot haga, y la interfaz interpreta esos comandos y los traduce a órdenes concretas para el robot.

Se ha implementado una interfaz de lanzamiento para facilitar lanzar la aplicación, con diferentes modos de uso para una mayor flexibilidad en el desarrollo.

2. Workspace y directorio de *scripts*

```
security_robot
├── CMakeLists.txt
├── config
│   └── ...
├── images
│   └── ...
├── launch
│   └── ...
├── nav_maps
│   └── ...
├── obj_models
│   ├── person_walking
│   │   ├── ...
│   │   ...
│   ├── qr_cocina
│   │   ├── ...
│   │   ...
│   ├── qr_habitacion
│   │   ├── ...
│   │   ...
│   ├── qr_salon
│   │   ├── ...
│   │   ...
│   └── qr_wc
│       ├── ...
│       ...
├── output_files
│   ├── restricted_maps
│   ...
├── package.xml
├── README.md
├── requirements.txt
├── src
│   ├── UI_launcher.py
│   ├── run.sh
│   ...
├── trained_models
│   └── ...
└── worlds
    └── ...
```

En el repositorio Github está explicado carpeta por carpeta qué contiene cada directorio del paquete.

DIRECTORIO DE SCRIPTS

	NOMBRE	DESCRIPCIÓN
APLICACIÓN	APP_config.py	Contiene todos los macros y definiciones globales utilizadas en la aplicación.
	APP_main.py	Contiene la máquina de estados y la gestión principal de la aplicación.
	APP_map_name_updater.py	Actualiza el macro MAP_NAME en APP_config.py.
	APP_map_processor.py	Realiza la segmentación del mapa en áreas.
	APP_move_person.py	Permite mover el modelo de la persona en las simulaciones de gazebo.
MOVE ROBOT	MR_follow_person.py	Ejecuta el nodo de seguimiento de personas.
	MR_move_to_point.py	Manda al robot un goal de navegación de un punto específico del mapa.
	MR_move_to_qrWaypoint.py	Manda al robot a un waypoint QR guardado previamente.
	MR_patrol_area.py	Patrulla dentro de un área específica.
	MR_patrol_route.py	Patrulla pasando por todas las áreas establecidas.
	run.sh	Fichero bash que lanza la aplicación
ROBOT VISION	RV_pDetector_mediapipe.py	Detección de personas usando Mediapipe.
	RV_pDetector_mediapipeGPU.py	Detección de personas usando Mediapipe y GPU.
	RV_pDetector_MoveNet.py	Detección de personas usando MoveNet de Tensorflow.
	RV_QR_finder.py	Ejecuta la búsqueda de QRs por el mapa.
USER INTERFACE	UI_bash_interface.py	Interfaz de terminal para introducir comandos de manera escrita.
	UI_gui_interface.py	Interfaz visual que permite un control más directo.
	UI_hand_control.py	Control por gestos ejecutado cuando se está siguiendo a una persona.
	UI_launcher.py	Interfaz visual para lanzar y configurar la aplicación de forma cómoda sencilla.
	UI_voice_control.py	Control por voz para interactuar con el robot.

3. Uso

La interfaz de lanzamiento permite ejecutar de manera sencilla e intuitiva la aplicación. Para ejecutarla, desde el directorio *src* del paquete, se debe lanzar la interfaz mediante un terminal utilizando python:

```
python UI_launcher.py
```

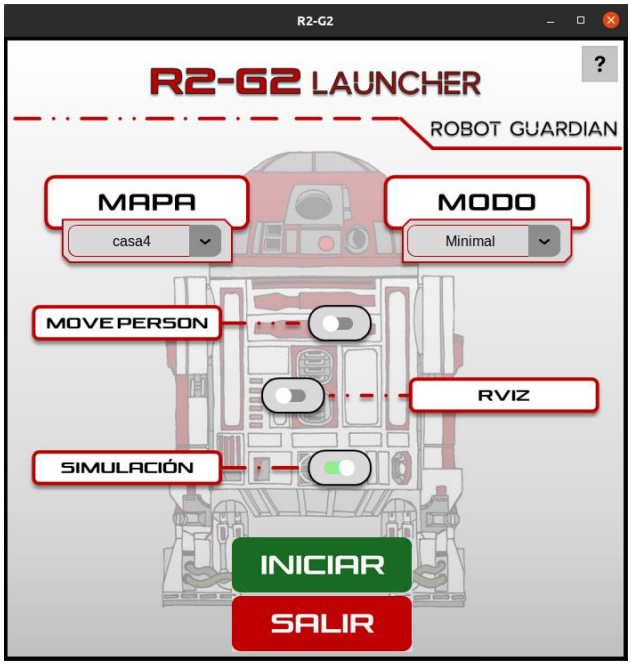


Ilustración 2. Interfaz de lanzamiento de la aplicación.

En el repositorio Github se explica de manera más detallada para qué sirve cada opción.

También es posible lanzar la aplicación sin usar la interfaz, haciendo uso del archivo run.sh y utilizando las opciones que proporciona.

```
./run.sh help
```

Esto mostrará por terminal las opciones existentes y su uso.

El lanzador incluye una ventana de opciones e información. En esta ventana se encuentra un botón que abre en el navegador el repositorio de Github, un botón para abrir esta misma documentación desde dentro del repositorio y un botón para instalar las dependencias necesarias mediante el fichero *requirements.txt*.

4. Control por voz

El control por voz se realiza mediante la librería *vosk*. Esta librería permite reconocer palabras mediante un diccionario ya preestablecido. La ventaja de su uso frente a otro tipo de reconocimiento es que se puede realizar de manera local sin necesidad de una conexión a internet.

Para poder aplicar el control por voz, se han establecido los siguientes comandos:

Lista de comandos de voz y sus acciones:

1. **"sígueme" / "ven" / "conmigo":** El robot activa el modo seguimiento y comienza a seguir a la persona.
2. **"quédate aquí" / "quieto" / "para"**: El robot se queda en su posición actual y detiene el seguimiento.
3. **"patrulla [ubicación]"**: El robot patrulla el área específica mencionada en la ubicación.
4. **"patrulla zona" / "patrulla la ruta"**: El robot patrulla la ruta predefinida.
5. **"me dirijo a [ubicación]"**: El robot se mueve hacia la ubicación especificada.
6. **"vuelve a la estación" / "descansa"**: El robot se dirige a la estación de carga para reposo o recarga.
7. **"no mires" / "apaga cámara":** El robot desactiva la detección visual.
8. **"mírame" / "enciende cámara"**: El robot activa la detección visual.
9. **"no oigas" / "no escuches"**: El robot desactiva la escucha de comandos de voz.
10. **"adiós"**: El robot se apaga y detiene su funcionamiento.
11. **"indique su nombre" (cuando se requiere identificación)**: El robot solicita el nombre del usuario para identificación.
12. **"indique su número de identificación" (tras decir el nombre)**: El robot solicita la clave de identificación del usuario.

Palabras de activación:

Para activar la escucha del robot, se deben usar las siguientes palabras clave:

- **“robot” o “robot y "hola", "escucha", "escúchame", "oye"**: Habilita la escucha activa para recibir comandos.

5. Búsqueda y detección de QRs

La búsqueda y detección de QRs se ha implementado para permitir al robot una interacción con el entorno. Mediante el uso de unos QRs con un *tag* asociado, el robot es capaz de asociar nombres de localizaciones y sus posiciones. Si se inicia el proceso de búsqueda de QRs, el robot se moverá de manera aleatoria por el mapa hasta detectar un QR. Si el QR no está guardado previamente en el *log* correspondiente, el robot se acerca hasta una distancia determinada y se guarda dicha posición y orientación y el *tag* del QR detectado.

Después, continúa la búsqueda por el entorno hasta que el usuario le indique otra tarea a realizar.

La detección de QRs se realiza mediante OpenCV y la librería *pyzbar*. Cuando un QR es detectado, se calcula el área que este QR ocupa dentro de la imagen y se acerca al robot hacia el QR hasta que ocupa un porcentaje determinado. De esta manera el robot queda posicionado cerca del punto de interés.

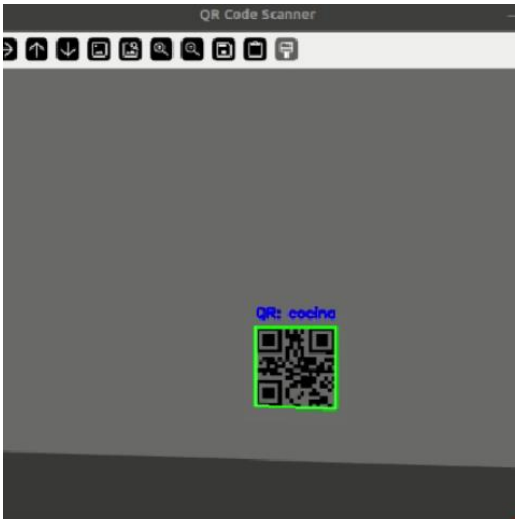


Ilustración 3. Detección del QR cocina.

Cuando posteriormente el usuario ordene al robot dirigirse hacia una localización concreta, se buscará en la base de datos si existe algún *waypoint* QR detectado para esa posición y se moverá al robot mediante el *stack* de navegación hacia la pose asociada a dicha localización.

6. Control Visual

Detección y seguimiento de personas

Para la implementación del módulo de detección y seguimiento de personas se han desarrollado tres códigos diferentes, estado **FollowPersonState**, incluido en la máquina de estados general del proyecto, el nodo **FollowPersonNode**, incluido en **MR_followPerson.py** y el nodo **PersonDetector**, incluido en el script **RV_pDetector_mediapipe.py**.

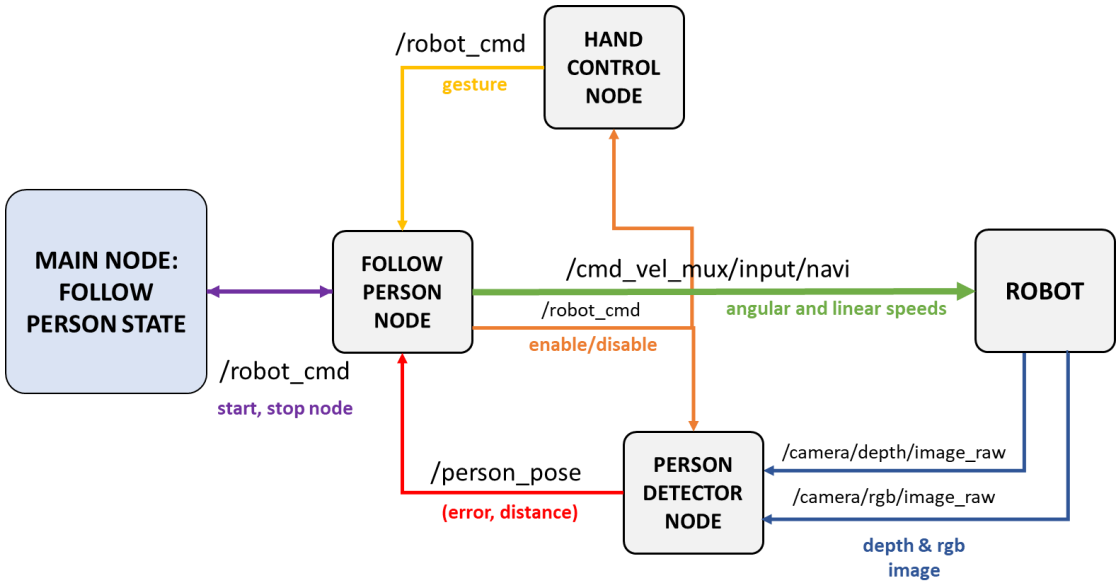


Ilustración 4. Estructura del módulo de detección y seguimiento de personas.

Detección de personas en la imagen

Modelo de detección

La detección de personas se planteó inicialmente empleando la red neuronal Yolo y realizando un entreno para la tarea, sin embargo, se descartó rápidamente debido al costo computacional y de tiempo que requeriría un entreno específico para las personas que se pretendía realizar el seguimiento.

En paralelo, se intentó implementar un seguimiento de QRs que permitiese al robot identificar a la persona mediante un QR (en una etiqueta en la ropa o mediante una imagen en un dispositivo móvil). Esta idea fue descartada para QRs en movimiento debido a las limitaciones existentes a la hora de reconocer aquellos que se encontraban lejos de la cámara (debido a la resolución de la imagen y el tamaño relativo del QR respecto al plano imagen). Estos métodos se pretendían usar para solventar el problema de detección indeseada de transeúntes, permitiendo al robot detectar únicamente la persona objetivo.

Finalmente se ha optado por usar modelos preentrenados ya existentes, como el detector de poses de *Mediapipe*, que permite detectar de manera general una persona en la imagen y obtener ciertos puntos característicos. Este método, aunque más generalista, cumple con las expectativas impuestas.

El principal problema que se plantea a la hora de emplear una detección con un modelo preentrenado de manera general es que el robot no es capaz de distinguir si la persona en el plano imagen es la que debe seguir o es una persona externa a la tarea.

Usando *Mediapipe*, siempre y cuando la persona detectada inicialmente permanezca en el plano imagen, no se tomarán a otras personas como objeto a detectar, teniendo sus limitaciones cuando la persona está muy cerca o muy lejos del robot, ya que se pierden ciertas características que permiten la detección.

Retardo en el control y aceleración por GPU

El reconocimiento de poses mediante *Mediapipe* introduce un retardo entre la captación de la imagen y la detección de la persona.

Este retardo, en simulación, es aceptable, ya que la captación de la imagen y la actuación posterior se producen de manera inmediata. Sin embargo, al utilizar el robot real, la latencia existente entre la captación de la imagen, la transmisión de la misma vía red inalámbrica al ordenador controlador y la posterior actuación desde el ordenador al robot, sumado al procesado de *Mediapipe*, introduce un retardo en el sistema que dificulta el control del robot en tiempo real.

Sería conveniente poder realizar la detección y el control del movimiento del robot de manera local en el robot para eliminar todos los retrasos inherentes a la comunicación inalámbrica, mejorando así el rendimiento de la aplicación.

Para intentar reducir el retardo introducido por el procesamiento de *Mediapipe* se han desarrollado diferentes métodos de detección con aceleración por GPU, los más satisfactorios usando el modelo preentrenado *MoveNet* de Tensorflow y el método *PoseLandmarker* de *Mediapipe*. Sin embargo, para el uso efectivo de estos métodos se requiere una combinación de versiones específicas para Tensorflow, CUDA, cudNN y OpenCV que dificultaba la instalación del proyecto. Debido a esto la aceleración introducida por la GPU, aunque notable, no compensa la inestabilidad de compatibilidades requerida. Es por eso por lo que se ha dejado como método por defecto el detector de poses de *Mediapipe* ya plenamente integrado sin aceleración por GPU.

Nodos de ROS implementado

El nodo **PersonDetector** es el encargado de realizar la detección de personas en la imagen. Permanece lanzado desde el inicio de la aplicación, pero en estado de espera, para ahorrar recursos mientras no se necesita la detección. Cuando el nodo recibe un mensaje de activación por el *topic* **/robot_cmd**, este se suscribe a los *topics* de la cámara (**/camera/rgb/image_raw** y **/camera/depth/image_raw**) y se comienza la detección activa mediante los *callbacks* asociados a los *topics*. Este mensaje es enviado por el nodo *main* cuando el robot entra en el estado de seguimiento de personas. Cuando acaba este estado, se envía un mensaje de desactivación para que el nodo PersonDetector se desuscriba de los *topics* de las cámaras. Se ha decidido utilizar este método para aprovechar la modularidad de ROS y así saturar el nodo principal (nodo *main*) con tareas de procesamiento de imágenes que retrasen el flujo de la aplicación.

Dado que las imágenes RGB y de profundidad se obtienen por dos *topics* y *callbacks* diferentes, si no se sincronizan, las imágenes no se corresponden a la hora de

obtener la profundidad del punto de la cadera obteniendo valores incorrectos. La sincronización se consigue mediante el método de ROS *ApproximateTimeSynchronizer*.

```
self.image_sub = Subscriber(TOPIC_RGBCAM, Image)
self.depth_sub = Subscriber(TOPIC_DEPTHCAM, Image)
ats = ApproximateTimeSynchronizer([self.image_sub, self.depth_sub],
queue_size=10, slop=0.05)
ats.registerCallback(self.callback)
```

La detección de poses obtiene una serie de puntos característicos (*landmarks)*, y usando uno de ellos, concretamente la cadera izquierda, se calcula el error con respecto al eje vertical central del plano de la imagen. A su vez, mediante la cámara de profundidad del Turtlebot se obtiene la distancia a la que el robot se encuentra de la persona tomando el valor de la imagen de profundidad correspondiente al punto obtenido en los *landmarks*.

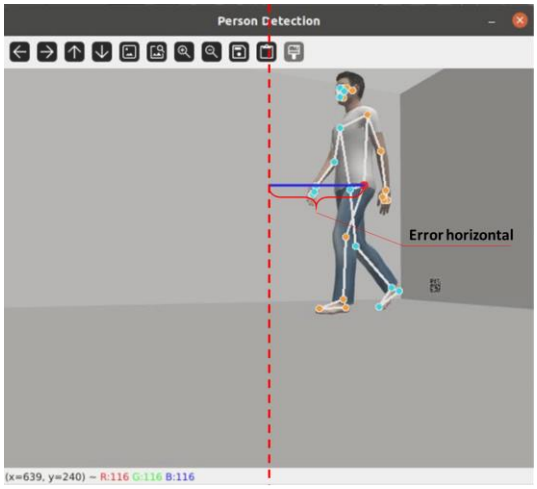


Ilustración 5. Error horizontal y *landmarks detectados*.

Una vez obtenidos estos valores, se envían a través del *topic* **/person_pose**, donde el nodo **FollowPersonNode***,* lanzado por el estado del nodo *main* **FollowPersonState** , recoge el mensaje para aplicar la actuación pertinente sobre el movimiento del robot.

Seguimiento de personas

Control y actuación

Para realizar el seguimiento de la persona detectada, el robot debe tratar de mantener la persona centrada en el plano imagen, a la distancia deseada. Dado que el turtlebot2 permite el control tanto en velocidad lineal como angular, se realiza un control sobre ambas de manera simultánea.

Si no se está detectando ninguna persona en la imagen, y está en el nodo **FollowPersonNode**, se envía al robot únicamente una velocidad angular constante para que rote sobre sí mismo. Cuando se detecta a una persona, se efectúa el control utilizando los valores de error y profundidad recibidos por el *topic*.

El control de la velocidad angular, que permite centrar el objetivo con el plano imagen se realiza de manera proporcional:

```
if abs(error_theta) > CENTER_TOLERANCE_X and error_theta != -1:

      twist.angular.z = -ANGULAR_GAIN * error_theta
      # Saturacion del valor para no exceder el maximo
      if twist.angular.z > MAX_WSPEED:
            twist.angular.z = MAX_WSPEED
      elif twist.angular.z < -MAX_WSPEED:
            twist.angular.z = -MAX_WSPEED

      self.last_twist.angular.z = twist.angular.z
```

Se satura el valor de velocidad angular obtenido tras el control para no exceder los límites máximos permitidos por el turtlebot2. El control de velocidad angular se aplica para corregir el error entre el punto de la cadera y el centro de la imagen cuando este error supera un umbral, de manera que es más flexible a la hora de realizar los movimientos.

La velocidad lineal se controla empleando el error entre la profundidad y la distancia deseada. Esta distancia umbral es un valor dinámico (con un valor base por defecto de 1.5 metros), ya que mediante el control por gestos se puede influir en esta distancia para que el robot se acerque o aleje de la persona. Se trata también de un control proporcional al error de la distancia, aplicando una saturación al valor de velocidad resultante para no sobrepasar los límites. Si el error detectado es menor a un umbral de seguridad, se envía una velocidad lineal nula para evitar que el robot choque.

Posteriormente, gracias al control de distancia, cuando esa distancia se incremente por encima del umbral, si el robot se encuentra por debajo de la distancia deseada, retrocederá hasta situarse en ella:

```
distance_error = pixel_depth - (TARGET_DISTANCE + self.dynamicDist)
#Error respecto a la distancia deseada

self.last_error = distance_error
# Control lineal para mantener la distancia
twist.linear.x = LINEAR_GAIN * distance_error

if(abs(distance_error) < DISTANCE_ERROR):
      twist.linear.x = 0

if twist.linear.x  > MAX_VSPEED:
      twist.linear.x  = MAX_VSPEED
elif twist.linear.x  < -MAX_VSPEED:
      twist.linear.x  = -MAX_VSPEED
```

Las velocidades calculadas se envían directamente por el *topic* **/cmd_vel_mux/input/navi**, que actúa sobre el control de movimiento del robot, por lo que el robot no está atendiendo a la detección de obstáculos como si ocurre cuando se envían comandos de movimiento mediante el *stack* de navegación.

Si la persona sale del plano de imagen el robot volverá buscar a la persona rotando sobre sí mismo. Esta rotación se realiza hacia la dirección donde se detectó por última vez la persona, de manera que la re-detección sea más rápida.

Control por gestos

Se ha implementado una detección por gestos para interactuar con el robot. Este control por gestos se ha desarrollado en un *script* externo al principal para que, de manera similar a la detección de personas, la detección de gestos no afecte al flujo principal del programa. Se realiza mediante *Mediapipe,* como se introdujo en la práctica 1 de la asignatura.

El control por gestos permite modificar la distancia entre el robot y la persona cuando el robot está en modo seguimiento, además de implementar un gesto para apagar la aplicación por defecto.

Gestos reconocidos

Inicialmente se plantearon los siguientes gestos y sus funciones:

- **Puño cerrado**: Detener seguimiento.
- **Palma abierta**: Iniciar seguimiento.

Sin embargo, dado que estas órdenes eran redundantes y podían ser ejecutadas mediante los comandos por voz, se plantearon modificar las funciones por aumentar y disminuir la distancia entre el robot y la persona.

Una vez implementado este control, dado que los gestos puño cerrado y palma abierta, son posturas naturales de la mano, se producían muchas detecciones no deseadas, en las que el usuario o alguna otra persona en el campo de visión del robot, realizaban el gesto de manera no intencional, pero siendo aceptadas por el robot como control por gestos. Se concluyó que era necesario cambiar los gestos detectados a unos gestos menos naturales para minimizar el control involuntario.

Se escogieron estos nuevos gestos y funciones:

1. **V de victoria** (Índice y corazón extendidos, demás dedos flexionados): Disminuir distancia con la persona (Avanzar).
2. **V de victoria inversa** (dedos apuntando hacia abajo): Aumentar distancia (Retroceder).
3. **Gesto OK** (Índice y pulgar tocándose, demás dedos extendidos): Resetear distancia a la distancia por defecto (1.5 m).
4. **Rock ‘n’ Roll** (Dedo índice y meñique extendido, demás flexionados): Dejar de seguir a la persona.

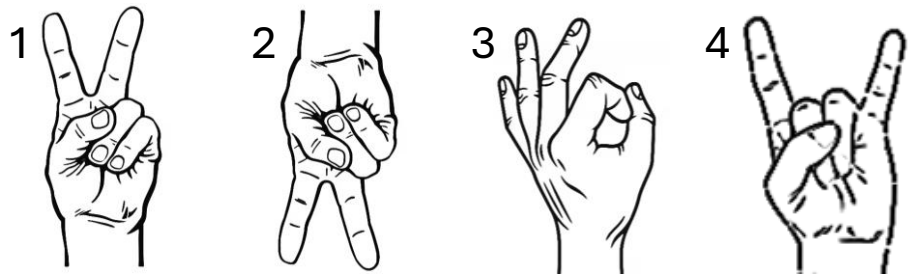


Ilustración 6. Gestos reconocibles por el control por gestos.

Nodo de ROS implementado

Para el control por gestos se ha implementado un nodo **HandControlNode** que permite, de manera paralela al nodo *main*, admitir órdenes mediante gestos. Cuando por el *topic* **/robot_cmd** se activa la detección, si el nodo está escuchando, se suscribirá al *topic* de la cámara RGB.

En el *callback* de los mensajes recibidos por el *topic* de la cámara se ejecuta la detección de los gestos, empleando los *landmarks* de los dedos presentes en la imagen. Se realizan comprobaciones entre las coordenadas ***y*** de los dedos para comprobar si están extendidos (y en qué dirección) y también se calcula la distancia entre la punta del dedo índice y el dedo pulgar para poder detectar el gesto de OK.

Por ejemplo, para el gesto OK:

```
thumb_tip = hand_landmarks.landmark[THUMB[0]]
index_tip = hand_landmarks.landmark[INDEX[0]]

# Calcular la distancia entre la punta del pulgar y la punta del índice
thumb_index_distance = self.calculate_distance(thumb_tip, index_tip)

# Verificar que los otros dedos no interfieran con el gesto
middle_extended = self.is_finger_extended_upwards(hand_landmarks.landmark,
*MIDDLE)
ring_extended = self.is_finger_extended_upwards(hand_landmarks.landmark,
*RING)
pinky_extended = self.is_finger_extended_upwards(hand_landmarks.landmark,
*PINKY)

# Si la distancia entre el pulgar y el índice es pequeña y los otros dedos
están recogidos
if thumb_index_distance < 0.05 and middle_extended and ring_extended and
pinky_extended:
      gesture = RESET_DIST_CMD
      return gesture
```

Una vez el gesto es detectado, se transmite este gesto al nodo *main* mediante el *topic* **/robot_cmd**. En el node **FollowPersonNode** se toma el gesto detectado y se incrementa, decrementa o resetea la distancia dinámica:

```
if msg.data == MOVE_CLOSER_CMD:
      self.dynamicDist = self.dynamicDist - 0.5
elif msg.data == MOVE_AWAY_CMD:
      self.dynamicDist = self.dynamicDist + 0.5
elif msg.data == RESET_DIST_CMD:
      self.dynamicDist = 0
```

La distancia dinámica se aplica luego al control de movimiento, añadiéndola a la distancia por defecto:

```
distance_error = pixel_depth - (TARGET_DISTANCE + self.dynamicDist)
```

Diagrama de Control Visual

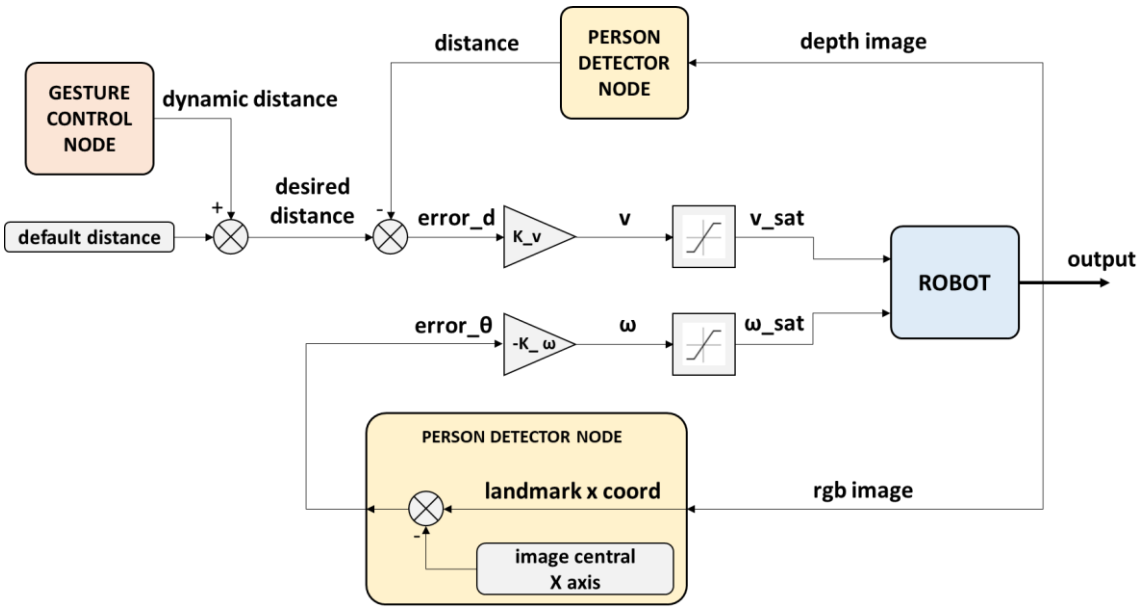


Ilustración 7. Diagrama de bloques del Control Visual.

En el diagrama anterior se puede apreciar cómo actúa el control visual sobre el robot. La velocidad lineal y angular son introducidas al controlador de bajo nivel que incorpora el Turtlebot2. Las variables implicadas son las siguientes:

- **Dynamic distance**: Distancia dinámica controlada por gestos, se suma (o resta) a la distancia por defecto.
- **Distance:** Distancia obtenida en la imagen de profundidad desde el robot hasta el punto de la cadera de la persona.
- **Default distance:** Distancia por defecto que el robot debe mantener.
- **Desired distance:** Resultado de aplicar la distancia dinámica a la distancia default.
- **Depth image:** Imagen de profundidades captada por la cámara del turtlebot2.
- **RGB image:** Imagen RGB captada por la cámara del turtlebot2.
- **Error_d:** Error en distancia existente entre la distancia de referencia y la actual.
- **V:** Velocidad lineal a aplicar según el control proporcional.
- **K_v:** Ganancia proporcional de la velocidad lineal.
- **V_sat:** Velocidad lineal resultante de aplicar la saturación (si sobrepasa el umbral).
- **Error_θ:** Error entre el eje vertical central de la imagen y el punto detectado.
- **ω:** Velocidad angular resultante del control proporcional.
- **K_ω:** Ganancia proporcional para la velocidad angular.
- **ω_sat:** Velocidad angular resultante de aplicar la saturación (si sobrepasa el umbral).
- **Landmark x coord:** Coordenada x del punto de la cadera de la persona detectada.
- **Image central X axis:** Anchura en pixeles de la imagen / 2.

7. Exploración del entorno y mapeado

La exploración se realiza utilizando el paquete **explore_lite** ya implementado para ROS. El paquete **explore_lite** de ROS está diseñado para la exploración autónoma de entornos desconocidos mediante un enfoque basado en fronteras. Este nodo se suscribe a mensajes de tipo nav_msgs/OccupancyGrid y map_msgs/OccupancyGridUpdate para construir un mapa y detectar las fronteras entre áreas exploradas y no exploradas.

A diferencia de otros paquetes similares, **explore_lite** no crea su propio *cost_map*, lo que facilita su configuración y lo hace más eficiente en términos de recursos. En su lugar, se suscribe a mapas existentes y envía comandos de movimiento al nodo **move_base** para navegar hacia las fronteras detectadas.

Las primeras pruebas se realizaron utilizando el paquete Rosbot de Husarion, ya que ya tenía un tutorial completamente detallado de cómo utilizar **explore_lite**. Posteriormente se procedió a la adaptación y configuración para TurtleBot2.

Para configurar **explore_lite** en un TurtleBot2, hay que seguir estos pasos:

1. **Configuración de move_base**:
    - Nodo **move_base** correctamente configurado y funcionando en su TurtleBot2. Debería poder navegar manualmente utilizando **move_base** a través de RViz.
2. **Habilitar la navegación en áreas desconocidas**:
    - Es necesario que el robot pueda navegar a través de espacios desconocidos en el mapa. Para ello, se configura planificador para permitir la planificación en áreas desconocidas, es decir, la opción *allow_unknown* esté habilitada.
3. **Configuración del mapa de costos**:
    - Si se desea utilizar el *cost map* proporcionado por **move_base**, debe habilitarse el seguimiento de espacios desconocidos estableciendo el parámetro *track_unknown_space* en true. Esto permite que el mapa de costos tenga en cuenta las áreas no exploradas, esencial para la exploración autónoma.
4. **Lanzar explore_lite**:
    - Una vez que move_base esté configurado correctamente, se puede lanzar el nodo explore_lite. En esta aplicación esto se hace desde la interfaz de lanzamiento seleccionando el modo Exploración.

8. Segmentación del entorno

Para realizar una correcta gestión del entorno debemos poder distinguir entre las distintas habitaciones y pasillos, de forma que podamos indicarle al robot si debe ir o patrullar una determinada estancia.

Primeramente, se intentó emplear una librería que tenía varios métodos implementados. Sin embargo, no se consiguió configurar correctamente, por lo que se determinó que el mejor método para nuestra posibilidad era emplear métodos de transformación de la imagen.

Para ello se requiere el mapa .pgm del área y el yaml para poder calcular la equivalencia entre los pixeles y las coordenadas del mapa. Se requiere convertir la imagen a blanco y negro binarizado para posteriormente realizar un proceso de erosionar las áreas y luego dilatarlas.

```
 # Convertir la imagen a blanco y negro
    _, binary_image = cv2.threshold(image, 240, 255, cv2.THRESH_BINARY)

    # Erosionar para aislar habitaciones
    kernel = np.ones((3, 3), np.uint8)
    eroded_image = binary_image.copy()
    for _ in range(39):
        eroded_image = cv2.erode(eroded_image, kernel)
    for _ in range(39):
        eroded_image = cv2.dilate(eroded_image, kernel)
```

Una vez hecho esto se obtiene las regiones se crea un mapa de colores que nos servirá para calcular los centroides de cada estancia, mediante una serie de fórmulas y considerando el punto 0,0 la posición del robot dada por el .yaml.

Luego se crea una máscara para cada una de las áreas y se superponen en el mapa pgm. Se guarda el mapa de cada área concreta de forma que podemos hacer que el robot navegue por el área específica que se desee.

Dentro del bucle anterior se guarda las coordenadas de los centroides, el color asignado a esa área y la ruta al mapa del área segmentada.

```
# Calcular los centroides de cada región conectada y pintar la región
    centroids = []
    region_colors = []  # Almacenar los colores asignados a las regiones
    for label in range(1, num_labels):  # Ignorar el fondo (label 0)
        # Crear una máscara para la región actual
        region_mask = (labels == label).astype(np.uint8)

        # Asignar un color aleatorio para cada región
        color = np.random.randint(0, 255, size=3).tolist()  # Genera un color
aleatorio
        region_colors.append(color)  # Guardar el color de la región
```

```
        # Pintar la región con el color correspondiente
        colored_image[region_mask == 1] = color

        # Calcular los momentos de la región
        moments = cv2.moments(region_mask)
        if moments["m00"] != 0:
            # Calcular las coordenadas del centroide en píxeles
            c_x_pixel = int(moments["m10"] / moments["m00"])
            c_y_pixel = int(moments["m01"] / moments["m00"])

            # Convertir a metros usando la resolución y el origen
            c_x_meter = c_x_pixel * resolution + origin_x
            c_y_meter = origin_y + (image.shape[0] - c_y_pixel) * resolution

            # Almacenar el centroide
            centroids.append((c_x_meter, c_y_meter))

            # Marcar el centroide en rojo en la imagen
            cv2.circle(colored_image, (c_x_pixel, c_y_pixel), 5, (0, 0, 255),
-1)  # Color rojo en BGR
```

Finalmente, se obtendrá los siguientes elementos que se usarán para la navegación y la patrulla.

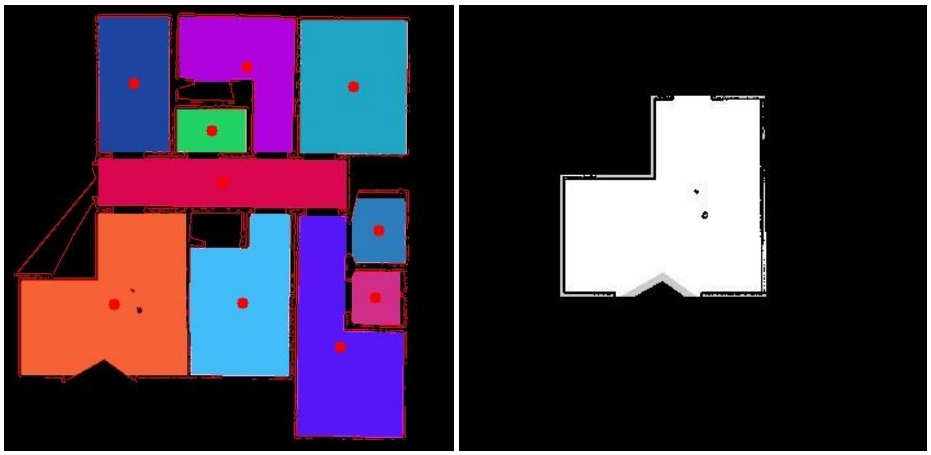


Ilustración 8. Mapa segmentado por áreas y pgm de una de las áreas.

9. Navegación

Una funcionalidad implementada ha sido la de ser capaz de enviar ordenes de movimiento a coordenadas y orientaciones especificas al robot, mediante el módulo move_base.

```
def mover_a_goal(self, x, y):
        # Si el cliente move_base esta activo
        if self.clientAvailable:
            goal = MoveBaseGoal()
            goal.target_pose.header.frame_id = "map"
            goal.target_pose.header.stamp = rospy.Time.now()
            goal.target_pose.pose.position.x = x
            goal.target_pose.pose.position.y = y
            goal.target_pose.pose.orientation.w = 1.0
            self.client.send_goal(goal)
            self.client.wait_for_result()
            if self.client.get_state() == actionlib.GoalStatus.SUCCEEDED:
                return 'succeeded'
            else:
                return 'aborted'
        else:
            return 'aborted'
```

Esto junto con la capacidad de poder calcular las coordenadas de un píxel determinado, nos permite que desde una interfaz pulsando sobre un pixel accesible del mapa podemos indicarle que se dirija a esa posición. Se ha de pulsar dos veces sobre el mapa, el primer click indica la posición de destino, el segundo indica la orientación que debe tener el robot cuando llegue.

El método empleado para la conversión entre pixeles y coordenadas es el mismo que se usó para para calcular las coordenadas de los centroides:

```
def convertir_a_coordenadas(self, px, py):
        x = px * self.resolucion + self.origen[0]
        y = self.origen[1] + (self.mapa_binario.shape[0] - py) *
self.resolucion
         return x, y
```

Para calcular la orientación se hace simplemente calculando el ángulo theta entre ambos puntos:

```
def calcular_orientacion(self, x1, y1, x2, y2):
        """
        Calcular la orientación (ángulo theta) entre dos puntos (x1, y1) y
(x2, y2).
        """
        delta_x = x2 - x1
        delta_y = y2 - y1
        return math.atan2(delta_y, delta_x)
```

10. Patrullaje

Para la tarea de realizar patrullas, dado el mapa de color y el csv con el color y la ruta al mapa segmentando se ha optado por otro mapa iterativo que al pulsar encima de un área coloreada se llame al script de patrulla. El robot solo patrullara sobre área que se ha seleccionado.

Para ello primeramente se detecta el color del píxel seleccionado y se compara con los colores guardados en csv.

```
def detectar_color_pixel(self, event):
        """
        Manejar el evento de clic y detectar el color del píxel seleccionado.
        """
        x = event.x
        y = event.y

        # Obtener el color del píxel seleccionado
        color_pixel_rgb = tuple(self.mapa_coloreado[y, x])

        # Buscar el área correspondiente al color seleccionado
        ruta_pgm = self.encontrar_area_por_color(color_pixel_rgb,
self.colores_y_rutas)
        if ruta_pgm:
            print("Área seleccionada: {0} -> {1}".format(color_pixel_rgb,
ruta_pgm))

            # Llamar al patrullaje en un hilo separado
            self.ejecutar_patrullaje(ruta_pgm)

        else:
            print("No se encontró un área para el color:
{0}".format(color_pixel_rgb))
```

Desde el script MR_patrol_area.py, se ejecutará el nodo **PatrolAreaNode** se generarán puntos que cumplan que estén en un área accesible, estén a la distancia especificada de otro y que estén a una distancia de seguridad de la pared. Luego se seleccionan 5 puntos aleatorios de entre todos los puntos generados y dado que estos puntos seleccionados son pixeles aún por lo que hay transfórmalos a coordenadas.

Por último, se envían las 5 ubicaciones a la función mover a goal vista anteriormente mediante un bucle. Esto se ejecutará hasta que se recorran los 5 puntos o hasta que una orden del usuario interrumpa la ejecución del patrullaje.

Otro método para realizar labores de patrulla es emplear el nodo **PatrolRouteNode,** en MR_patrol_route.py, que hace el robot recorra las áreas en orden.

El script consiste en tres funciones, siendo la primera la de leer los centroides del csv.

```
def leer_centroides(self, csv_path):
        """Leer los centroides desde un archivo CSV."""
        centroides = []
        with open(csv_path, mode='r') as file:
            reader = csv.reader(file)
            next(reader)  # Saltar la cabecera
            for row in reader:
                x = float(row[0])  # Coordenada X en metros
                y = float(row[1])  # Coordenada Y en metros
                centroides.append((x, y))
        return centroides
```

La segunda se encarga de enviar a la función de movimiento los centroides antes leídos para que el robot se dirija a ellos.

```
def mover_a_centroides(self, centroides):
        """Mover el robot a cada uno de los centroides."""
        for idx, (x, y) in enumerate(centroides):
            rospy.loginfo(f"Moviendo al paso {idx+1} a las coordenadas: ({x},
{y})")
            self.log_pub.publish("[INFO] PATROL NODE: Patroling route...")
            result = self.mover_a_goal(x, y)

            if result:
                self.log_pub.publish("[INFO] PATROL NODE: Centroid reached")
                rospy.loginfo(f"Llegamos al paso {idx+1}")
            else:
                self.log_pub.publish("[INFO] PATROL NODE: Centroid not
reached")
                rospy.logwarn(f"Falló al llegar al paso {idx+1}")

            if self.is_active == False:
                break
            rospy.sleep(2)
```

Y finalmente la función antes vista para usar **move_base.**

Para acceder a esta funcionalidad solamente hay que pulsar el botón de nombre centroides en la interfaz o indicar al robot mediante comandos por voz o texto que debe “patrullar la zona”.

11. Interfaz por línea de comandos

Se ha implementado una interfaz por terminal simple que permite interactuar con el robot de manera similar al control por voz, donde el usuario introduce de manera escrita qué desea que el robot haga. Para hacer la interfaz más atractiva se ha utilizado la librería de Python **rich** que permite una mayor personalización y decoración de la línea de comandos.

La interfaz acepta mensajes complejos, como por ejemplo “ve a la cocina”, de donde interpretará la palabra “cocina” como una localización posible y enviará la orden move_state:cocina por el *topic* **/robot_cmd**.

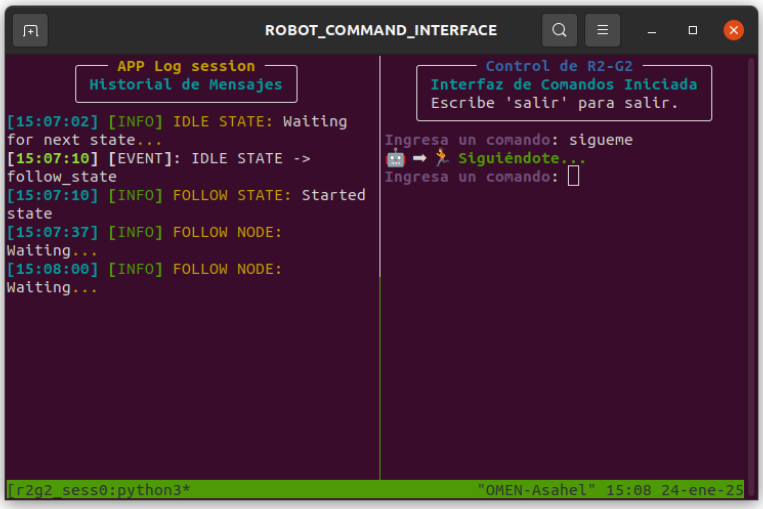


Ilustración 9. Interfaz por línea de comandos.

La lista de comandos que puede efectuar la interfaz es similar a la presentada para el control por voz, con la adición de estos comandos específicos:

Lista de comandos y sus acciones:

1. **"enciende el control por voz":** El robot activa el control por voz.
2. **"busca qrs":** El robot busca QRs por el entorno.
3. **“ayuda”:** Listado de comandos disponibles

La interfaz está compuesta por dos terminales *bash* unidos. En el terminal de la izquierda se pueden visualizar los mensajes que la aplicación va lanzando para notificar al usuario del estado. En el terminal de la derecha se pueden ingresar los comandos deseados.

12. Interfaz Visual

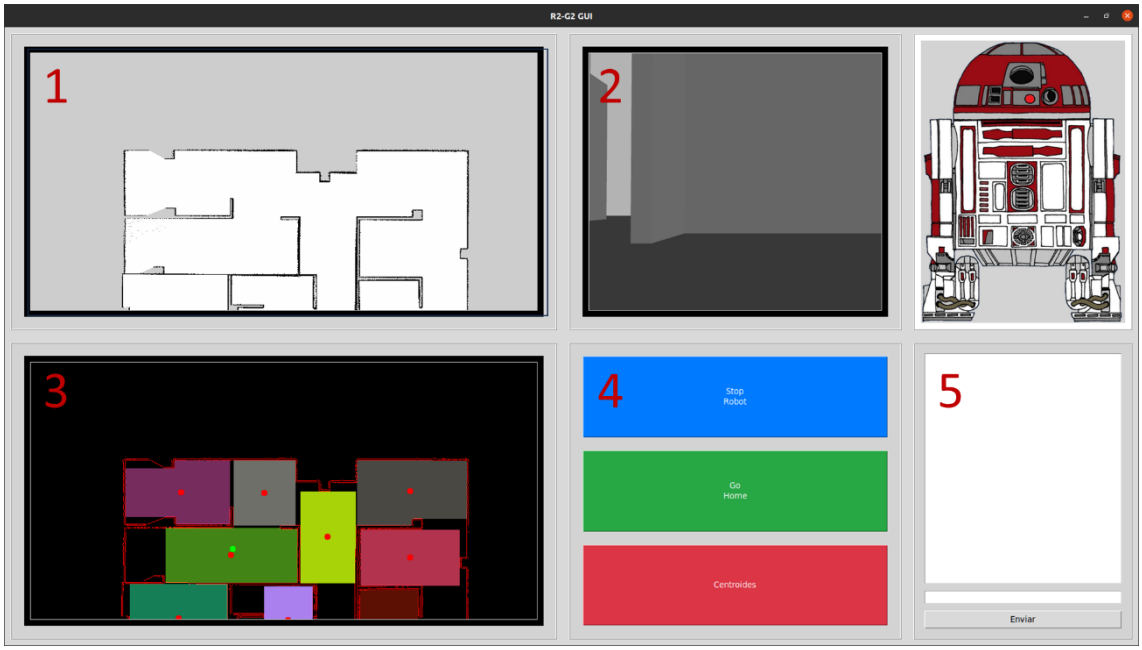


Ilustración 10. Interfaz visual

Para el diseño de la interfaz se ha optado por emplear la librería Tkinter. La interfaz consta de 5 partes:

1. **Mapa PGM del mundo cargado:** permite hacer click en una posición cualquiera y se enviará el robot allí.
2. **Cámara del robot:** POV del robot.
3. **Mapa segmentado por áreas:** Mapa con las áreas detectadas. Si se hace click sobre un área el robot patrullará dentro de dicha área.
4. **Botones:** Control rápido sobre el robot.
5. **Línea de comandos:** Permite enviar algunos comandos al robot.

La interfaz no realiza ningún control directo sobre el robot, simplemente publica en el tópic **/robot_cmd** la orden que el usuario desea realizar. Es el nodo principal el que se encarga de realizar las acciones.

13. Máquina de estados y funcionamiento de la aplicación

La máquina de estados se implementa con la librería smach_ros la cual permite generar y configurar una máquina de estados de manera sencilla y eficaz. Es interesante plantear la implementación de una máquina de estados, pues esto siempre supone una organización centralizada y más organizada de un sistema con diferentes funciones

La máquina de estados que se plantea en esta ocasión sirve para gestionar las señales de los comandos que se reciben tanto por parte de los comandos de voz como por parte de las interfaces y permite hacer una transición de un estado de espera al estado deseado De esta manera, si se trabaja con la máquina de estados de smach y se combina con el uso de mensajes enviados por *topics*, se pueden gestionar todas las instrucciones y llevar a un funcionamiento ordenado y más controlado.

Para esta máquina de estados se han implementado 5 estados:

1. **IDLE WAIT**: Estado servidor. Es el estado inicial y el estado terminal al que transicionan los demás estados. Se encarga de dejar la aplicación en un estado de espera mientras escucha en el *topic* **/robot_cmd** la siguiente orden. Cuando se recibe la orden de cambiar de estado a **FollowPerson** (seguimiento), al estado de **MoveState** (moverse a una posición concreta) o al estado **QRFinder** (búsqueda de QRs) el estado transiciona. Si se recibe la orden de apagar la aplicación, se acaba la máquina de estados y finaliza el proceso.

2. **FOLLOW PERSON:** Este estado se encarga de lanzar el nodo de control visual **FollowPersonNode** para el seguimiento de personas. Cuando el estado acaba transiciona a **IdleWait**.

3. **MOVE STATE:** Estado que ejecuta una tarea de movimiento. Se ejecuta el *stack* de navegación y queda esperando hasta que acaba o se cancela. Dentro de este estado se ejecutan los siguientes nodos según se requiera:

    a. **MoveToQRNode:** Se mueve al waypoint QR guardado en log de posiciones, si esta posición existe. Usado en los comandos del tipo “ve a la cocina”, donde “cocina” sería un waypoint QR previamente indexado.

    b. **MoveToPositionNode:** Se mueve a la posición deseada x,y,w. Se puede ejecutar desde la interfaz gráfica.

    c. **PatrolRouteNode:** Patrulla por todo el mapa moviéndose entre áreas segmentadas.

    d. **PatrolAreaNode:** Calcula un número definido de puntos dentro del área deseada y se mueve entre ellos.

4. **QR FINDER:** Este estado ejecuta el node **QRFinderNode**. Este nodo se mueve por posiciones aleatorias por el mapa mientras ejecuta la búsqueda visual de QRs. Si un QR se encuentra, se acerca hasta una distancia determinada al QR y guarda la posición.

5. **SHUTDOWN:** Acaba la máquina de estados y lanza los procesos de cierre de la aplicación.

La máquina de estados está diseñada para lanzar los nodos en subprocesos aparte. Cuando un nodo debe terminar, se cierra dicho proceso y se envía al nodo principal la señal de nodo finalizado. Esto permite al estado en el que se encuentre transicionar al estado **IDLE WAIT**.

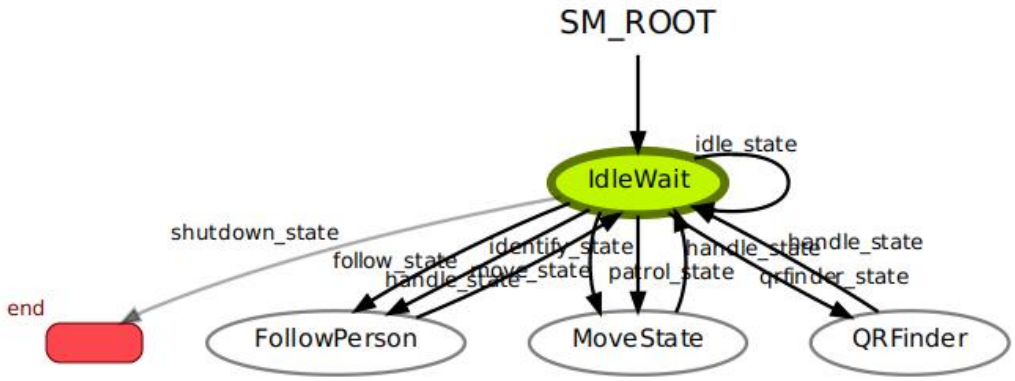


Ilustración 11. Visualización de la máquina de estados mediante SMACH

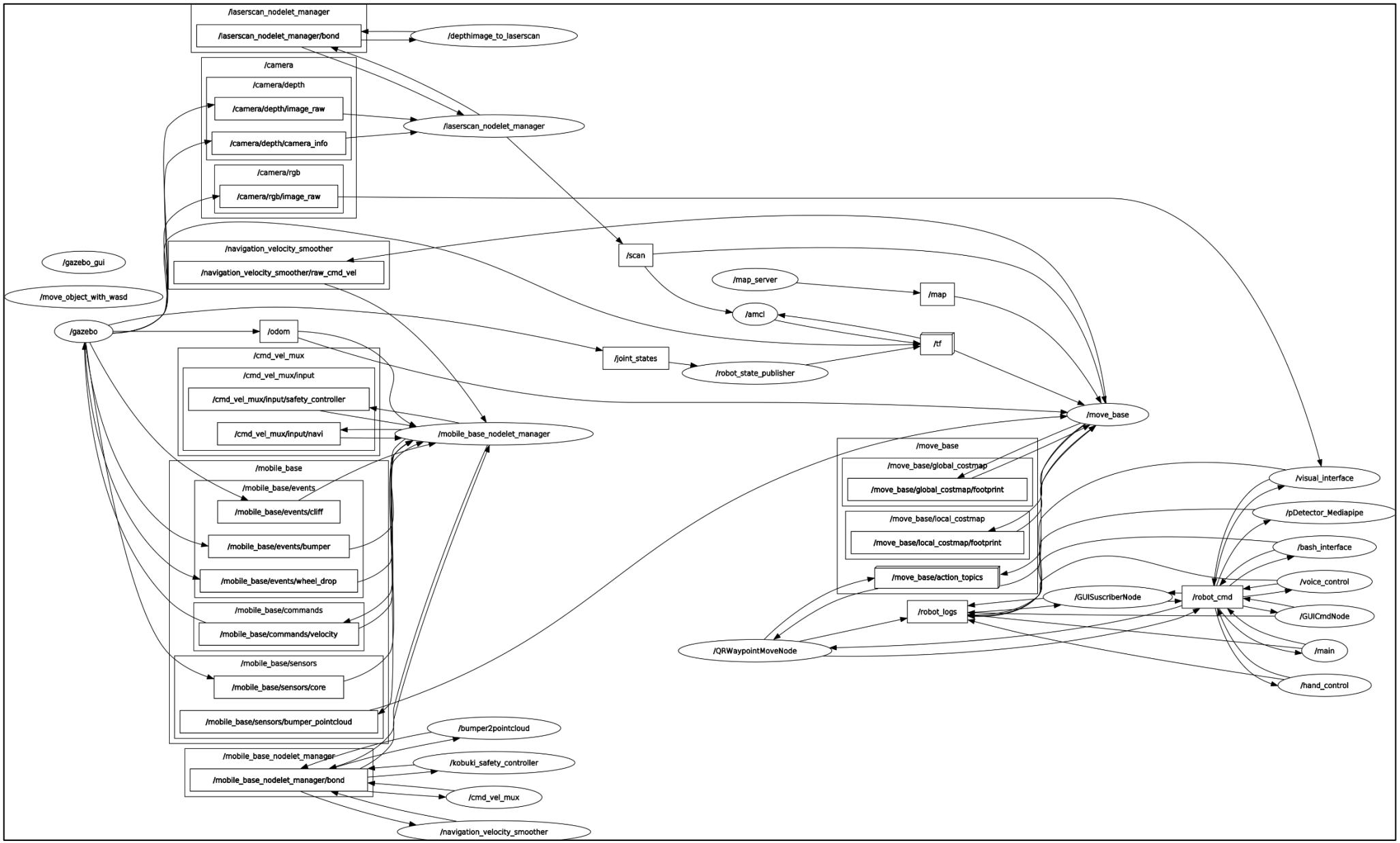
La lógica de implementación que se ha seguido para el funcionamiento de la aplicación es la siguiente

El nodo ***main*** siempre permanece ejecutándose, puesto que es el nodo principal donde se encuentra la máquina de estados.

Conforme se reciben órdenes del usuario mediante el topic **/robot_cmd,** se interpretan y se actúa en consecuencia. Desde los diferentes estados del nodo ***main*** se ejecutan los nodos temporales que ejecutan tareas concisas, a su vez, todas las interfaces de control se comunican con el nodo **main** para transmitir las órdenes.

De esta manera, la interacción con el robot es más fluida y se encuentra centralizada, minimizando el riesgo de conflictos

14. Visualización de nodos y *topics* del proyecto



15. Resultados y pruebas realizadas

SIMULACIÓN

1. **Detección de QRs**
2. **Seguimiento de personas 1**
3. **Seguimiento de personas 2**
4. **Seguimiento de personas 3**
5. **Control por gestos**
6. **Seguimiento, movimiento a Waypoint QR**
7. **Mapeado y exploración**
8. **Mapeado y exploración Rosbot**
9. **Patrullaje área 1**
10. **Patrullaje área 2**
11. **Patrullaje ruta**
12. **Demo diferentes tareas**

ROBOT REAL

1. **Detección de QRs: (segundo 44 en adelante)**
2. **Navegación a un punto**
3. **Patrulla área**
4. **Patrulla ruta**

REPOSITORIO DE TODOS LOS VIDEOS DE PRUEBAS

16. Bibliografía

Detección por voz con Google:

https://inteligencia-artificial.dev/reconocimiento-voz-python/

https://joserzapata.github.io/courses/mineria-audio/reconocimiento_voz/

Detección de voz con Vosk:

https://www.restack.io/p/vosk-speech-recognition-answer-python-example-cat-ai

https://github.com/shirosweets/vosk-speech-to-text/blob/main/src/my_vosk.py

Detección de QRs:

https://blog.aspose.com/es/barcode/python-qr-code-reader/

Detección de personas:

https://mediapipe.readthedocs.io/en/latest/

https://ai.google.dev/edge/mediapipe/solutions/vision/pose_landmarker?hl=es-419

https://omes-va.com/estimacion-postura-mediapipe-python/

https://www.tensorflow.org/hub/tutorials/movenet?hl=es-419

Detección de gestos:

https://mediapipe.readthedocs.io/en/latest/solutions/hands.html

Interfaz Terminal:

https://devjaime.medium.com/c%C3%B3mo-usar-la-biblioteca-rich-con-python-db09f1752c94

https://github.com/Textualize/rich

Explore lite:

https://wiki.ros.org/explore_lite

https://husarion.com/tutorials/ros-tutorials/10-exploration/

CustomTkinter:

https://customtkinter.tomschimansky.com/