

Curso Introductorio para la Categoría Robocup@Home Beginners

Instructores:

Marco Antonio Negrete Villanueva

Luis González Nava

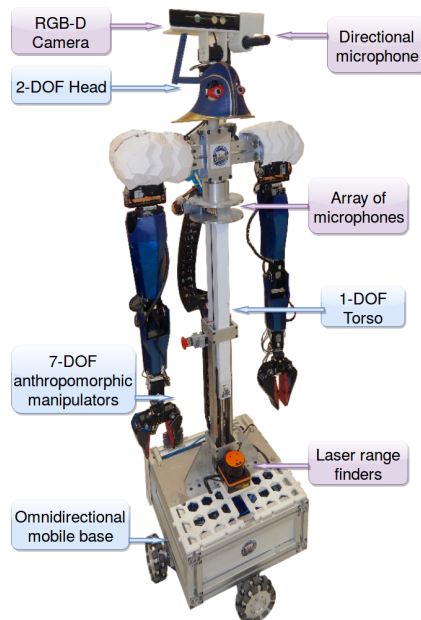
Facultad de Ingeniería, UNAM

Escuela de Invierno de Robótica 2020, Saltillo, México.

- Revisar el hardware necesario para tener un robot de servicio doméstico: sensores y actuadores necesarios.
- Dar un panorama general del software necesario para desarrollar un robot de servicio doméstico.
- Revisar las herramientas disponibles para cubrir las habilidades requeridas en la categoría @Home Beginners:
 - ▶ Navigation stack (planeación de movimientos)
 - ▶ Pocketsphinx (reconocimiento de voz)
 - ▶ Sound Play (para síntesis de voz)
 - ▶ OpenCV (reconocimiento de objetos y rostros)
 - ▶ MoveIt (para manipulación de objetos)

Su objetivo es el desarrollo de robots de servicio doméstico y está enfocada principalmente en las siguientes áreas:

- Interacción humano-robot
- Navegación en ambientes dinámicos
- Visión computacional y reconocimiento de objetos
- Manipulación de objetos
- Comportamientos adaptables
- Integración de Comportamientos
- Estandarización e integración de sistemas

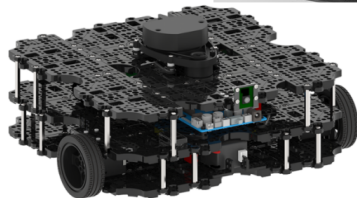
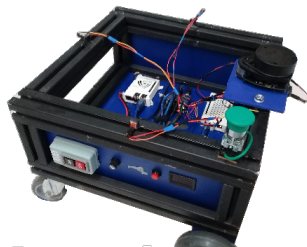




Esta competencia presenta un desafío introductorio a la categoría de @Home Major, basándose en una etapa de pruebas y una final.

- En la etapa de pruebas se evalúan funcionalidades básicas por separado:
 - ▶ Navegación
 - ▶ Reconocimiento de objetos
 - ▶ Manipulación
 - ▶ Reconocimiento de voz
- La prueba final es una integración de las habilidades anteriores.
- El robot debe ejecutar un comando del tipo “Bring [OBJETO] from [LUGAR]”.

- De preferencia, debe ser omnidireccional
- Turtle Bot (<https://www.turtlebot.com/>)
- Festo Robotino (<https://wiki.openrobotino.org/>)
- DIY: 3 ó 4 motores de corriente directa con ruedas omnidireccionales, 2 tarjetas Roboclaw, baterías de LiPo y chasis de aluminio estructural.





- Se pueden usar sólo cámaras RGB, pero es altamente recomendable tener información de profundidad.
- Kinect (<https://github.com/OpenKinect/libfreenect2>)
- Intel RealSense (<https://github.com/IntelRealSense/librealsense>)
- También se pueden usar cámaras estéreo, pero es mucho más sencillo usar cámaras con luz estructurada.

- Hokuyo (<https://www.hokuyo-aut.jp/>)
- RPLidar (<https://www.robotshop.com/en/slamtec.html>)
- SICK (<https://www.sick.com/ag/en/detection-and-ranging-solutions/2d-lidar-sensors/c/g91900>)
- El paquete http://wiki.ros.org/urg_node facilita su operación.
- Si no se tiene uno, se puede simular a partir de una cámara RGB-D con el paquete http://wiki.ros.org/pointcloud_to_laserscan.

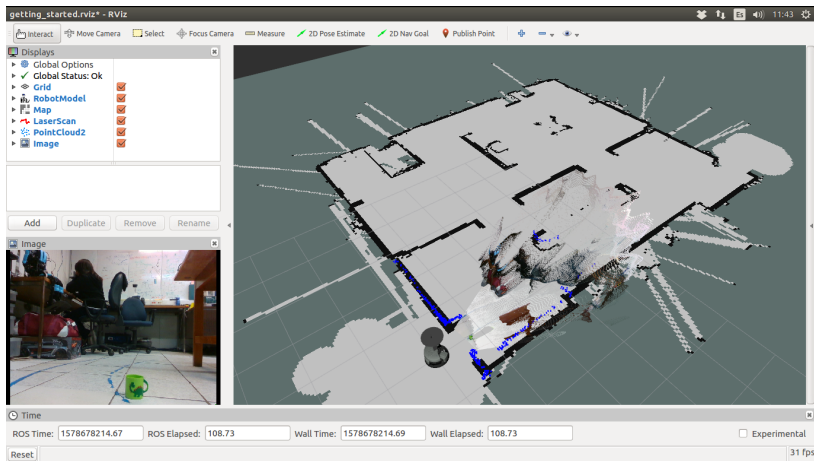




- Son recomendables por lo menos 5 DOF.
- Kuka LBR iiwa
(<http://wiki.ros.org/kuka>)
- Neuronics Katana
(<http://wiki.ros.org/katana>)
- DIY: Servomotores y Brackets Dynamixel
(<http://wiki.ros.org/dynamixel>)

- Se requiere de un marco de referencia absoluto, comúnmente llamado `map`. En Rviz, `map` se selecciona como referencia global.
- La base móvil debe publicar su odometría y aceptar comandos de movimiento.
 - ▶ Para la odometría, debe publicar la transformación de `odom` a `base_link`.
 - ▶ Para los comandos de movimiento, debe suscribirse al tópico `/cmd_vel` de tipo `geometry_msgs/Twist`.
- Se requiere de un nodo que publique la transformación de `odom` a `map`.
 - ▶ Si se está construyendo un mapa, esta transformación la publican paquetes como `gmapping` o `hector-mapping`.
 - ▶ Si ya se tiene un mapa, la transformación la publica el nodo de localización, generalmente `amcl`.
- Se requiere de un archivo que describa la cinemática del robot (archivo `urdf`), es decir, el árbol de transformaciones. Se recomienda que el *frame* raíz tenga el nombre `base_link`. Ejemplo: `catkin_ws/src/hardware/robot_description/robotino.urdf`
- Cada `joint` del robot corresponderá a una transformación publicada por el nodo `robot_state_publisher`.

Ejecutar el comando `roslaunch bring_up robotino_simul.launch`. Debe aparecer un rviz como el siguiente:

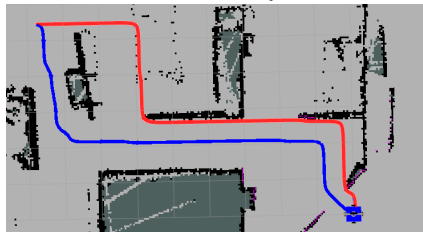


- ❶ Ejecutar el comando `roslaunch tf view_frames` y verificar en el archivo resultante (*frames.pdf*) las transformaciones y qué nodos las publican.
- ❷ Mediante el comando `rostopic info`, desplegar la información de los tópicos `/cmd_vel` , `/scan` y `/camera/depth_registered/points`.
- ❸ Detener la ejecución y modificar el archivo `catkin_ws/src/bring_up/launch/robotino_simul.launch` para cambiar lo siguiente:
 - ▶ La descripción del robot (`robotino.urdf` o `justina_simple.urdf`)
 - ▶ El mapa del ambiente (Universum, Biorobotica o TMR_2019)
- ❹ Modificar el archivo `catkin_ws/src/hardware/robot_description/robotino.urdf` y ver qué sucede cuando:
 - ▶ Se cambian los valores de la etiqueta `origin` en la línea 114.
 - ▶ Se elimina alguno de los campos `<joint>`.

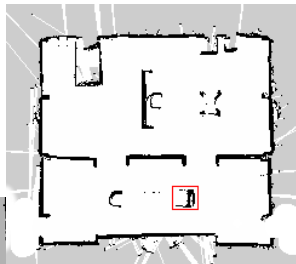
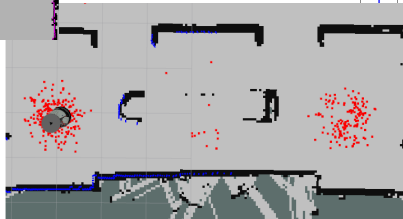
Planeación de rutas.

Encontrar un mapeo:

$$P(\alpha) : [0, 1] \rightarrow Q_{free}$$

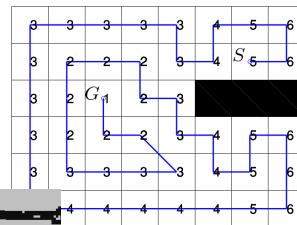


Localización: Determinar la configuración $q \in Q$ del robot.



Mapeo: Construir una representación del espacio:

$$Q = Q_{free} \cup Q_{occupied}$$

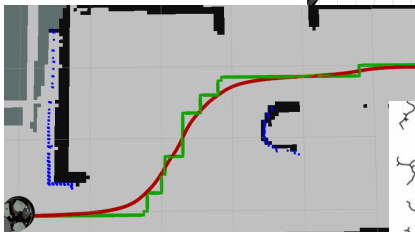


Cobertura: Mover al robot por todos los puntos $q \in Q_{free}$

Métodos variacionales

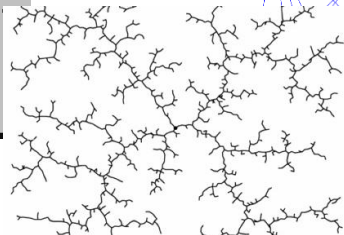
Ejemplo:

Suavizado de una ruta



Basados en muestreo

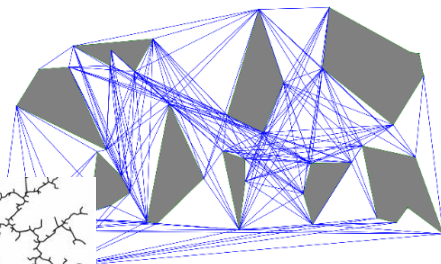
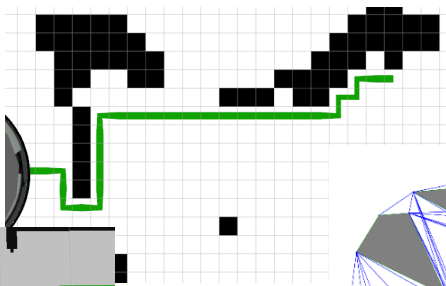
Ejemplo: Rapidly Exploring
Random Trees (RRT)



Búsqueda en grafos

Ejemplos:

Dijkstra y A*



Geométricos

Ejemplo:

Grafo de visibilidad

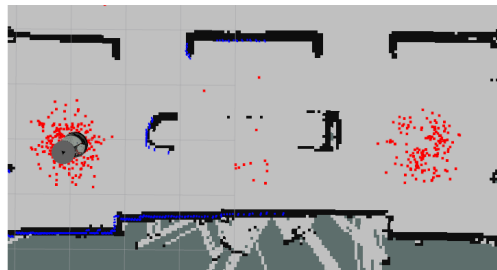
Filtro de Kalman:

- Con base en un modelo, filtra el ruido de las mediciones de posición.
- Supone que la posición tiene una distribución unimodal (normal).
- Converge sólo si la estimación inicial está cerca de la real.
- El número de estados crece con el número de marcas.
- Bajo costo computacional.



Filtro de Partículas:

- También requiere de un modelo de movimiento.
- La distribución de probabilidad de la posición es multimodal.
- Funciona para cualquier estimación inicial de la posición.
- Cada partícula mantiene una estimación
- Alto costo computacional.



Contiene varios paquetes para planeación de rutas, mapeo, localización y evasión de obstáculos (<http://wiki.ros.org/navigation>). Para este curso se usaron los siguientes:

- `map_server`: Lee el mapa de dos archivos, una imagen `.pgm` y un `yaml` con meta datos. Publica el mapa usando un mensaje de tipo `nav_msgs/OccupancyGrid`.
- `amcl`: Realiza la localización usando el mapa, la odometría y las lecturas del láser. Publica la transformación de `odom` a `map`. `move_base`: Realiza la mayor parte de las tareas de planeación de movimientos, para lo que usa los paquetes:
 - ▶ `dwa_local_planner`
 - ▶ `navfn`
 - ▶ `costmap_2d`

- ❶ Ejecutar los comandos
 - ▶ `roslaunch bring_up robotino_simul.launch`
 - ▶ `roslaunch bring_up navigation_move_base.launch`
- ❷ En el cuadro *Displays* de *Rviz* agregar los tópicos:
 - ▶ `/move_base/DWAPlanerROS/global_plan`
 - ▶ `/move_base/DWAPlanerROS/local_plan`
 - ▶ `/move_base/global_costmap/costmap`
- ❸ Fijar una meta con el botón *2D Nav Goal* y observar el comportamiento.

- ❶ Detener la ejecución de `navigation_move_base.launch`.
- ❷ En el archivo
`catkin_ws/src/config_files/move_base_params/costmap_common_params.yaml`:
 - ▶ Cambiar `cost_scaling_factor` a 1.0
 - ▶ Cambiar `inflation_radius` a 2.5
- ❸ Relanzar `navigation_move_base.launch` y observar qué sucede.
- ❹ Detener la ejecución de `navigation_move_base.launch`.
- ❺ En el archivo
`catkin_ws/src/config_files/move_base_params/dwa_local_planner_params.yaml`:
 - ▶ Cambiar `max_vel_x` a 2.0
 - ▶ Cambiar `max_trans_vel` a 2.0
 - ▶ Cambiar `acc_lim_x` a 2.0
- ❻ Relanzar `navigation_move_base.launch` y observar qué sucede.

Nota: En un robot real, los parámetros anteriores deben ser ligeramente menores a las capacidades físicas de la base móvil.

Pocketsphinx es un *toolkit* open source desarrollado por la Universidad de Carnegie Mellon (<https://cmusphinx.github.io/>).

- Aunque el toolbox original no está hecho específicamente para ROS, ya existen varios repositorios con nodos ya implementados que integran ROS y Pocketsphinx:
 - ▶ <https://github.com/mikeferguson/pocketsphinx>
 - ▶ <https://github.com/Pankaj-Baranwal/pocketsphinx>
- El usuario debe estar agregado al grupo *audio* para el correcto funcionamiento: `sudo usermod -a -G audio <user_name>`
- Se puede hacer reconocimiento usando una lista de palabras, un modelo de lenguaje o una gramática.
- Se utilizarán gramáticas y sus correspondientes diccionarios.
- Para construir diccionarios, visitar <https://cmusphinx.github.io/wiki/tutorialdict/>
- Para construir gramáticas, visitar <https://www.w3.org/TR/2000/NOTE-jsgf-20000605/>

- 1 Ejecutar el comando `roslaunch bring_up pocketsphinx_test.launch`
- 2 Verificar los volúmenes del micrófono
- 3 Revisar el archivo `catkin_ws/src/pocketsphinx/vocab/voice_cmd.gram` para ver las frases que se pueden reconocer de acuerdo con la gramática.
- 4 Probar el reconocimiento de voz.

- Es un paquete que permite reproducir archivos .wav o .ogg, sonidos predeterminados y síntesis de voz.
- La síntesis de voz se hace utilizando Festival (<http://www.cstr.ed.ac.uk/projects/festival/>).
- Para sintetizar voz, basta con correr el nodo `soundplay_node` y publicar un mensaje de tipo `sound_play/SoundRequest` con lo siguiente:
 - ▶ `msg_speech.sound = -3`
 - ▶ `msg_speech.command = 1`
 - ▶ `msg_speech.volume = 1.0`
 - ▶ `msg_speech.arg2 = "voz a utilizar"`
 - ▶ `msg_speech.arg = "texto a sintetizar"`

- 1 Ejecutar el comando

Dar algo de teoría

Cosas de ROS y ejercicios

Dar algo de teoría

Cosas de ROS y ejercicios

Dar opciones: FSM, clips, MDP.

Hacer programa para obedecer un comando de navegar a un lugar y buscar a una persona.