

# Introducción a los Robots de Servicio Doméstico

Instructor: Dr. Marco Antonio Negrete Villanueva

Facultad de Ingeniería, UNAM

Torneo Nacional de Robótica y Aeronáutica 2023  
Universidad Tecnológica de Tijuana  
<https://github.com/mnegretev/UTT-TNRYA-2023>

# Objetivos:

**Objetivo General:** Brindar los conocimientos básicos necesarios para desarrollar un robot de servicio doméstico.

## **Objetivos Específicos:**

- ▶ Revisar el hardware necesario para tener un robot de servicio doméstico: sensores y actuadores más comunes.
- ▶ Dar un panorama general del software necesario para desarrollar un robot de servicio doméstico.
- ▶ Revisar las herramientas disponibles para cubrir las habilidades básicas requeridas en un robot de servicio doméstico:
  - ▶ Navegación
  - ▶ Vision
  - ▶ Manipulación
  - ▶ Síntesis y reconocimiento de voz
  - ▶ Planeación de acciones

# Contenido

Introducción

ROS

Navegación

Visión

Manipulación

Síntesis de Voz

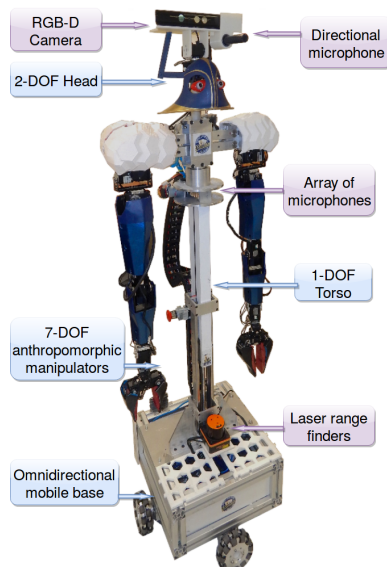
Reconocimiento de voz

Planeación de acciones

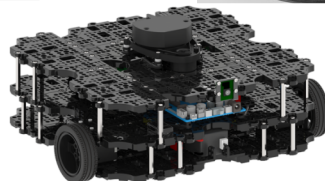
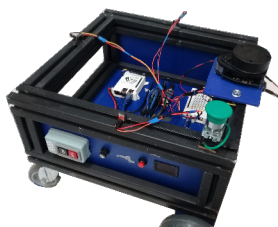
# Introducción

Son robots pensados para ayudar en tareas comunes del hogar u oficina. Requieren de varias habilidades:

- ▶ Interacción humano-robot
- ▶ Navegación en ambientes dinámicos
- ▶ Reconocimiento de objetos
- ▶ Manipulación de objetos
- ▶ Comportamientos adaptables
- ▶ Planeación de acciones



- ▶ 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](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](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>)

# La plataforma ROS



**ROS (Robot Operating System)** es un *middleware* de código abierto para el desarrollo de robots móviles.

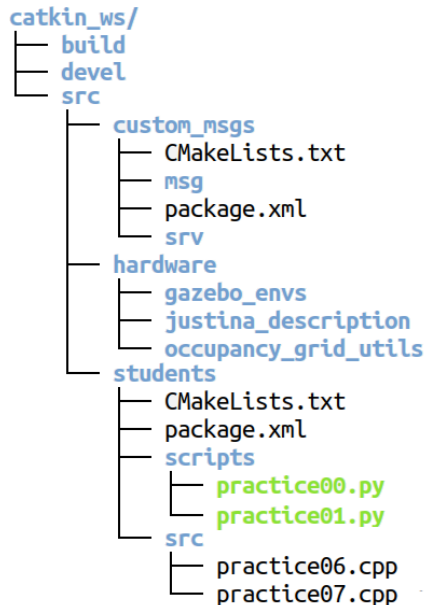
- ▶ Implementa funcionalidades comúnmente usadas en el desarrollo de robots como el paso de mensajes entre procesos y la administración de paquetes.
- ▶ Muchos drivers y algoritmos ya están implementados.
- ▶ Es una plataforma distribuida de procesos (llamados *nodos*).
- ▶ Facilita el reuso de código.
- ▶ Independiente del lenguaje (Python y C++ son los más usados).
- ▶ Facilita el escalamiento para proyectos de gran escala.

ROS se puede entender en dos grandes niveles conceptuales:

- ▶ **Sistema de archivos:** Recursos de ROS en disco
- ▶ **Grafo de procesos:** Una red *peer-to-peer* de procesos (llamados nodos) en tiempo de ejecución.

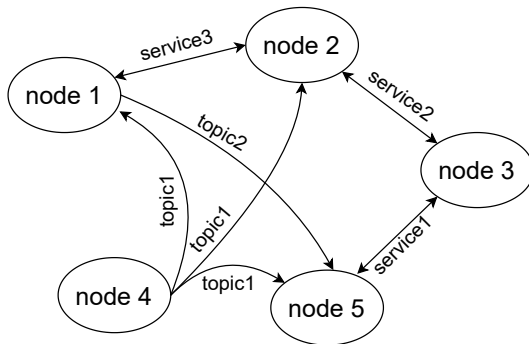
Recursos en disco:

- ▶ **Workspace:** carpeta que contiene los paquete desarrollados
- ▶ **Paquetes:** Principal unidad de organización del software en ROS (concepto heredado de Linux)
- ▶ **Manifiesto:** (`package.xml`) provee metadatos sobre el paquete (dependencias, banderas de compilación, información del desarrollador)
- ▶ **Mensajes (`msg`):** Archivos que definen la estructura de un *mensaje* en ROS.
- ▶ **Servicios (`srv`):** Archivos que definen las estructuras de la petición (*request*) y respuesta (*response*) de un servicio.



El grafo de procesos es una red *peer-to-peer* de programas (nodos) que intercambian información entre sí. Los principales componentes del este grafo son:

- ▶ master
- ▶ servidor de parámetros
- ▶ nodos
- ▶ mensajes
- ▶ servicios



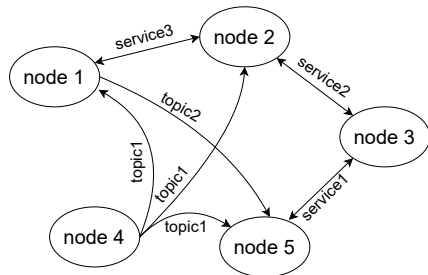
Los nodos (procesos) en ROS intercambian información a través de dos grandes patrones:

### ► Tópicos

- Son un patrón 1 :  $n$  de tipo *publicador/suscriptor*
- Son no bloqueantes
- Utilizan estructuras de datos definidas en archivos \*.msg para el envío de información

### ► Servicios

- Son un patrón 1 : 1 de tipo *petición/respuesta*
- Son bloqueantes
- Utilizan estructuras de datos definidas en archivos \*.srv para el intercambio de información.



Para mayor información:

- Tutoriales <http://wiki.ros.org/ROS/Tutorials>
- Koubâa, A. (Ed.). (2020). Robot Operating System (ROS): The Complete Reference. Springer Nature

# Navegación



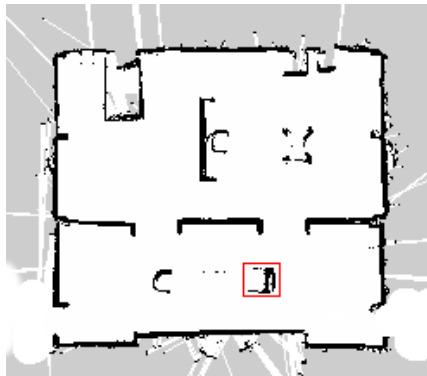
El problema de la planeación de movimientos comprende cuatro tareas básicas:

- ▶ Navegación (encontrar una ruta por el espacio libre de un punto inicial a uno final). Si la ruta está parametrizada con respecto al tiempo, se dice que es una trayectoria.
- ▶ Mapeo (construir una representación del ambiente a partir de las lecturas de los sensores y la configuración)
- ▶ Localización (determinar la configuración a partir de un mapa y de lecturas de los sensores)
- ▶ Barrido (pasar un actuador por todos los puntos de un subespacio)

Comúnmente el mapeo y la localización se realizan al mismo tiempo en el proceso conocido como SLAM (*Simultaneous Localization and Mapping*)

Es una discretización del espacio con una resolución determinada donde a cada celda se le asigna un número  $p \in [0, 1]$  que indica su nivel de ocupación.

- ▶ En un enfoque probabilístico,  $p$  indica la certeza de que la celda esté ocupada: 0, certeza de que está libre, 1, certeza de que está ocupada, 0.5, no se tiene información.
- ▶ En este curso, los niveles de ocupación solo serán 0 o 1. Para evitar el manejo de flotantes, el nivel de ocupación suele representarse con un entero en el intervalo  $[0,100]$  y un -1 si no hay información.



Ejecute el comando:

```
1   roslaunch bring_up path_planning.launch  
2
```

Inspeccione el mapa en el visualizador RViz. Luego abra el archivo:

`catkin_ws/src/config_files/maps/appartment.pgm` con cualquier editor de imágenes y modifíquelo.

Detenga la ejecución y vuelva a correr el comando anterior.

Dado un espacio de configuraciones  $Q$  con espacio libre  $Q_{free} \subset Q$  y espacio ocupado  $Q_{occ} \subset Q$ , la planeación de rutas consiste en contrar un mapeo:

$$f : [0, 1] \rightarrow Q_{free} \quad \text{con} \quad f(0) = q_s \quad f(1) = q_g$$

donde  $q_s$  y  $q_g$  son las configuraciones inicial y meta, respectivamente. Es decir, se debe encontrar una secuencia de puntos del espacio libre que permitan al robot moverse del punto inicial al punto meta sin chocar. Los métodos para planear rutas se pueden agrupar en:

- ▶ Basados en búsqueda en grafos ( $A^*$ , Dijkstra)
  - ▶ En un mapa de celdas de ocupación, cada celda libre es un nodo del grafo.
  - ▶ Cada nodo está conectado con las celdas vecinas del espacio libre.
- ▶ Basados en muestreo (RRT)
- ▶ Variacionales

---

**Data:** Mapa  $M$  de celdas de ocupación, configuración inicial  $q_{start}$ , configuración meta  $q_{goal}$

**Result:** Ruta  $P = [q_{start}, q_1, q_2, \dots, q_{goal}]$

Obtener los nodos  $n_s$  y  $n_g$  correspondientes a  $q_{start}$  y  $q_{goal}$

Lista abierta  $OL = \emptyset$  y lista cerrada  $CL = \emptyset$

**forall** Nodo  $n \in M$  **do**

$g(n) = \infty$        $f(n) = \infty$

**end**

Agregar  $n_s$  a  $OL$

$g(n_s) = 0$        $f(n_s) = 0$

Nodo actual  $n_c = n_s$

**while**  $OL \neq \emptyset$  y  $n_c \neq n_g$  **do**

    Seleccionar de  $OL$  el nodo  $n_c$  con el menor valor  $f$  y agregar  $n_c$  a  $CL$

**forall** Vecino  $n$  de  $n_c$  **do**

        Calcular los valores  $g$  y  $h$  para el nodo  $n$

**if**  $g < g(n)$  **then**

$g(n) = g$        $f(n) = g + h$        $Previo(n) = n_c$

**end**

**end**

    Agregar a  $OL$  los vecinos de  $n_c$  que no estén ya en  $OL$  ni en  $CL$

**end**

**if**  $n_c \neq n_g$  **then**

    Anunciar Falla

**end**

Obtener la configuración  $q_i$  para cada nodo  $n_i$  de la ruta

---

El valor  $g$  es una función de costo y el algoritmo A\* siempre devolverá la ruta que minimice el costo total del nodo inicial al nodo meta. Las funciones más comunes son:

- ▶ Distancia de Manhattan:  $d(p_1, p_2) = |p_{1_x} - p_{2_x}| + |p_{1_y} - p_{2_y}|$
- ▶ Distancia Euclidiana:  $d(p_1, p_2) = ((p_{1_x} - p_{2_x})^2 + (p_{1_y} - p_{2_y})^2)^{1/2}$

En la función  $g$  se puede incluir cualquier criterio para planear una ruta: la más corta, la más rápida, la más segura, etc.

La heurística  $h$  sirve para expandir menos nodos y es una función que debe *subestimar* el costo real de llegar de un nodo  $n$  al nodo meta  $n_g$ . La distancia Euclidiana y la distancia de Manhattan son dos funciones también muy usadas como heurísticas.

Ejecute el comando:

```
1 roslaunch bring_up path_planning.launch  
2
```

En otra terminal, ejecute el comando:

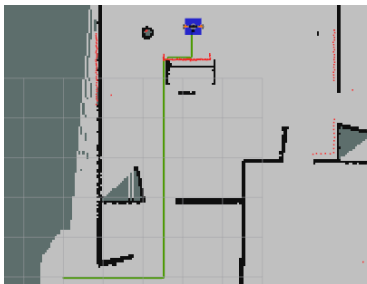
```
1 rosrun exercises a_star.py  
2
```

Utilizando la GUI calcule una ruta desde la posición del robot hasta el punto (0,0). En el visualizador RViz se mostrará la ruta calculada.

Start Pose:

Goal Pose:

Inflation:



Detenga la ejecución del programa `a_star.py`.

Modifique el archivo `catkin_ws/src/exercises/scripts/a_star.py` para utilizar distancia Euclidiana en lugar de distancia de Manhattan. Modifique las secciones marcadas con el comentario `#Ejercicio`.

```
26 # EJERCICIO:  
27 # Modifique la lista de nodos adyacentes para usar conectividad ocho  
28 # en lugar de conectividad 4  
29 adjacents = [[1,0],[0,1],[-1,0],[0,-1]]  
30 #adjacents= [[1,0],[0,1],[-1,0],[0,-1],[1,1],[-1,1],[-1,-1],[1,-1]]
```

```
49 # EJERCICIO:  
50 # Modifique el calculo del valor 'g' y la heuristica 'h' para  
51 # utilizar distancia euclidiana en lugar de distancia de Manhattan.  
52 g = g_values[row, col] + abs(row-r) + abs(col-c)  
53 h = abs(goal_r - r) + abs(goal_c - c)  
54 # g = g_values[row, col] + math.sqrt((row-r)**2 + (col - c)**2)  
55 # h = math.sqrt((goal_r-r)**2 + (goal_c - c)**2)
```

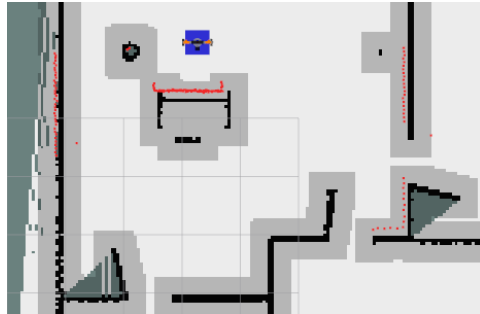
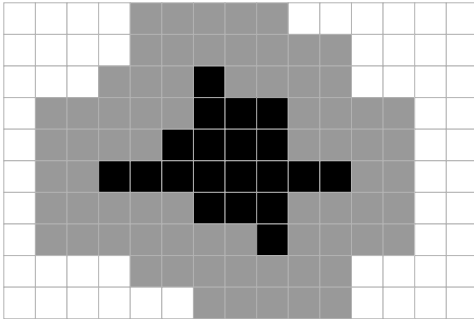
Ejecute nuevamente el comando

```
1      rosrun exercises a_star.py  
2
```

Y compare los resultados.



Para evitar que se calculen rutas por espacios por donde el robot en realidad no puede pasar debido a sus dimensiones, los obstáculos del mapa se deben *inflar* cuando menos un número de celdas igual al radio del robot.



Modifique el archivo `catkin_ws/src/exercises/scripts/inflation.py` para implementar el algoritmo de inflado de obstáculos:

```
19 # EJERCICIO:
20 for i in range(height):
21     for j in range(width):
22         if static_map[i,j] > 50:
23             for k1 in range(-inflation_cells, inflation_cells+1):
24                 for k2 in range(-inflation_cells, inflation_cells+1):
25                     inflated[i+k1, j+k2] = 100
```

Ejecute el comando:

```
1 roslaunch bring_up path_planning.launch
2
```

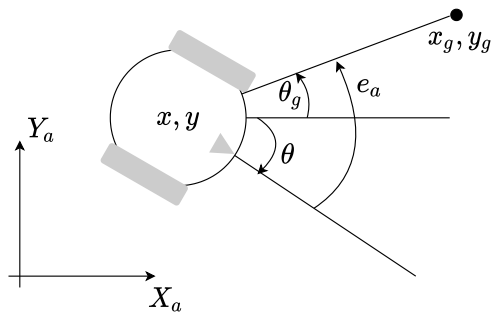
En otra terminal, ejecute el comando:

```
1 rosrun exercises a_star.py
2
```

Y en una tercera terminal ejecute el comando

```
1 rosrun exercises inflation.py
2
```

Suponiendo que el robot está en la posición y orientación  $(x, y, \theta)$  y que se quiere alcanzar la posición deseada  $(x_g, y_g)$ , como se muestra en la figura. Las leyes de control

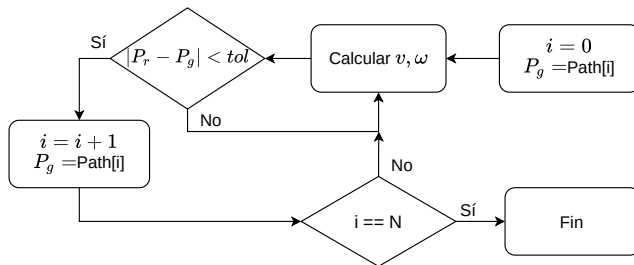


$$v = v_{max} e^{-e_a^2/\alpha}$$

$$\omega = \omega_{max} \left( \frac{2}{1 + e^{-e_a/\beta}} - 1 \right)$$

con  $e_a = \text{atan2}(y_g - y, x_g - x) - \theta$ , permiten que el robot alcance la posición deseada. Los parámetros  $v_{max}$  y  $\omega_{max}$  se deben seleccionar de acuerdo con las capacidades del robot. Las constantes  $\alpha$  y  $\beta$  permiten controlar qué tan rápido cambian la velocidad lineal y angular del robot.

El control de posición se puede utilizar para seguir una ruta si se fija como punto meta cada punto de la ruta, hasta alcanzar el último.



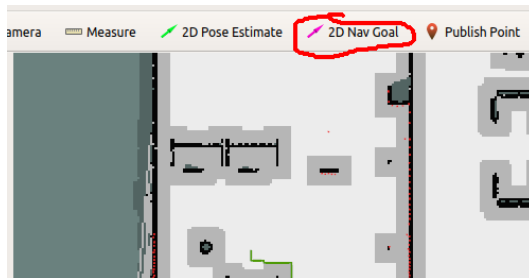
Ejecute el comando:

```
1  roslaunch bring_up path_planning.launch
2
```

En otra terminal, ejecute el comando:

```
1  roslaunch bring_up exercises_path_planning.launch
2
```

Utilizando RViz, envíe el robot a un punto meta:



Detenga la ejecución de los ejercicios. Abra el archivo `catkin_ws/src/exercises/scripts/control.py` y modifique las constantes  $v_{max}$ ,  $\omega_{max}$ ,  $\alpha$  y  $\beta$ .

```
23 # EJERCICIO
24 # Modifique las constantes v_max, w_max, alpha y beta y observe
25 # el cambio en el comportamiento del robot.
26 v_max = 0.4
27 w_max = 0.5
28 alpha = 1.0
29 beta = 0.1
30 [error_x, error_y] = [goal_x - robot_x, goal_y - robot_y]
31 error_a = (math.atan2(error_y, error_x) - robot_a + math.pi)%(2*math.
    pi) - math.pi
32 cmd_vel.linear.x = v_max*math.exp(-error_a*error_a/alpha)
33 cmd_vel.angular.z = w_max*(2/(1 + math.exp(-error_a/beta)) - 1)
```

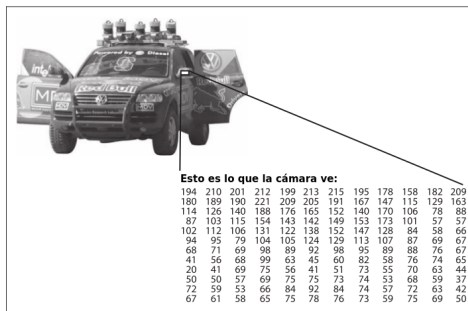
Vuelva a ejecutar el comando

```
1 roslaunch bring_up exercises_path_planning.launch
2
```

Y compare el comportamiento del robot.

# Visión

Una imagen (en escala de grises) es una función  $I(x, y)$  donde  $x, y$  son variables discretas en coordenadas de imagen y la función  $I$  es intensidad luminosa. Las imágenes también pueden considerarse como arreglos bidimensionales de números entre un mínimo y un máximo (usualmente 0-255).



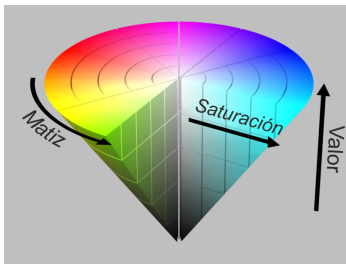
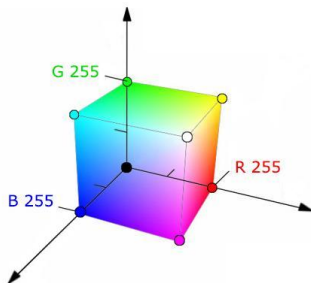
Aunque formalmente una imagen es un mapeo  $f : \mathbb{R}^2 \rightarrow \mathbb{R}$ , en la práctica, tanto  $x, y$  como  $I$  son variables discretas con valores entre un mínimo y un máximo.



Las imágenes de color son funciones vectoriales  $f : \mathbb{R}^2 \rightarrow \mathbb{R}^3$  donde cada componente de la función se llama canal:

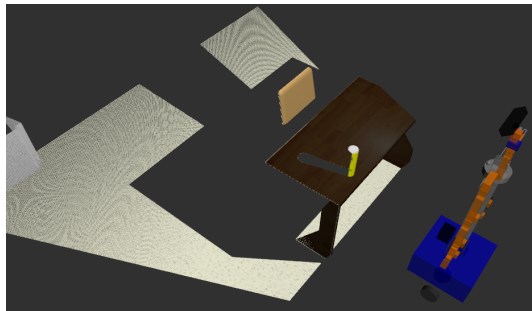
$$I(x, y) = \begin{bmatrix} r(x, y) \\ g(x, y) \\ b(x, y) \end{bmatrix}$$

Los espacios de color son diferentes formas de representar el color mediante la combinación de un conjunto de valores.



En segmentación por color se recomienda más usar HSV, pues es más robusto ante cambios en la iluminación.

Las nubes de puntos son conjuntos de vectores que representan puntos en el espacio. Estos vectores generalmente tienen información de posición  $(x, y, z)$ . También pueden contener información de color  $(x, y, z, r, g, b)$ .



Son útiles para determinar la posición en el espacio de los objetos reconocidos.

- ▶ OpenCV es un conjunto de bibliotecas que facilita la implementación de algoritmos de visión computacional.
- ▶ Se puede usar con diversos lenguajes: C++, Python, Java.
- ▶ En Python utiliza la biblioteca Numpy.
- ▶ Las imágenes se representan como matrices donde cada elemento puede ser un solo valor, o bien tres valores, dependiendo de si la imagen está en escala de grises o a color.
- ▶ La configuración más común es que cada pixel esté representado por tres bytes.
- ▶ Las nubes de puntos se representan también como matrices donde cada elemento es una terna de flotantes con la posición  $(x, y, z)$ .

La segmentación de una imagen se refiere a obtener regiones significativas con ciertas características. En este caso, la característica es que estén en un cierto intervalo de color. Los pasos generales para esto son:

1. Transformación de la imagen del espacio BGR al HSV (función `cvtColor`)
2. Obtención de aquellos pixeles que están en un rango de color (función `inRange`)
3. Eliminación de *outliers*, generalmente con operadores morfológicos (funciones `erode` y `dilate`)
4. Obtención del centroide de la región (funciones `findNonZero` y `mean`)
5. Si se dispone de una nube de puntos, se puede obtener la posición  $(x, y, z)$  del centroide de la región segmentada.

Modifique el archivo `catkin_ws/src/exercises/scripts/color_segmentation.py` para fijar los límites de color, superior e inferior, dependiendo del objeto que se desea segmentar.

```
17 def segment_by_color(img_bgr, points, obj_name):  
18     # Para 'pringles': [25, 50, 50] - [35, 255, 255]  
19     # Para 'drink':    [10,200, 50] - [20, 255, 255]  
20     lower_limit = numpy.array([25, 50, 50])  
21     upper_limit = numpy.array([35,255,255])
```

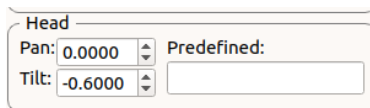
Ejecute el comando:

```
1   roslaunch bring_up path_planning.launch  
2
```

Y en otra terminal ejecute el comando

```
1   rosrun exercises color_segmentation.py  
2
```

Utilizando la GUI, baje la cabeza del robot hasta que los objetos del escritorio estén en el campo visual:



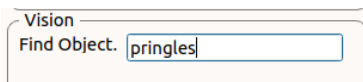
Head

Pan: 0.0000

Tilt: -0.6000

Predefined:

Utilizando la GUI, ejecute la segmentación por color. Los objetos que se pueden segmentar son “pringles” o “drink”.



Vision

Find Object. pringles

# Manipulación

Un cuerpo rígido en el espacio puede tener una posición  $(x, y, z)$  y una orientación. La orientación se puede representar de varias formas:

- ▶ Mediante ángulos de Euler: roll, pitch y yaw  $RPY = (\psi, \theta, \phi)$
- ▶ Mediante cuaterniones
- ▶ Mediante una matriz de rotación  $R \in SO(3)$

Los ángulos  $RPY$  son rotaciones intrínsecas sobre los ejes  $X$ ,  $Y$ , y  $Z$  respectivamente. Se llaman intrínsecas porque son rotaciones que ocurren sobre un sistema de referencia *atado* a un cuerpo rígido.

Cualquier orientación se puede obtener mediante la composición de tres rotaciones básicas:

$$R = R_{z,\phi} R_{y,\theta} R_{x,\psi}$$

Es decir, primero una rotación de  $\phi$  radianes sobre el eje  $Z$ , seguida de una rotación de  $\theta$  radianes sobre el eje  $Y$  del sistema resultante y una rotación de  $\psi$  radianes sobre el eje  $X$  del sistema rotado.



Una Transformación Homogénea es una matriz de la forma

$$T = \begin{bmatrix} & R \in SO(3) & \begin{matrix} d_x \\ d_y \\ d_z \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Puede servir para

- ▶ Representar la posición y orientación de un cuerpo rígido
- ▶ Representar una transformación de coordenadas  $T_{ab}$  de un sistema de referencia  $b$  a un sistema  $a$

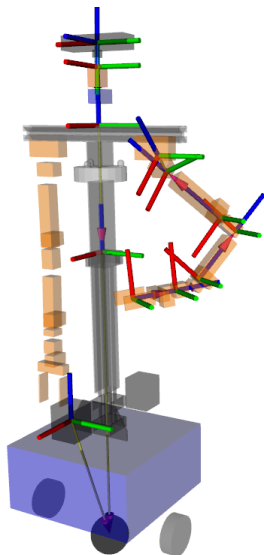
Propiedades:

- ▶ Asociatividad:  $(T_1 T_2) T_3 = T_1 (T_2 T_3)$
- ▶ Inversa:

$$T^{-1} = \begin{bmatrix} R^T & -R^T d \\ 0 & 1 \end{bmatrix}$$

- ▶ Cancelación de índices:  $T_{ab} = T_{ac} T_{cb}$

Es útil tener una descripción de la forma en que están conectadas las diferentes articulaciones del robot. Se considera que sobre cada articulación hay un sistema de referencia (*frame*) que está trasladado y rotado con respecto al sistema anterior.



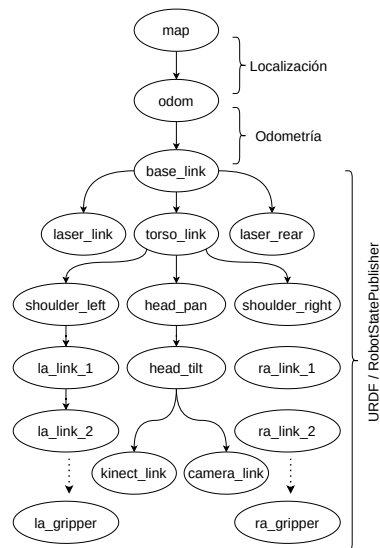
El sistema *absoluto* se suele llamar *map*

El sistema base del robot se suele llamar *base\_link*

Las transformaciones de *map* a *base\_link* las determina el sistema de localización

El resto de las transformaciones se determinan con la posición de cada articulación

El árbol cinemático se traduce en una cadena de multiplicaciones de Transformaciones Homogéneas.



El formato URDF permite describir el arbol cinemático del robot mediante etiquetas XML:

```
1 <robot>
2   <!-- Define la forma y tamaño de la base móvil-->
3   <link name="base_link">
4     <visual>
5       <origin xyz="0 0 0.235" rpy="0 0 0"/><material name="blue"/>
6       <geometry> <box size="0.42 0.42 0.20"/></geometry>
7     </visual>
8   </link>
9   <!-- Define la forma y tamaño del sensor laser-->
10  <link name="laser_link">
11    <visual>
12      <origin xyz="0 0 0" rpy="0 0 0"/><material name="black"/>
13      <geometry> <box size="0.08 0.08 0.1"/></geometry>
14    </visual>
15  </link>
16  <!-- Define la posición del laser con respecto a la base móvil-->
17  <joint name="laser_connect" type="fixed">
18    <origin xyz="0.17 0 0.44" rpy="0 0 0"/>
19    <parent link="base_link"/><child link="laser_link"/>
20  </joint>
21 </robot>
```

Cada etiqueta <joint> representará una Transformación Homogénea.

El formato Xacro es un lenguaje de *macros* que permite obtener archivos XML más cortos. Es útil para especificar parámetros físicos en el URDF como inercias y volúmenes:

```
1 <robot name="justina" xmlns:xacro="http://www.ros.org/wiki/xacro">
2   <xacro:property name="width" value="0.42"/>
3   <xacro:property name="depth" value="0.42"/>
4   <xacro:property name="height" value="0.2"/>
5   <xacro:property name="mass" value="30.0"/>
6
7   <link name="base_link">
8     <visual>
9       <origin xyz="0 0 0.235" rpy="0 0 0"/><material name="blue"/>
10      <geometry> <box size="{width} {depth} {height}"/></geometry>
11    </visual>
12    <collision>
13      <origin xyz="0 0 0.235" rpy="0 0 0"/>
14      <geometry> <box size="{width} {depth} {height}"/></geometry>
15    </collision>
16    <inertial>
17      <origin xyz="0 0 0.235" rpy="0 0 0"/><mass value="50.00"/>
18      <xacro:box_inertia m="{mass}" x="{depth}" y="{width}" z="{
19        height}"/>
19    </inertial>
20  </link>
21 </robot>
```

Abra el archivo

catkin\_ws/src/hardware/justina\_description/urdf/justina\_base.xacro y vaya a la línea 228:

```
227 <joint name="laser_connect" type="fixed">
228   <origin xyz="0.17 0 0.44" rpy="0 0 0" />
229   <parent link="base_link" />
230   <child link="laser_link" />
231 </joint>
```

Modifique el atributo xyz y aumente 1 m en la coordenada en z. Después ejecute el comando:

```
1   roslaunch bring-up path_planning.launch
2
```

Detenga la simulación. Ahora modifique el atributo rpy, cambie los valores a "1.5708 0 0" y ejecute de nuevo la simulación.

La cinemática directa consiste en determinar la posición y orientación del efector final del manipulador a partir de la posición de cada articulación. Esta se puede calcular con la ecuación:

$$P_1 = T_{12} T_{23} T_{34} T_{45} T_{56} T_{67} T_{7g} P_g$$

donde  $P_g = [0, 0, 0, 1]^T$  es la posición del gripper con respecto al sistema del gripper,  $P_1$  es la posición del gripper con respecto al sistema base y  $T_{ab}$  es la transformación homogénea que define la rotación y traslación producida por cada articulación. Las matrices  $T_{ab}$  tienen la forma:

$$T_{ab} = \begin{bmatrix} & R_{ab} \in SO(3) & \begin{matrix} dx_{ab} \\ dy_{ab} \\ dz_{ab} \end{matrix} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Donde  $R_{ab}$  representa la rotación del sistema  $b$  respecto al sistema  $a$  y  $(dx_{ab}, dy_{ab}, dz_{ab})$  es la traslación del sistema  $b$  respecto al sistema  $a$ .

La rotación  $R_{ab}$  está definida en el URDF por el atributo “rpy” de la sub etiqueta origin de la etiqueta joint y por la posición de la articulación. La traslación  $(dx_{ab}, dy_{ab}, dz_{ab})$  está definida por el atributo “xyz”.

La cinemática inversa consiste en determinar las posiciones que debe tener cada articulación para que el efector final tenga la posición y orientación deseadas.

- ▶ Mientras la cinemática directa siempre tiene solución, la cinemática inversa, no.
- ▶ Se puede resolver por métodos geométricos para obtener una solución cerrada, aunque el análisis puede ser muy complicado.
- ▶ Una solución más general se puede obtener mediante un método numérico.

Suponiendo que se tiene una configuración deseada  $p_d \in \mathbb{R}^6$  ( $xyz - RPY$ ), se desea encontrar el conjunto de posiciones articulares  $q \in \mathbb{R}^7$  que satisfagan la ecuación

$$FK(q) - p_d = 0$$

donde la función  $FK$  representa la cinemática directa.

El método numérico de Newton-Raphson sirve para encontrar raíces, es decir, para resolver ecuaciones de la forma

$$f(x) = 0$$

El algoritmo es el siguiente:

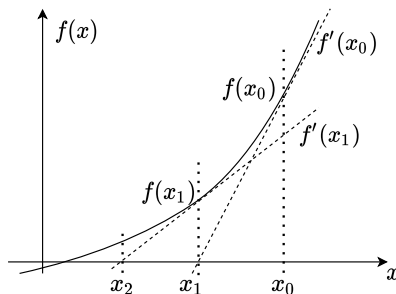
---

---

```
 $x_i \leftarrow x_0$   
while  $|f(x)| > \epsilon$  do  
   $x_{i+1} \leftarrow x_i - \frac{f(x_i)}{f'(x_i)}$   
end
```

---

---





El Jacobiano es una matriz que relaciona la velocidad articular  $\dot{q}$  con la velocidad en el espacio cartesiano  $[v \ \omega]^T$  (velocidad lineal y angular):

$$\dot{p} = \begin{bmatrix} v \\ \omega \end{bmatrix} = J\dot{q} \quad p = [x, y, z, roll, pitch, yaw] \in \mathbb{R}^6, \quad J \in \mathbb{R}^{6 \times 7}, \quad q \in \mathbb{R}^7$$

$$J = \begin{bmatrix} \frac{\partial p_1}{\partial q_1} & \dots & \frac{\partial p_1}{\partial q_7} \\ \vdots & \ddots & \vdots \\ \frac{\partial p_6}{\partial q_1} & \dots & \frac{\partial p_6}{\partial q_7} \end{bmatrix}$$

- La matriz  $J$  se puede obtener analíticamente, sin embargo, dado el número de grados de libertad, resulta muy complicado
- Se puede obtener aproximando las derivadas parciales con diferencias finitas:

$$J = \begin{bmatrix} \frac{FK(q_+^1) - FK(q_-^1)}{2\Delta q} & \dots & \frac{FK(q_+^7) - FK(q_-^7)}{2\Delta q} \end{bmatrix}$$

$$q_+^i = [q_1 \quad \dots \quad q_i + \Delta q \quad \dots \quad q_7]$$

$$q_-^i = [q_1 \quad \dots \quad q_i - \Delta q \quad \dots \quad q_7]$$

con  $\Delta q$ , un valor lo suficientemente pequeño para una buena aproximación de la derivada.

Aplicando Newton-Raphson para resolver la ecuación:

$$FK(q) - p_d = 0$$

Se tiene:

---

---

```
 $q_i \leftarrow q_0$  //Una estimación inicial que puede ser la posición articular actual  
 $p \leftarrow FK(q_i)$  //La posición cartesiana que tendría el gripper con la estimación inicial  
while  $|p - p_d| > \epsilon$  do  
     $J \leftarrow \text{Jacobiano}(q)$   
     $q_{i+1} \leftarrow q_i - J^\dagger(p - p_d)$   
     $p \leftarrow FK(q_i)$   
end
```

---

Puesto que el Jacobiano  $J$  no es una matriz cuadrada, no tiene inversa, por lo que se utiliza la matriz pseudoinversa  $J^\dagger$ .

- ▶ Es importante que las variables angulares siempre estén en el intervalo  $(-\pi, \pi]$ :
  - ▶ Las posiciones articulares
  - ▶ Los ángulos roll, pitch, yaw
  - ▶ Las componentes angulares del error  $p - p_d$

Abra el archivo `catkin_ws/src/exercises/scripts/inverse_kinematics.py` e inspeccione los segmentos de código marcados con el comentario `#Ejercicio` para comparar con los algoritmos para el cálculo de las cinemáticas directa e inversa y del Jacobiano. En una terminal ejecute el comando:

```
1   roslaunch bring_up manipulation.launch
2
```

Y en otra ejecute el comando:

```
1   rosrun exercises inverse_kinematics.py
2
```

Utilizando la GUI, calcule la cinemática inversa para que el brazo izquierdo alcance la posición  $(x, y, z, roll, pitch, yaw) = (0.2, 0.1, -0.3, 3.0, -1.57, -3.0)$  y el brazo derecho, la posición  $(x, y, z, roll, pitch, yaw) = (0.2, -0.1, -0.3, 3.0, -1.57, -3.0)$ :

Th 7:	0.0000	Th 7:	0.0000
G:	0.0000	G:	0.0000
Left arm cartesian:		Right arm cartesian:	
0.2 0.1 -0.3 3 -1.57 -3		0.2 -0.1 -0.3 3 -1.57 -3	

El control Proporcional-Integral-Derivativo es un tipo de control lineal en lazo cerrado que calcula la acción de control mediante una combinación lineal del error, la integral del error y la derivada del error.

- ▶ Para el manipulador, el ángulo deseado  $q_d$  está dado por el resultado de la cinemática inversa.
- ▶ La posición angular  $q$  se obtiene de los motores o del simulador.
- ▶ La salida del controlador es el torque  $\tau$  que se envía a los motores.

En la versión continua:

$$\tau(t) = K_p e(t) + K_I \int e(t) dt + K_d \dot{e}(t)$$

con  $e = q_d - q$  En la versión discreta:

$$\tau_i = K_p e_i + K_I \sum_{j=0}^i e_j + K_d \frac{e_i - e_{i-1}}{\Delta t}$$

con  $\Delta t$ , el periodo de muestreo.

Aunque la interacción de las tres señales (error, integral del error y derivada del error) es compleja y depende mucho del sistema, de manera intuitiva se pueden indicar las siguientes funciones de cada componente:

- ▶ **Proporcional:** Aumenta o disminuye el tiempo de asentamiento.
- ▶ **Integral:** Reduce el error en estado estable, aunque puede producir inestabilidad.
- ▶ **Derivativa:** Funciona como amortiguamiento y ayuda a disminuir el sobrepaso.

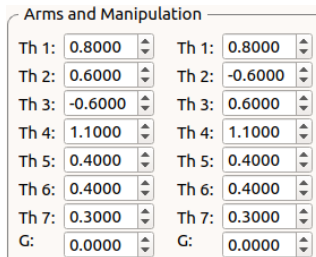
Son un conjunto de paquetes que implementan controladores PID y varias interfaces de hardware.

- ▶ El `stack ros_control` implementa varias interfaces de hardware. En este curso, la interfaz usada es la que interactúa con la simulación de Gazebo. De este `stack`, el paquete `controller_manager` es importante porque utiliza un archivo *yaml* para lanzar otros nodos que implementan controladores PID.
- ▶ El `stack ros_controllers` implementa varios algoritmos de control para diferentes tipos de actuadores.

Ejecute el comando:

```
1   roslaunch bring_up manipulation.launch  
2
```

Utilizando la GUI, mueva las articulaciones de los manipuladores a alguna posición diferente de cero y observe el comportamiento.



The image shows a GUI titled "Arms and Manipulation" with two columns of joint position controls. Each control consists of a label (Th 1 through Th 7, and G) followed by a numerical input field and a vertical slider. The values in the input fields are as follows:

Manipulator	Th 1	Th 2	Th 3	Th 4	Th 5	Th 6	Th 7	G
Left	0.8000	0.6000	-0.6000	1.1000	0.4000	0.4000	0.3000	0.0000
Right	0.8000	-0.6000	0.6000	1.1000	0.4000	0.4000	0.3000	0.0000

Abra el archivo `src/config_files/ros_control/justina_controllers.yaml` y modifique las ganancias de los diferentes controladores. Detenga la simulación y ejecútela de nuevo. Compare el comportamiento.

# Síntesis 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"`

Ejecute el comando:

```
1    roslaunch bring_up speech_synthesis.launch
2
```

En otra terminal, ejecute el comando:

```
1    rosrun exercises speech_synthesis.py "my first synthetized voice"
2
```

Para instalar más voces:

- ▶ Ejecute el comando `sudo apt-get install festvox-<voz deseada>`
- ▶ Para ver qué voces se tienen instaladas: `ls /usr/share/festival/voices/english/`

Modifique el archivo `catkin_ws/src/exercises/scripts/speech_synthesis.py` y cambie la voz a utilizar en el mensaje `SoundRequest`.

```
17 msg_speech = SoundRequest()
18 msg_speech.sound = -3
19 msg_speech.command = 1
20 msg_speech.volume = 1.0
21 #
22 # EJERCICIO
23 # Cambie la voz por alguna de las voces instaladas
24 #
25 msg_speech.arg2 = "voice_kal_diphone"
26 msg_speech.arg = text_to_say
```

El nombre de la voz se compone de `voice_` más el nombre que aparece en la carpeta `/usr/share/festival/voices/english/`.

# Reconocimiento de voz

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. Verifique el volumen del micrófono
2. Inspeccione el archivo `catkin_ws/src/pocketsphinx/vocab/gpsr.gram` para ver las frases que se pueden reconocer de acuerdo con la gramática.
3. Ejecute el comando `roslaunch bring_up speech_recognition.launch`
4. En otra terminal, ejecute el comando `rostopic echo /recognized`
5. Pruebe el reconocimiento de voz con alguna de las siguientes frases:
  - 5.1 Robot, take the pringles to the table
  - 5.2 Robot, take the drink to the table
  - 5.3 Robot, take the pringles to the kitchen
  - 5.4 Robot, take the drink to the kitchen

# Planeación de acciones

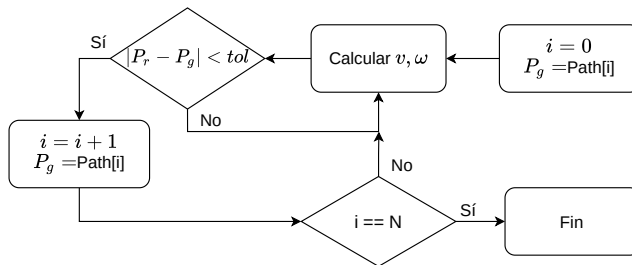
Las FSM son modelos que permiten representar procesos discretos donde el sistema puede estar en un estado bien definido y existen un conjunto de reglas para definir el estado siguiente en función del estado actual y las entradas. Opcionalmente, se pueden definir salidas para cada estado.

Formalmente, una FSM está definida por:

- ▶  $S$  : Conjunto finito no vacío de estados
- ▶  $\Sigma$  : Conjunto finito no vacío de entradas
- ▶  $s_0 \in S$  : Estado inicial
- ▶  $\delta : S \times \Sigma \rightarrow S$  : Función de transición de estados
- ▶  $F$  : Conjunto de estados finales (puede ser un conjunto vacío)

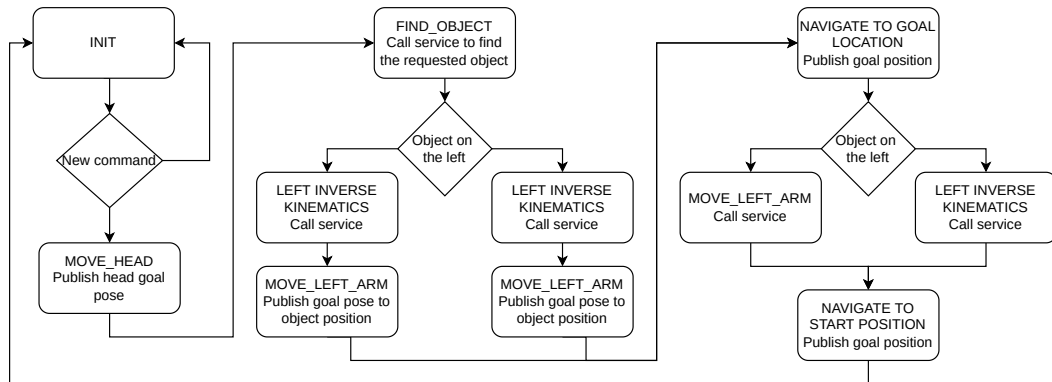


Una forma de representar los estados y transiciones de una FSM es mediante una carta ASM.  
Ejemplo:



Las máquinas de estados se pueden utilizar para ejecutar tareas “sencillas” en un robot de servicio doméstico.

En cada estado podemos enviar comandos de movimiento y utilizar resultados de percepción para definir el siguiente estado:



Abra el archivo `catkin_ws/src/exercises/scripts/et_facta_est_lux.py` y agregue comandos de voz en los estados relevantes.

```
155 # EJERCICIO FINAL
156 # Agregue funciones 'say' en los estados de modo que el robot
157 # indique por voz la parte de la tarea que se esta ejecutando
158 while not rospy.is_shutdown():
159     if current_state == "SM_INIT":
160         print("Waiting for new task")
161         current_state = "SM_WAITING_NEW_TASK"
162     elif current_state == "SM_WAITING_NEW_TASK":
163         if new_task:
164             req_object, req_loc = parse_command(recognized_speech)
165             say("Executing the command, " + recognized_speech)
166             current_state = "SM_MOVE_HEAD"
167             new_task = False
168             executing_task = True
```

En una terminal, ejecute el comando:

```
1   roslaunch bring_up fiat_lux.launch
2
```

Y en otra terminal, corra el ejercicio final:

```
1   rosrn  exercises  et_facta_est_lux.py
2
```

# Gracias

## Contacto

Dr. Marco Negrete  
Profesor Asociado C  
Departamento de Procesamiento de Señales  
Facultad de Ingeniería, UNAM.

[mnegretev.info](mailto:mnegretev.info)  
[marco.negrete@ingenieria.unam.edu](mailto:marco.negrete@ingenieria.unam.edu)