

Tareas Básicas en Robots de Servicio Doméstico

Instructor: Dr. Marco Antonio Negrete Villanueva

Facultad de Ingeniería, UNAM

Escuela de Invierno de Robótica 2023
Zacatecas

<https://github.com/mnegretev/EIR-2023-AtHomeBasicTasks>

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

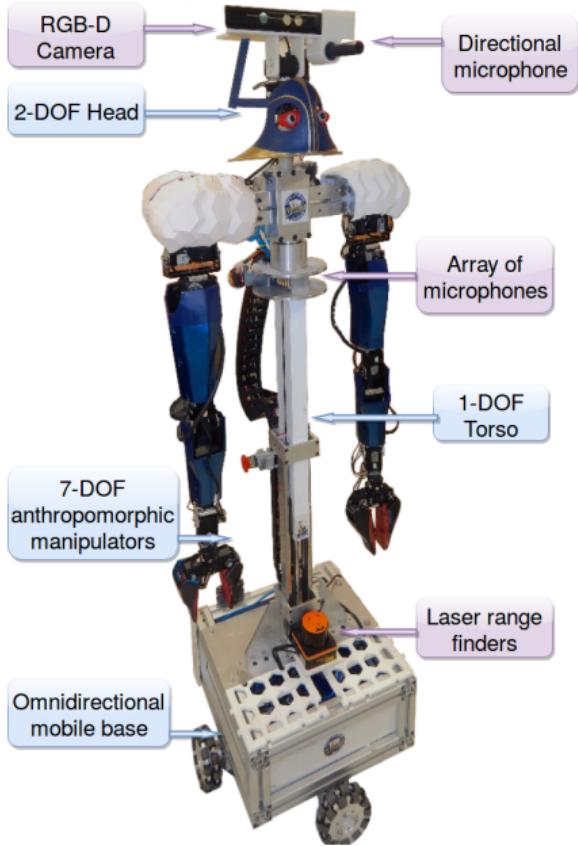
Planeación de acciones

Introducción

Los robots de servicio doméstico

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



Hardware necesario: Base móvil

- ▶ 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.



Hardware necesario: Cámaras



- ▶ 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.

Hardware necesario: Sensor láser

- ▶ 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.



Hardware necesario: Manipulador



- ▶ 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.

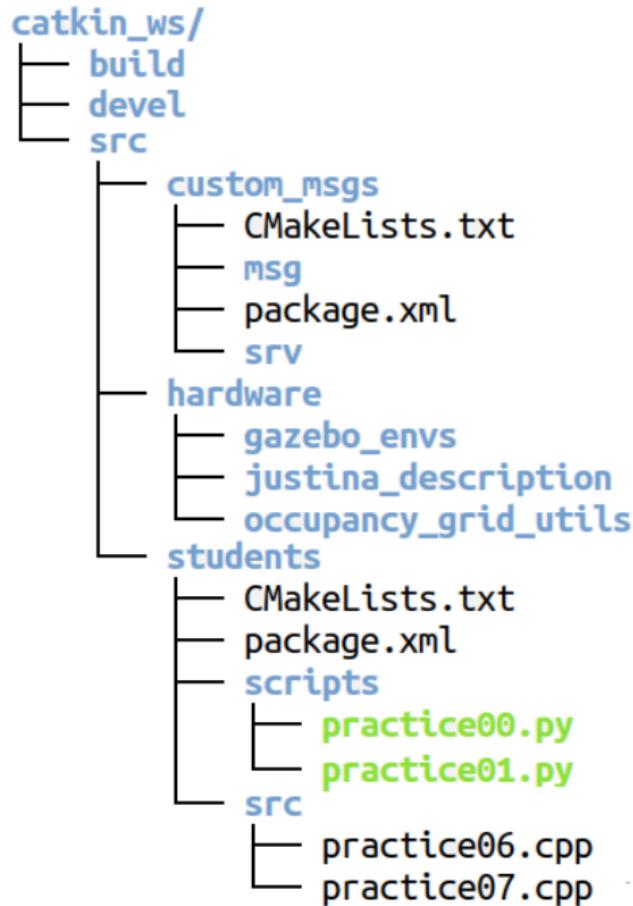
ROS se puede entender en dos grandes niveles conceptuales:

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

Sistema de archivos

Recursos en disco:

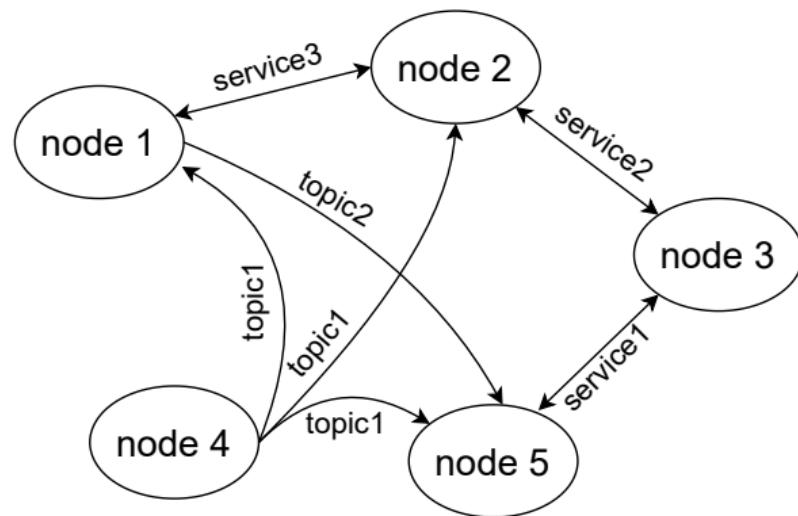
- ▶ **Workspace:** carpeta que contiene los paquetes desarrollados
- ▶ **Paquetes:** Principal unidad de organización del software en ROS (concepto heredado de Linux)
- ▶ **Manifiesto:** (`package.xml`) provee metadatos sobre el paquete (dependencias, banderas de compilación, información del desarrollador)
- ▶ **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.



Grafo de procesos

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



Tópicos y servicios

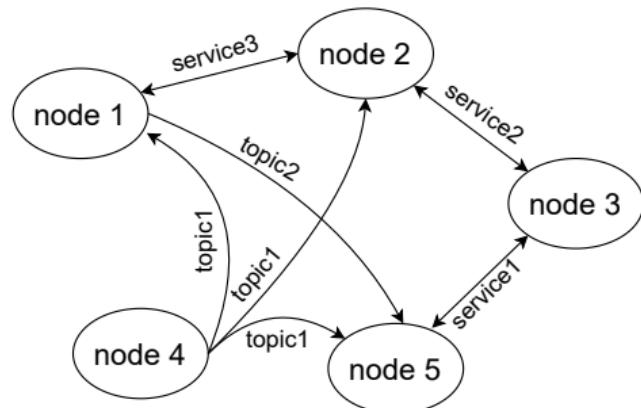
Los nodos (procesos) en ROS intercambian información a través de dos grandes patrones:

► 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 principales:

- ▶ Navegación: encontrar un conjunto de puntos $q \in Q_{free}$ que permitan al robot moverse desde una configuración inicial q_{start} a una configuración final q_{goal} .
- ▶ Mapeo: construir una representación del ambiente a partir de las lecturas de los sensores y la trayectoria del robot.
- ▶ Localización: determinar la configuración q dado un mapa y lecturas de los sensores.
- ▶ Barrido: pasar un actuador por todos los puntos $q \in Q_b \subset Q$.

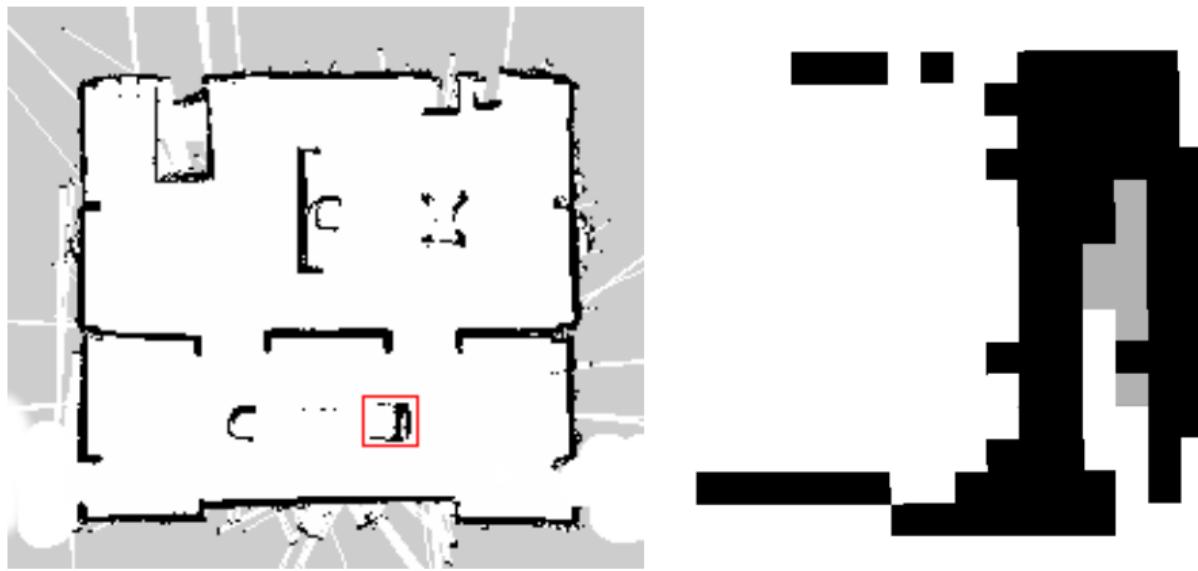
Representación del ambiente

Un mapa es cualquier representación del ambiente útil en la toma de decisiones.

- ▶ Interiores (se suelen representar en 2D)
 - ▶ Celdas de ocupación
 - ▶ Mapas de líneas
 - ▶ Mapas topológicos: Diagramas de Voronoi generalizados.
 - ▶ Mapas basados en *Landmarks*
- ▶ Exteriores (suelen requerir una representación 3D)
 - ▶ Celdas de elevación
 - ▶ Celdas de ocupación 3D
 - ▶ Octomaps

Celdas de ocupación

Es un tipo de mapa geométrico. El espacio se discretiza con una resolución determinada y a cada celda se le asigna un número $p \in [0, 1]$ que indica su nivel de ocupación. En un enfoque probabilístico este número se puede interpretar como la certeza que se tiene de que una celda esté ocupada.



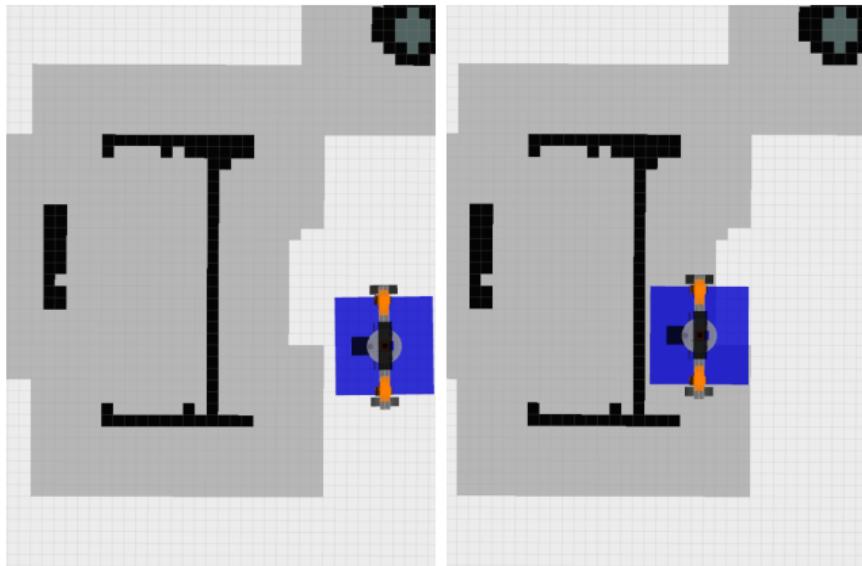
El mapa resultante se representa en memoria mediante una matriz de valores de ocupación. En ROS, los mapas utilizan el mensaje `nav_msgs/OccupancyGrid`.

Ejercicio 1 - Celdas de ocupación

1. Abra una terminal (Ctrl + Alt + t) y ejecute el comando `roslaunch bring_up path_planning.launch`
2. Inspeccione el mapa de celdas de ocupación
3. Detenga la ejecución (Ctrl + C)
4. Abra el archivo `catkin_ws/src/config_files/maps/appartment.pgm` con cualquier editor de imágenes y realice alguna modificación (puede ser agregar una mancha negra o borrar algún obstáculo).
5. Ejecute de nuevo la simulación y observe los cambios

Inflado de celdas de ocupación

Aunque las celdas de ocupación representan el espacio donde hay obstáculos y donde no, en realidad, el robot no puede posicionarse en todas las celdas libres, debido a su tamaño, como se observa en la figura:



- ▶ Celdas blancas: espacio libre.
- ▶ Celdas negras: espacio con obstáculos.
- ▶ Celdas grises: espacio sin obstáculos donde el robot no puede estar debido a su tamaño.

- ▶ Un mapa de celdas de ocupación debe *inflarse* antes de usarse para planear rutas.
- ▶ Esta operación se conoce como *dilatación* y es un operador morfológico como se verá en la sección de conceptos de visión.
- ▶ El inflado se usa para planeación de rutas, no para localización.

Inflado de celdas de ocupación

Algoritmo 1: Algoritmo de inflado de mapas

Data:

Mapa M de celdas de ocupación

Radio de inflado r_i

Result: Mapa inflado M_{inf}

M_{inf} = Copia de M

foreach $i \in [0, \dots, \text{rows}]$ **do**

foreach $j \in [0, \dots, \text{cols}]$ **do**

 //Si la celda está ocupada, marcar como ocupadas las r_i celdas de alrededor.

if $M[i, j] == 100$ **then**

foreach $k_1 \in [-r_i, \dots, r_i]$ **do**

foreach $k_2 \in [-r_i, \dots, r_i]$ **do**

$M_{inf}[i + k_1, j + k_2] = 100$

end

end

end

end

Ejercicio 2 - Inflado de mapas

1. Ejecute la simulación con el comando `roslaunch bring_up path_planning.launch`
2. Ejecute el algoritmo de inflado de mapas `rosrun exercises map_inflation.py`
3. Mediante la GUI, modifique el valor de inflación en el campo correspondiente y observe los cambios en el visualizador RViz.

Mapas de líneas

También son mapas geométricos, pero al almacenar *features* requieren mucho menos memoria. La desventaja es la dificultad para extraer líneas del ambiente y la poca precisión en el empateado.



Algunos métodos para extraer líneas:

- ▶ *Split and merge*
- ▶ Transformada Hough (se verá en la sección de visión computacional)
- ▶ RANSAC

Algoritmo *Split and Merge*

Se utiliza principalmente cuando los datos provienen de un sensor Lidar y por lo tanto, los puntos están en secuencia.

Algoritmo 2: *Split and Merge*

Data: Conjunto de puntos P

Result: Conjunto de líneas en forma normal (ρ, θ)

Ajustar una recta L al conjunto P por mínimos cuadrados

Encontrar el punto p_i más lejano a la recta

if $d(p_i, L) > umbral$ **then**

 Dividir P en dos subconjuntos P_1 y P_2 usando p_i como pivote

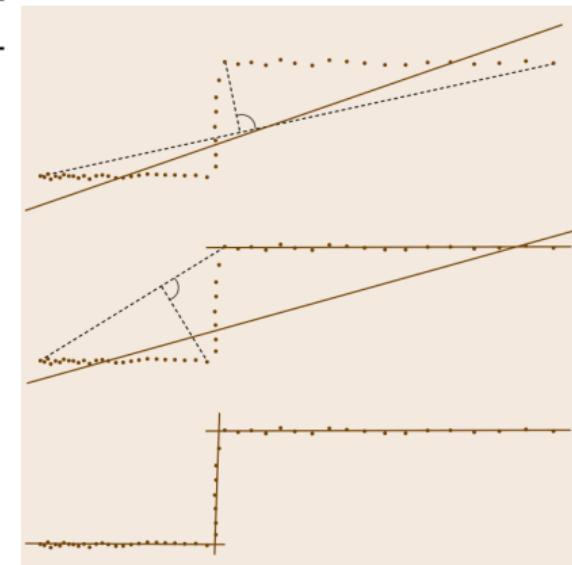
 Aplicar este algoritmo recursivamente para P_1 y P_2

 Devolver las rectas de ambos subconjuntos

else

 Devolver la recta L en forma normal (ρ, θ)

end



Mínimos cuadrados

Este método busca minimizar las distancias entre los puntos (x_i, y_i) y la recta en forma normal dada por los parámetros (ρ, θ) .

Dado un conjunto de puntos (x_i, y_i) , la recta (ρ, θ) que mejor se ajusta se puede obtener con:

$$\begin{aligned}\theta &= \frac{1}{2} \operatorname{atan2} \left(-2 \sum_i (\bar{x} - x_i)(\bar{y} - y_i) , \sum_i [(\bar{y} - y_i)^2 - (\bar{x} - x_i)^2] \right) \\ \rho &= \bar{x} \cos \theta + \bar{y} \sin \theta\end{aligned}$$

con

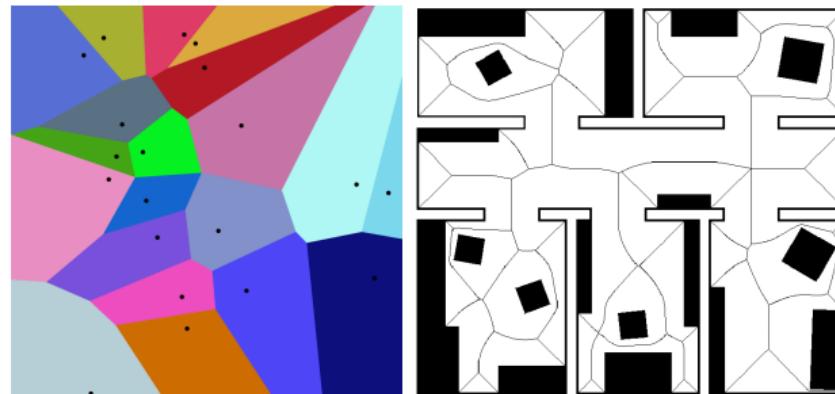
$$\begin{aligned}\bar{x} &= \frac{1}{n} \sum_i x_i \\ \bar{y} &= \frac{1}{n} \sum_i y_i\end{aligned}$$

Ejercicio 3 - Mapas de líneas

1. Ejecute la simulación con el comando `roslaunch bring_up path_planning.launch`
2. Ejecute el algoritmo *split and merge* con el comando: `rosrun exercises split_and_merge.py`
3. Mediante la GUI, mueva el robot en el espacio libre y observe el desempeño del algoritmo

Diagrama de Voronoi Generalizado

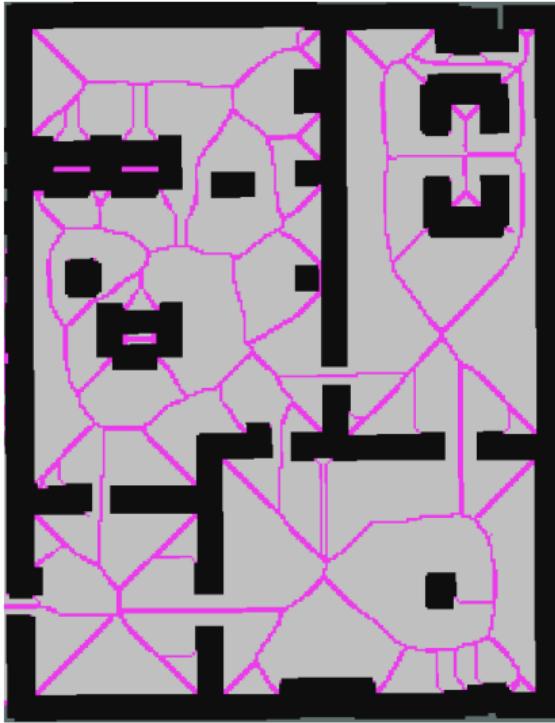
- ▶ A diferencia de los mapas geométricos, donde se busca reflejar la forma exacta del ambiente, los **mapas topológicos** buscan representar solo las relaciones espaciales de los puntos de interés.
- ▶ Los Diagramas de Voronoi dividen el espacio en regiones. Cada región está asociada a un punto llamado semilla, sitio o generador. Una región asociada a una semilla x contiene todos los puntos p tales que $d(x, p)$ es menor o igual que la distancia $d(x', p)$ a cualquier otra semilla x' .
- ▶ Un diagrama de Voronoi generalizado (GVD) considera que las semillas pueden ser objetos con dimensiones y no solo puntos.



- ▶ La forma de las regiones depende de la función de distancia que se utilice.

El algoritmo *Brushfire*

- ▶ Obtener un GVD es aún un problema abierto
- ▶ Se simplifica el problema si se asume que el espacio está representado por Celdas de Ocupación
- ▶ En este caso el GVD se puede obtener mediante el algoritmo *Brushfire*
- ▶ El mapa de rutas mostrado en la figura se forma con las celdas que son máximos locales en el mapa de distancias devuelto por Brushfire, es decir, son las celdas que son fronteras entre las regiones de Voronoi.
- ▶ Estas celdas también son aquellas equidistantes a los dos obstáculos más cercanos.



El algoritmo *Brushfire*

Algoritmo 3: Brushfire

Data: Mapa de celdas de ocupación M

Result: Distancias de cada celda al objeto más cercano

Fijar $d(p) = 0$ para toda celda p en los obstáculos

Fijar $d(p) = -1$ para toda celda p en el espacio libre

Crear una cola Q y agregar toda p en los obstáculos

while Q no esté vacía **do**

x = desencolar de Q

forall celdas p vecinas de x **do**

if $d(p) == -1$ **then**

 Aregar p a Q

 Fijar $d(p) = x + d(p, x)$

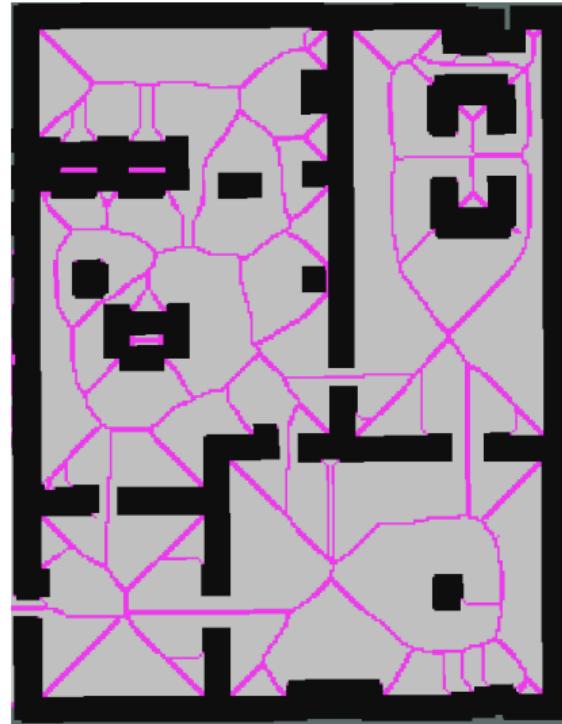
else

 Fijar $d(p) = \min(d(p), x + d(p, x))$

end

end

end



Ejercicio 4 - Diagrama de Voronoi Generalizado

1. Ejecute la simulación con el comando `roslaunch bring_up path_planning.launch`
2. Ejecute el inflado de mapas: `rosrun exercises map_inflation.py`
3. Ejecute el algoritmo de brushfire mediante el comando `rosrun exercises voronoi.py`
4. Observe el diagrama resultante en el visualizador RViz.

Planeación de rutas

La planeación de rutas consiste en encontrar una secuencia de puntos $q \in Q_{free}$ que permitan al robot moverse desde una configuración inicial q_{start} hasta una configuración final q_{goal} .

- ▶ Una **ruta** es solo la secuencia de configuraciones para llegar a la meta.
- ▶ Cuando la secuencia de configuraciones se expresa en función del tiempo, entonces se tiene una **trayectoria**.

En este curso solo vamos a hacer planeación de rutas, no de trayectorias (para navegación).

Existen varios métodos para planear rutas. La mayoría de ellos se pueden agrupar en:

- ▶ Métodos basados en muestreo
- ▶ Métodos basados en grafos

Como su nombre lo indica, consisten en tomar muestras aleatorias del espacio libre. Si es posible llegar en línea recta de la configuración actual al punto muestrado, entonces se agrega a la ruta. Ejemplos:

- ▶ RRT (Rapidly-exploring Random Trees)
- ▶ RRT-Bidireccional
- ▶ RRT-Extendido

Rapidly-exploring Random Trees

Consiste en construir un árbol a partir de muestras aleatorias del espacio libre.

Algoritmo 4: RRT

Data: Mapa, q_s = Punto origen

Result: Espacio explorado

Árbol[0] = q_s ;

$k = 0$;

while $k < k_{max}$ **do**

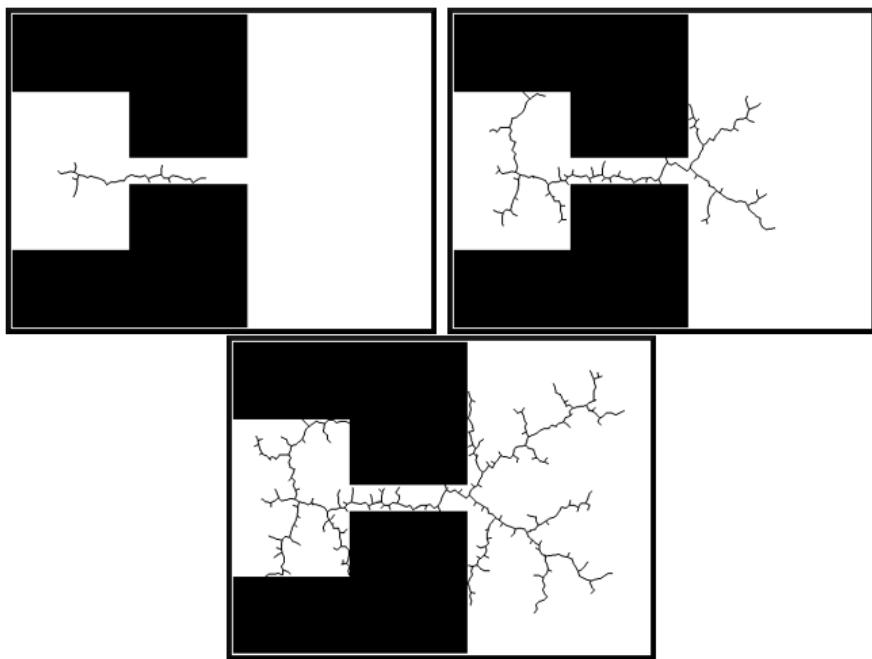
q_r = ConfiguracionAleatoria();

 Extiende(Árbol, q_r);

$k++$;

end

return Árbol;



Métodos basados en grafos

Estos métodos consideran el ambiente como un grafo. En el caso de celdas de ocupación, cada celda libre es un nodo que está conectado con las celdas vecinas que también estén libres. Los pasos generales de este tipo de algoritmos se pueden resumir en:

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 $CL = \emptyset$

Agregar n_s a OL

Nodo actual $n_c = n_s$

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

 Seleccionar n_c de OL bajo algún criterio

 Agregar n_c a CL

 Expandir n_c

 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

Métodos basados en grafos

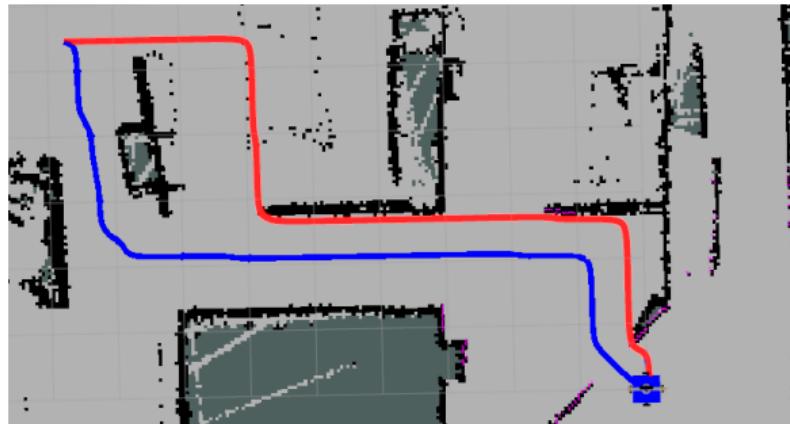
El criterio para seleccionar el siguiente nodo a expandir n_c de la lista abierta, determina el tipo de algoritmo:

- ▶ Criterio FIFO: Búsqueda a lo ancho BFS (la lista abierta es una cola)
- ▶ Criterio LIFO: Búsqueda en profundidad DFS (la lista abierta es una pila)
- ▶ Menor valor g : Dijkstra (la lista abierta es una cola con prioridad)
- ▶ Menor valor f : A* (la lista abierta es una cola con prioridad)

Si el costo g para ir de una celda a otra es siempre 1, entonces Dijkstra es equivalente a BFS. A* y Dijkstra siempre calculan la misma ruta pero A* lo hace más rápido.

Mapas de costo

- ▶ Los métodos como Dijkstra y A* minimizan una función de costo. Esta función podría ser distancia, tiempo de recorrido, número de vuelta, energía gastada, entre otras.
- ▶ En este curso se empleará como costo una combinación de distancia recorrida más peligro de colisión (cercanía a los obstáculos).
- ▶ De este modo, las rutas serán un equilibrio entre rutas cortas y rutas seguras.



Mapas de costo

- Se utilizará como costo una función de cercanía.
- Se calcula de forma similar al algoritmo Brushfire, pero la función decrece conforme nos alejamos de los objetos.

Algoritmo 5: Mapa de costo

Data:

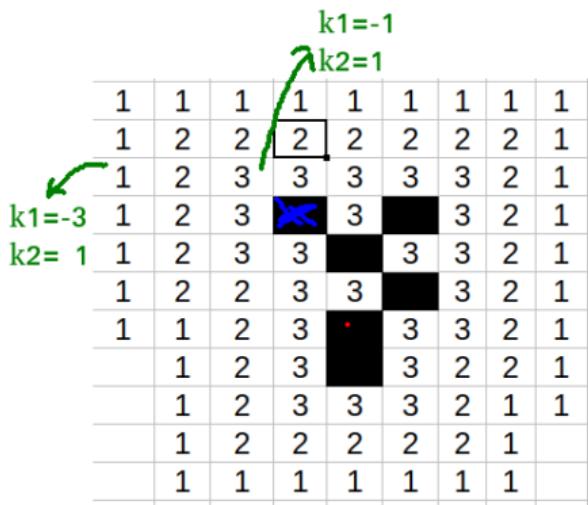
Mapa M de celdas de ocupación

Radio de costo r_c

Result: Mapa de costo M_c

M_c = Copia de M

```
foreach i ∈ [0, ..., rows) do
    foreach j ∈ [0, ..., cols) do
        //Si está ocupada, calcular el costo de  $r_c$  celdas alrededor.
        if M[i, j] == 100 then
            foreach k1 ∈ [-rc, ..., rc] do
                foreach k2 ∈ [-rc, ..., rc] do
                    C = rc - max(|k1|, |k2|) + 1
                    Mc[i + k1, j + k2] = max(C, Mc[i + k1, j + k2])
            end
        end
    end
end
end
```



El algoritmo A*

- ▶ Es un algoritmo completo, es decir, si la ruta existe, seguro la encontrará, y si no existe, lo indicará en tiempo finito.
- ▶ Al igual que Dijkstra, A* encuentra una ruta que minimiza una función de costo, es decir, es un algoritmo óptimo.
- ▶ Es un algoritmo del tipo de búsqueda informada, es decir, utiliza información sobre el estimado del costo restante para llegar a la meta para priorizar la expansión de ciertos nodos.
- ▶ El nodo a expandir se selecciona de acuerdo con la función:

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

donde

- ▶ $g(n)$ es el costo acumulado del nodo n
- ▶ $h(n)$ es una función heurística que **subestima** el costo de llegar del nodo n al nodo meta n_g .
- ▶ Se tienen los siguientes conjuntos importantes:
 - ▶ Lista abierta: conjunto de todos los nodos en la frontera (visitados pero no conocidos). Es una cola con prioridad donde los elementos son los nodos y la prioridad es el valor $f(n)$.
 - ▶ Lista cerrada: conjunto de nodos para los cuales se ha calculado una ruta óptima.
- ▶ A cada nodo se asocia un valor $g(n)$, un valor $f(n)$ y un nodo padre $p(n)$.

El algoritmo A*

Data: Mapa M , nodo inicial n_s con configuración q_s , nodo meta n_g con configuración q_g

Result: Ruta óptima $P = [q_s, q_1, q_2, \dots, q_g]$

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

Fijar $f(n_s) = 0$, $g(n_s) = 0$ y $prev(n_s) = NULL$

Agregar n_s a OL y fijar nodo actual $n_c = n_s$

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

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

forall n vecino de n_c **do**

$g = g(n_c) + costo(n_c, n)$

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

$g(n) = g$

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

$prev(n) = n_c$

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

while $n_c \neq NULL$ **do**

 | Insertar al inicio de la ruta P la configuración correspondiente al nodo n_c

 | $n_c = prev(n_c)$

end

Devolver ruta óptima P

El algoritmo A*

- ▶ La función de costo será el número de celdas más el mapa de costo de la clase anterior.
- ▶ Puesto que el mapa está compuesto por celdas de ocupación, los nodos vecinos se pueden obtener usando conectividad 4 o conectividad 8.
- ▶ Si se utiliza conectividad 4, la distancia de Manhattan es una buena heurística.
- ▶ Si se utiliza conectividad 8, se debe usar la distancia Euclídea.
- ▶ La lista abierta se puede implementar con una *Heap*, de este modo, la inserción de los nodos n se puede hacer en tiempo logarítmico y la selección del nodo con menor f se hace en tiempo constante.
- ▶ La obtención de las coordenadas (x, y) a partir de los nodos n se puede hacer con:

$$\begin{aligned}x &= (c)\delta + M_{ox} \\y &= (r)\delta + M_{oy}\end{aligned}$$

- ▶ La obtención del renglón-columna (r, c) del nodo n a partir de (x, y) , se puede obtener con:

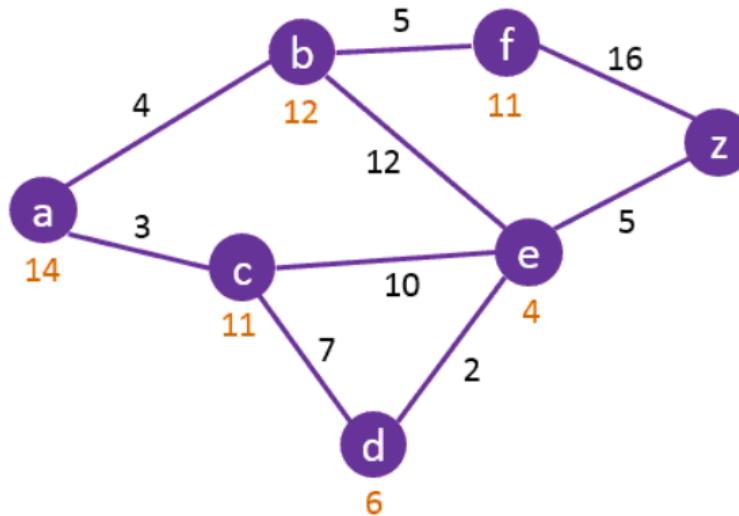
$$\begin{aligned}r &= \text{int}((y - M_{oy})/\delta) \\c &= \text{int}((x - M_{ox})/\delta)\end{aligned}$$

donde

- ▶ (M_{ox}, M_{oy}) es el origen del mapa, es decir, las coordenadas cartesianas de la celda $(0,0)$.
- ▶ δ es la resolución, es decir, el tamaño de cada celda.
- ▶ La función *int()* convierte a entero el argumento.
- ▶ Todos estos valores están en los metadatos del mapa.

El algoritmo A*

Ejemplo: ¿Cuál es la ruta óptima del nodo A al nodo Z?



El algoritmo A*

Paso	Nodo actual	Lista Cerrada	Lista abierta
0	NULL	\emptyset	{A}
1	A	{A}	{B, C}
2	C	{A, C}	{B, D, E}
3	B	{A, C, B}	{D, E, F}
4	D	{A, C, B, D}	{E, F}
5	E	{A, C, B, D, E}	{F, Z}
6	Z	{A, C, B, D, E, Z}	{F }

A g,f,p	B g,f,p	C g,f,p	D g,f,p	E g,f,p	F g,f,p	Z g,f,p
0,0,NULL	$\infty, \infty, \text{NULL}$					
0,0,NULL	4, 16, A	3, 14, A	$\infty, \infty, \text{NULL}$	$\infty, \infty, \text{NULL}$	$\infty, \infty, \text{NULL}$	$\infty, \infty, \text{NULL}$
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	13, 17, C	$\infty, \infty, \text{NULL}$	$\infty, \infty, \text{NULL}$
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	13, 17, C	9, 20, B	$\infty, \infty, \text{NULL}$
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	12, 16, D	9, 20, B	$\infty, \infty, \text{NULL}$
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	12, 16, D	9, 20, B	17, 17, E
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	12, 16, D	9, 20, B	17, 17, E

Ejercicio 5 - Planeación de rutas

1. Ejecute la simulación con el comando `roslaunch bring_up path_planning.launch`
2. Ejecute el algoritmo de inflado de mapas `rosrun exercises map_inflation.py`
3. Ejecute el cálculo de mapas de costo `rosrun exercises map_cost`
4. Ejecute el algoritmo A* `rosrun exercises a_star.py`
5. Mediante la GUI, fije un punto meta y observe la ruta calculada en el visualizador RViz.
6. Detenga el nodo de A*
7. Abra el archivo `catkin_ws/src/exercises/scripts/a_start.py`, comente la línea 43 y descomente la línea 44.
8. Ejecute nuevamente el algoritmo A*
9. Fije un punto meta mediante la GUI y observe los cambios en las rutas calculadas.

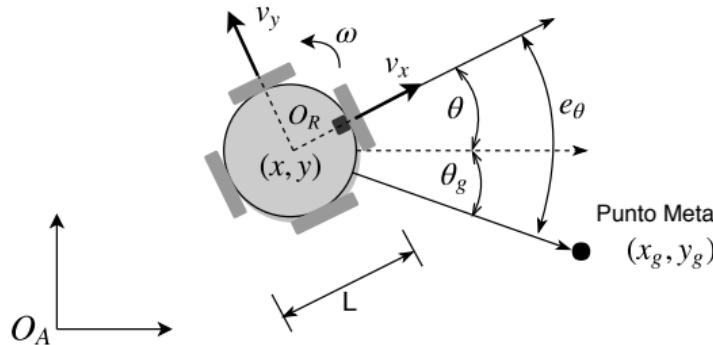
Seguimiento de rutas

Hasta el momento ya se tiene una representación del ambiente y una forma de planear rutas. Ahora falta diseñar las leyes de control que hagan que el robot se mueva por la ruta calculada. Este control se hará bajo los siguientes supuestos:

- ▶ Se conoce la posición del robot (más adelante se abordará el problema de la localización)
- ▶ El modelo cinemático es suficiente para modelar el movimiento del robot
- ▶ Las dinámicas no modeladas (parte eléctrica y mecánica de los motores) son lo suficientemente rápidas para poder despreciarse

Modelo cinemático

Considere la base móvil omnidireccional de la figura con configuración $q = (x, y, \theta)$.



El modelo cinemático está dado por

$$\begin{aligned}\dot{x} &= v_x \cos \theta - v_y \sin \theta \\ \dot{y} &= v_x \sin \theta + v_y \cos \theta \\ \dot{\theta} &= \omega,\end{aligned}$$

- ▶ (v_x, v_y, ω) se consideran como señales de control
- ▶ Corresponden a las velocidades lineales frontal y lateral, y la velocidad angular, con respecto al robot.
- ▶ La forma de convertir (v_x, v_y, ω) a velocidades de cada motor varía dependiendo del número de motores y de su posición.

Control de posición

- ▶ Las leyes de control se diseñarán considerando una base diferencial
- ▶ Es mejor mover al robot así, pues los sensores están generalmente al frente

Si se quiere alcanzar el punto meta (x_g, y_g) , las siguientes leyes de control siguientes permiten alcanzar dicho punto meta:

$$\begin{aligned}v_x &= v_{max} e^{-\frac{e_\theta^2}{\alpha}} \\ \omega &= \omega_{max} \left(\frac{2}{1 + e^{-\frac{e_\theta}{\beta}}} - 1 \right)\end{aligned}$$

con

$$e_\theta = \text{atan2}(y_g - y, x_g - x) - \theta$$

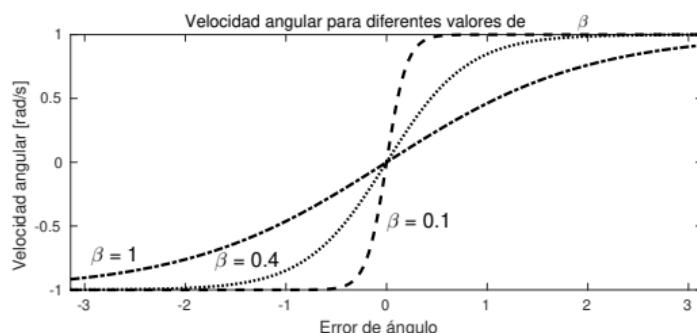
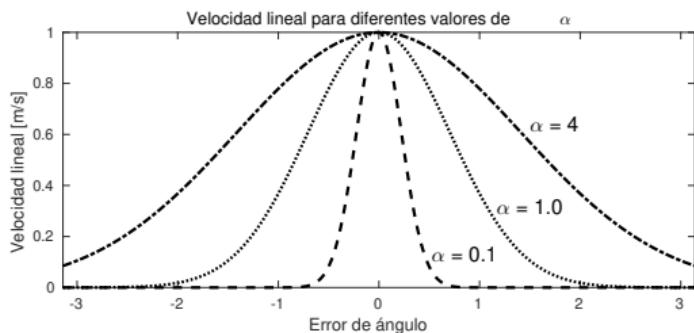
El error de ángulo e_θ debe estar siempre en el intervalo $(-\pi, \pi]$. Si la diferencia resulta en un valor fuera de este ángulo, se puede acotar mediante:

$$e_\theta \leftarrow (e_\theta + \pi) \% (2\pi) - \pi$$

donde $\%$ denota el operador módulo (residuo).

Control de posición

- ▶ v_{max} y ω_{max} son las velocidades linear y angular máximas y dependen de las capacidades físicas del robot.
- ▶ α y β determinan qué tan rápido varían dichas velocidades cuando cambia el error de ángulo.
- ▶ En general, valores pequeños de α y β logran que el robot alcance el punto meta casi en línea recta, sin embargo, valores muy pequeños pueden producir oscilaciones.
- ▶ Valores grandes de α y β producen un movimiento más suave pero puede hacer que el robot describa curvas muy extensas.



Seguimiento de rutas

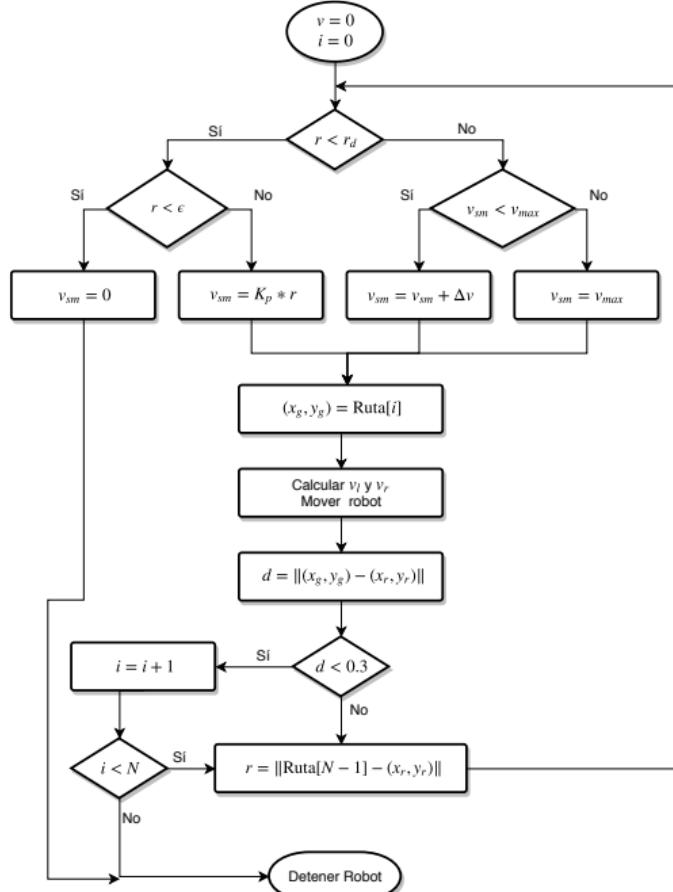
- ▶ Hasta el momento se ha planteado cómo alcanzar una posición, pero, ¿para una ruta?
- ▶ Las rutas son secuencias de puntos. Esta secuencia se prodría parametrizar con respecto al tiempo para tener una trayectoria, sin embargo esto resulta muy complicado debido a la complejidad de las rutas.
- ▶ Una solución más sencilla es aplicar el control de posición para cada punto hasta recorrer toda la ruta.
- ▶ Las leyes de control solo dependen de e_θ por lo que el robot no desacelera al acercarse a la meta, provocando fuertes oscilaciones.
- ▶ Una forma de resolver este problema es ejecutar la ley de control sólo si la distancia al punto meta

$$d = \sqrt{(x_g - x_r)^2 + (y_g - y_r)^2}$$

es mayor que una tolerancia ϵ .

- ▶ En este caso, el robot se detendrá abruptamente cuando el error de distancia sea menor que ϵ , lo cual tampoco es deseable
- ▶ Una forma fácil de hacer que el robot acelere y desacelere, o en general, obtener un perfil de velocidad, es mediante el uso de una máquina de estados

Perfil de velocidad



Considere una máquina de estados que calcule v_{max} en el control. Sea v_{sm} la nueva velocidad lineal máxima, de modo que ahora se tiene:

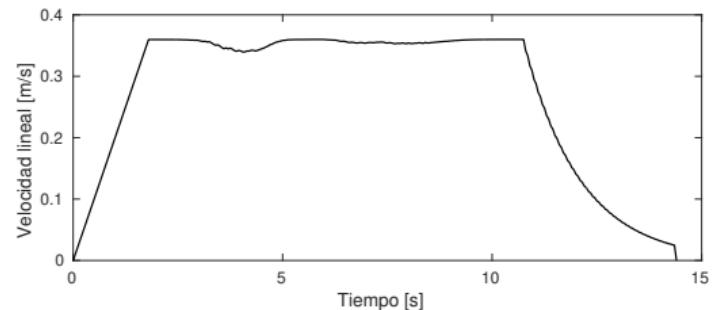
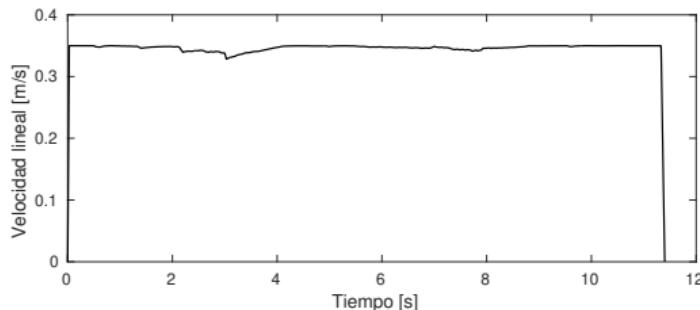
$$v = v_{sm} e^{-\frac{e^2}{\alpha}}$$
$$\omega = \omega_{max} \left(\frac{2}{1 + e^{-\frac{e^2}{\beta}}} - 1 \right)$$

con

- ▶ r : Distancia a la meta global
- ▶ ϵ : Distancia a la que se considera que el robot alcanzó la meta global
- ▶ r_d : Distancia a la meta global para desacelerar
- ▶ Δv : Aceleración

Perfil de velocidad

La siguiente figura muestra un ejemplo de una ruta y las velocidades lineales generadas usando solo las leyes de control (izquierda) y usando la máquina de estados para un perfil de velocidad (derecha).



Ejercicio 6 - Seguimiento de rutas

1. Ejecute la simulación, el inflado de mapas, mapas de costo, el algoritmo A* y el control con el comando `roslaunch bring_up navigation.launch`
2. En el simulador RViz, con el botón “2D Nav Goal”, indique una posición meta y observe el comportamiento.

Evasión de obstáculos

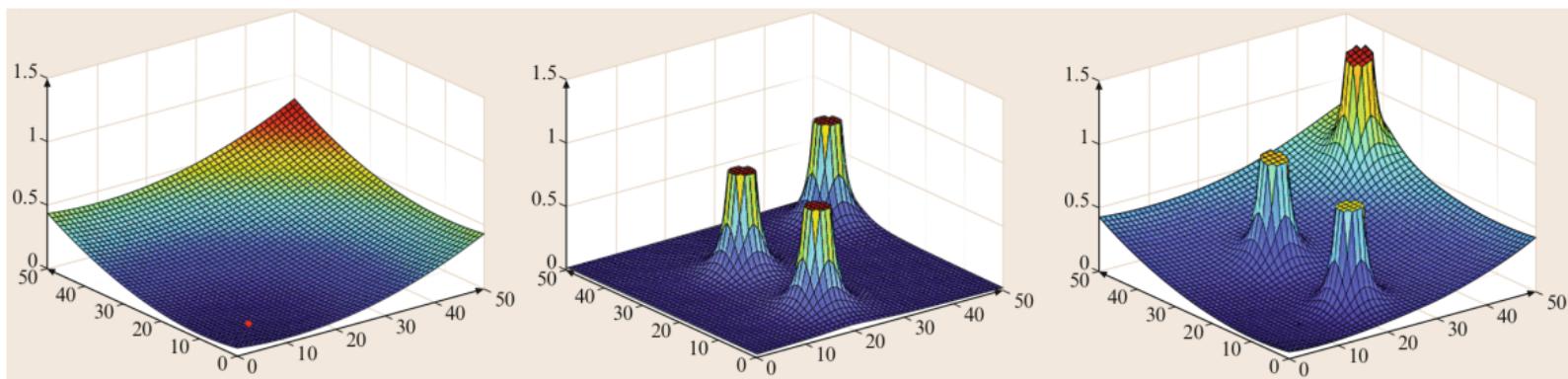
- ▶ Hasta el momento se tiene una manera de representar el ambiente, planear una ruta y seguirla
- ▶ ¿Qué pasa si en el ambiente hay un obstáculo que no estaba en el mapa?
- ▶ Se requiere de una técnica reactiva para evadir obstáculos
- ▶ Una posible solución es el uso de campos potenciales artificiales

Campos potenciales artificiales

El objetivo de esta técnica es diseñar una función $U(q) : \mathbb{R}^n \rightarrow \mathbb{R}$ que represente energía potencial.

- ▶ El gradiente $\nabla U(q) = \left[\frac{\partial U}{\partial q_1}, \dots, \frac{\partial U}{\partial q_n} \right]$ es una fuerza.
- ▶ Se debe diseñar de modo que tenga un mínimo global en el punto meta y máximos locales en cada obstáculo.
- ▶ Si el robot se mueve siempre en sentido contrario al gradiente ∇U llegará al punto meta siguiendo una ruta alejada de los obstáculos.
- ▶ Hay varias formas de diseñar la función $U(q)$, algunas son:
 - ▶ Algoritmo *wavefront*, requiere una discretización del espacio (requiere mapa previo), pero no presenta mínimos locales.
 - ▶ Campos atractivos y repulsivos, no requieren mapa previo, pero pueden presentar mínimos locales.

Potenciales atractivos y repulsivos



- ▶ **Campos repulsivos:** Por cada obstáculo se diseña una función $U_{rej_i}(q)$ con un máximo local en la posición q_{o_i} del obstáculo.
- ▶ **Campo atractivo:** Se diseña una función $U_{att}(q)$ con un mínimo global en el punto meta q_g .
- ▶ La función potencial total $U(q)$ se calcula como

$$U(q) = U_{att}(q) + \frac{1}{N} \sum_{i=1}^N U_{rej_i}(q)$$

Fuerzas atractiva y repulsivas

Puesto que el gradiente es un operador lineal, se pueden diseñar directamente las fuerzas atractiva $F_{att}(q) = \nabla U_{att}(q)$ y repulsivas $F_{rej_i}(q) = \nabla U_{rej_i}(q)$, de modo que la fuerza total será:

$$\nabla U(q) = F(q) = F_{att}(q) + \frac{1}{N} \sum_{i=1}^N F_{rej_i}(q)$$

Una propuesta de estas fuerzas es:

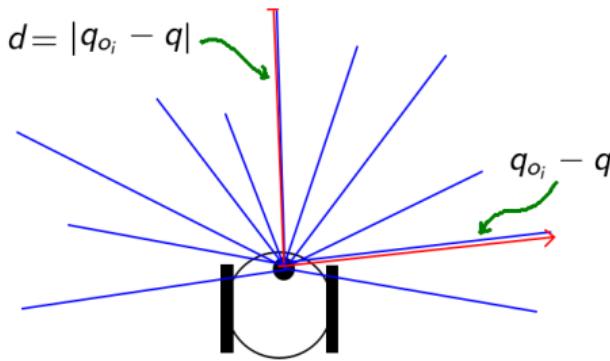
$$F_{att} = \zeta \frac{(q - q_g)}{\|q - q_g\|}, \quad \zeta > 0$$
$$F_{rej} = \begin{cases} \eta \left(\sqrt{\frac{1}{d} - \frac{1}{d_0}} \right) \frac{q_{o_i} - q}{d} & \text{si } d < d_0 \\ 0 & \text{en otro caso} \end{cases}$$

donde

- ▶ $q = (x, y)$ es la posición del robot
- ▶ $q_g = (x_g, y_g)$ es el punto que se desea alcanzar
- ▶ $q_{o_i} = (x_{o_i}, y_{o_i})$ es la posición del i -ésimo obstáculo
- ▶ d_0 es una distancia de influencia. Más allá de d_0 los obstáculos no producen efecto alguno
- ▶ ζ y η , junto con d_0 , son constantes de sintonización

Evasión de obstáculos por campos potenciales

- ▶ Aunque las ecuaciones anteriores suponen que se conoce la posición de cada obstáculo q_{oi} , en realidad ésta aparece siempre en la diferencia $q_{oi} - q$, es decir, solo se requiere su posición relativa al robot.
- ▶ Los campos potenciales se implementan utilizando el lidar, donde cada lectura se considera un obstáculo.



Las lecturas del lídar generalmente son pares distancia-ángulo (d_i, θ_i) expresados con respecto al robot, por lo que, si se conoce la posición del robot (x_r, y_r, θ_r) , la posición de cada obstáculo se puede calcular como:

$$\begin{aligned}x_{oi} &= x_r + d_i \cos(\theta_i + \theta_r) \\y_{oi} &= y_r + d_i \sin(\theta_i + \theta_r)\end{aligned}$$

Evasión de obstáculos por campos potenciales

Finalmente, para que el robot alcance el punto de menor potencial, se puede emplear el descenso del gradiente:

Algoritmo 6: Descenso del gradiente para mover al robot a través de un campo potencial.

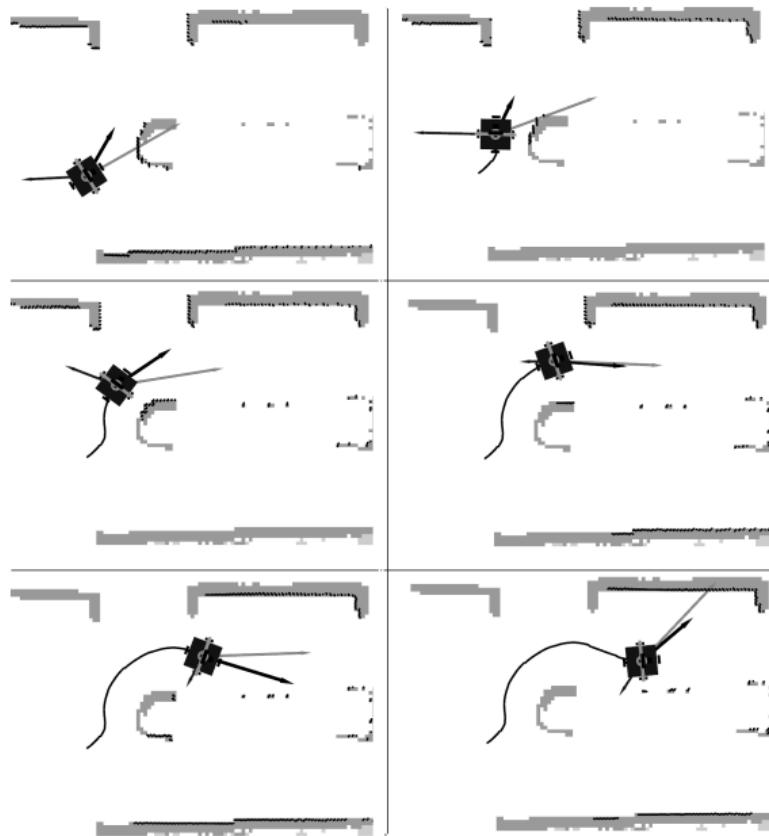
Data: Posición inicial q_s , posición meta q_g , posiciones q_{oi} de los obstáculos y tolerancia tol

Result: Secuencia de puntos $\{q_0, q_1, q_2, \dots\}$ para evadir obstáculos y alcanzar el punto meta

```
q ←  $q_s$ 
while  $\|\nabla U(q)\| > tol$  do
    |   q ←  $q - \epsilon F(q)$ 
    |   [ $v, \omega$ ] ← leyes de control con  $q$  como posición deseada
end
```

Evasión de obstáculos por campos potenciales

Ejemplo de movimiento:



Ejercicio 7 - Evasión de obstáculos

1. Ejecute la simulación con el comando `roslaunch bring_up obstacle_avoidance.launch`
2. Ejecute la evasión por campos potenciales mediante el comando
`rosrun exercises pot_fields.py`
3. Con la opción *2D Nav Goal* del visualizador *RViz*, seleccione un punto meta en el mapa.
4. Observe el comportamiento.
5. Detenga la ejecución de los campos potenciales.
6. Abra el archivo `catkin_ws/src/exercises/scripts/pot_fields.py` y modifique las constantes de atracción (línea 66), repulsión (línea 86) y distancia de influencia (línea 87)
7. Ejecute nuevamente los campos potenciales y observe el cambio en el comportamiento.

El problema de la localización consiste en determinar la configuración q del robot dada un mapa y un conjunto de lecturas de los sensores.

- ▶ La localización se podría lograr simplemente integrando los comandos de velocidad del robot.
- ▶ Si se conoce perfectamente la configuración inicial y el robot ejecuta perfectamente los comandos de movimiento, entonces la simple integración de la velocidad de los motores sería suficiente.
- ▶ Esto por supuesto no es posible. Se tiene incertidumbre tanto en la estimación inicial de la posición como en la ejecución de cada movimiento.
- ▶ Es decir, el robot pierde información sobre su posición en cada movimiento.

Existen principalmente dos tipos:

Localización local:

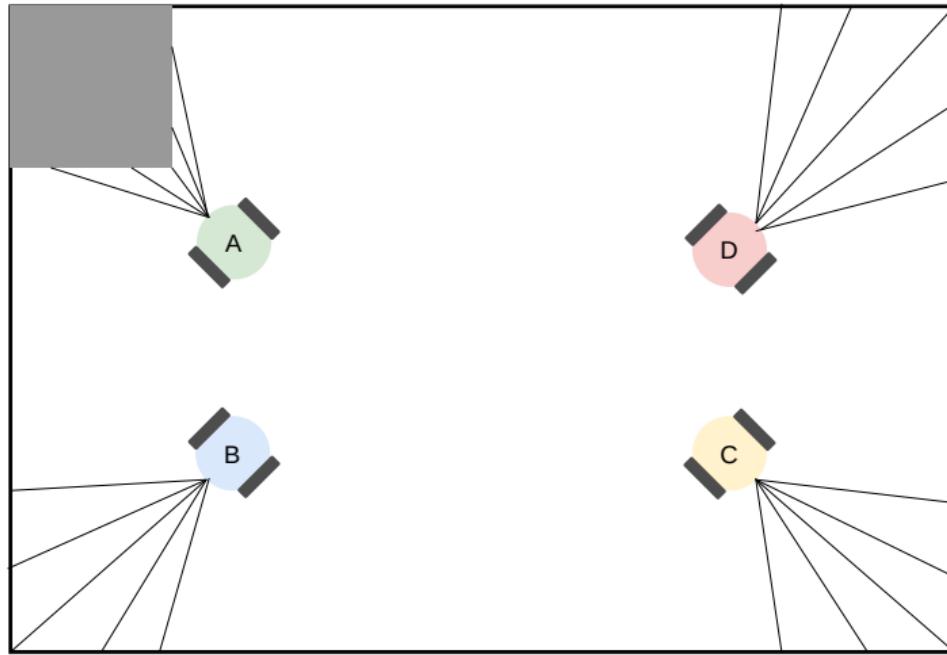
- ▶ Requiere una estimación inicial *cercana* a la posición real del robot, de otro modo, no converge.
- ▶ Suele ser menos costosa computacionalmente.
- ▶ Un método común es el Filtro de Kalman Extendido.

Localización global:

- ▶ La estimación inicial puede ser cualquiera.
- ▶ Suele ser computacionalmente costosa.
- ▶ Un método común son los Filtros de Partículas.

- ▶ En la localización probabilística, en lugar de llevar una sola hipótesis sobre la posición del robot, se mantiene una *distribución de probabilidad* sobre todo el espacio de hipótesis.
- ▶ El enfoque probabilístico permite manejar las incertidumbres inherentes al movimiento y al sensado.
- ▶ El reto es obtener una distribución de densidad de probabilidad (PDF) sobre todas las posibles posiciones del robot.
- ▶ En general, los métodos probabilísticos de estimación se componen de dos pasos:
 1. **Predicción:** Se modifica la PDF de la posición del robot con base en los comandos y el modelo de movimiento.
 2. **Actualización:** Se corrige la predicción mezclando la información de PDF predicha con información de los sensores. Se obtiene una PDF de la posición y se repite el proceso.

Filtros de Partículas



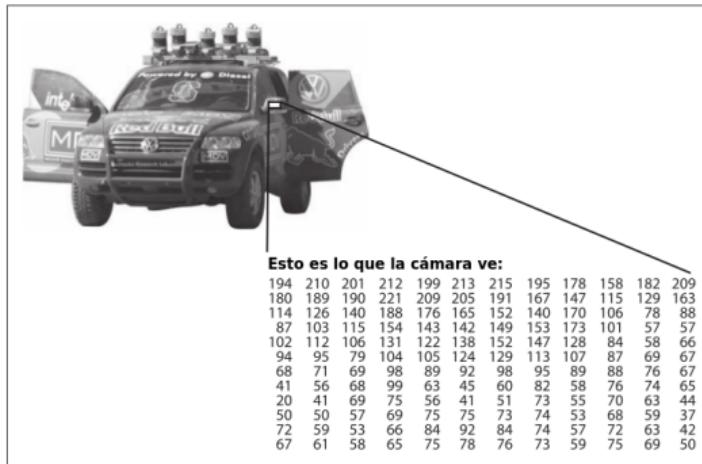
Ejercicio 8 - Filtros de partículas

1. Ejecute la simulación con el comando `roslaunch bring_up localization.launch`
2. Ejecute la localización por filtros de partículas mediante el comando `rosrun exercises particle_filters`
3. Con la GUI mueva el robot un poco hacia adelante.
4. Observe el comportamiento de las partículas conforme se mueve el robot.

Visión

Las imágenes como funciones

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).



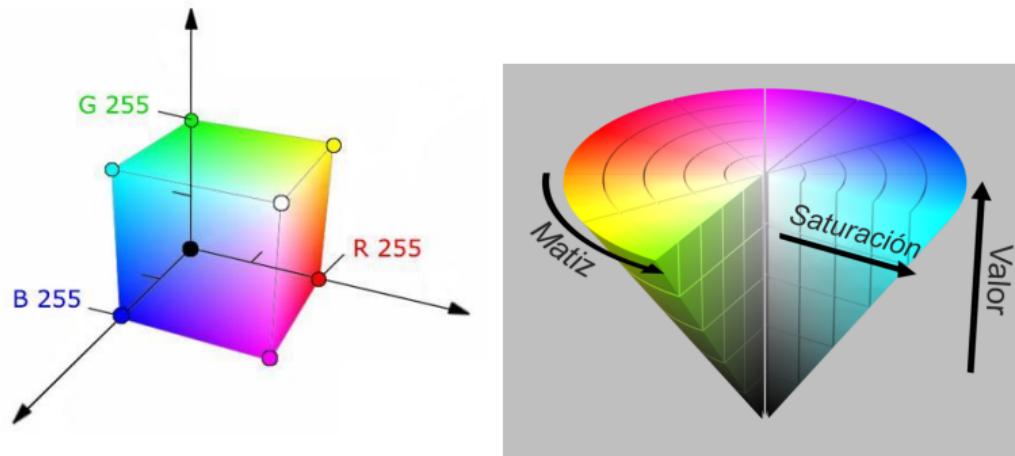
Aunque formalmente una imagen es un mapeo $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, en la práctica, tanto x, y como I son variables discretas con valores entre un mínimo y un máximo.

Espacios de color

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

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

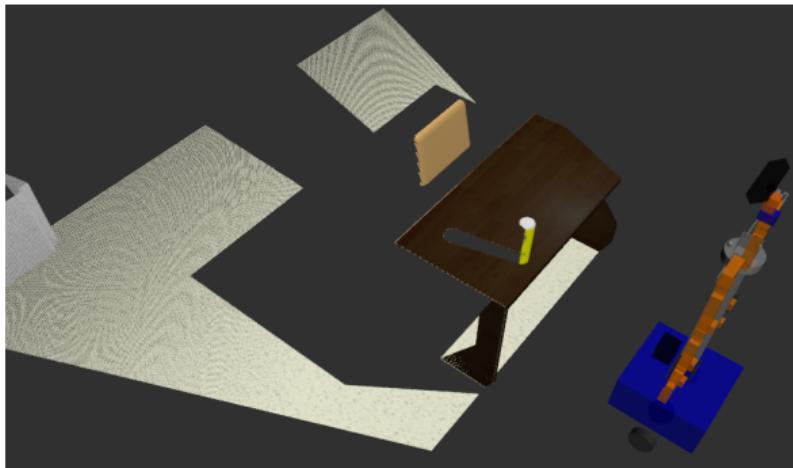
Los espacios de color son diferentes formas de representar el color mediante la combinación de un conjunto de valores.



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

Nubes de puntos

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).

Segmentación por color

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.

Ejercicio 9 - Segmentación por color

Ejecute la simulación con el comando:

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

Y en otra terminal ejecute la segmentación por color:

```
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:



En la pestaña *Simple Tasks*, teclee las palabras “pringles” o “drink” en el campo *Find Object*. Observe el resultado en el visualizador RViz.

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 ϕ radianes sobre el eje Z , seguida de una rotación de θ radianes sobre el eje Y del sistema resultante y una rotación de ψ radianes sobre el eje X del sistema rotado.

Transformaciones Homogéneas

Una Transformación Homogénea es una matriz de la forma

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

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:

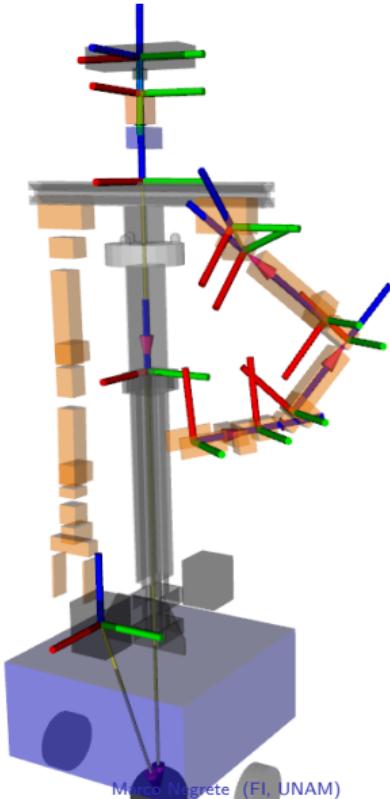
- ▶ Asociatividad: $(T_1 T_2) T_3 = T_1(T_2 T_3)$
- ▶ Inversa:

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

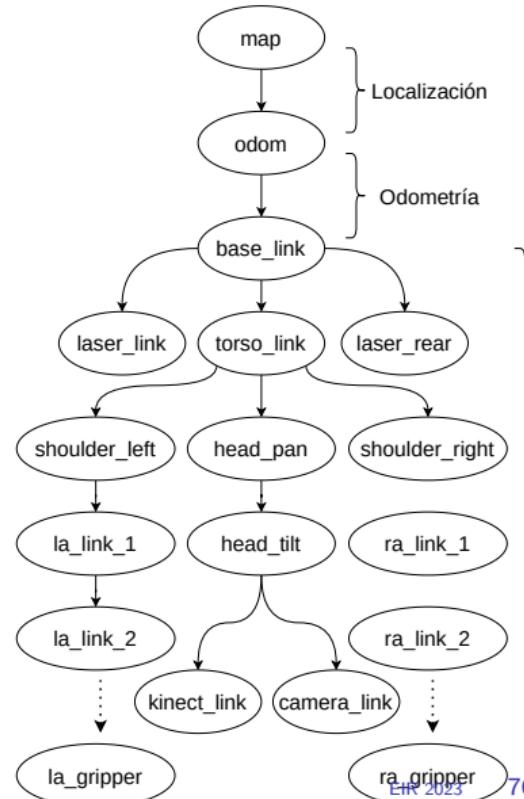
- ▶ Cancelación de índices: $T_{ab} = T_{ac} T_{cb}$

El árbol cinemático

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

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

```
1 <robot>
2   <link name="base_link">!--Forma y tamano de la base movil-->
3     <visual>
4       <origin xyz="0 0 0.235" rpy="0 0 0"/><material name="blue" />
5       <geometry> <box size="0.42 0.42 0.20" /></geometry>
6     </visual>
7   </link>
8   <link name="laser_link">!--Forma y tamano del sensor laser-->
9     <visual>
10      <origin xyz="0 0 0" rpy="0 0 0"/><material name="black" />
11      <geometry> <box size="0.08 0.08 0.1" /></geometry>
12     </visual>
13   </link>
14   <!-- Define la posicion del laser con respecto a la base movil--&gt;
15   &lt;joint name="laser_connect" type="revolute"&gt;
16     &lt;origin xyz="0.17 0 0.44" rpy="0 0 0"/&gt;
17     &lt;parent link="base_link"/&gt;&lt;child link="laser_link" /&gt;
18     &lt;axis xyz="1 0 0" /&gt;
19     &lt;dynamics damping ="0.1" /&gt;
20     &lt;limit effort="3.0" velocity="3.0" lower="-0.3" upper="1.5" /&gt;
21   &lt;/joint&gt;
22 &lt;/robot&gt;</pre>
```

Cada etiqueta `<joint>` representará una Transformación Homogénea.

El formato Xacro

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="${height}" />
19    </inertial>
20  </link>
21 </robot>
```

Ejercicio 10 - Formato Xacro

Abra el archivo `catkin_ws/src/hardware/justina_description/urdf/justina.xacro` y modifique la línea 17 de acuerdo con lo siguiente:

```
16 <joint name="shoulder_left_connect" type="fixed">
17   <origin xyz="0 0.495 -0.08" rpy="0 -1.5708 0"/>
18   <parent link="spine_link"/>
19   <child link="shoulders_left_link"/>
20 </joint>
```

Ejecute la simulación:

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

Detenga la simulación. Regrese las variables a sus valores originales.

La cinemática directa

La cinemática directa consiste en determinar la posición y orientación del efecto 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:

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

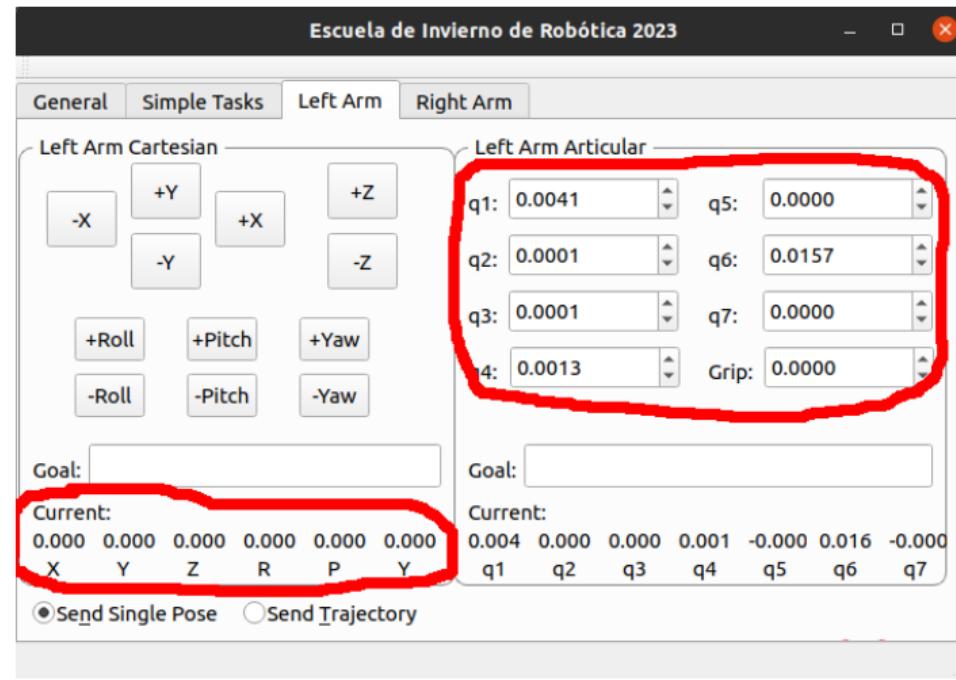
Donde R_{ab} representa la rotación del sistema b respecto al sistema a y $(dx_{ab}, dy_{ab}, dz_{ab})$ es la traslación del sistema b respecto al sistema a .

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”.

Ejercicio 11 - Cinemática directa

- ▶ Ejecute la simulación con el comando `roslaunch bring_up inverse_kinematics.launch`
- ▶ Ejecute el cálculo de la cinemática: `rosrun exercises inverse_kinematics.py`

En la GUI, mueva las articulaciones con los controles del bloque *Articular* y observe los cambios en la etiqueta *Current* del bloque *Cartesian*



La cinemática inversa

La cinemática inversa consiste en determinar las posiciones que debe tener cada articulación para que el efecto 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.

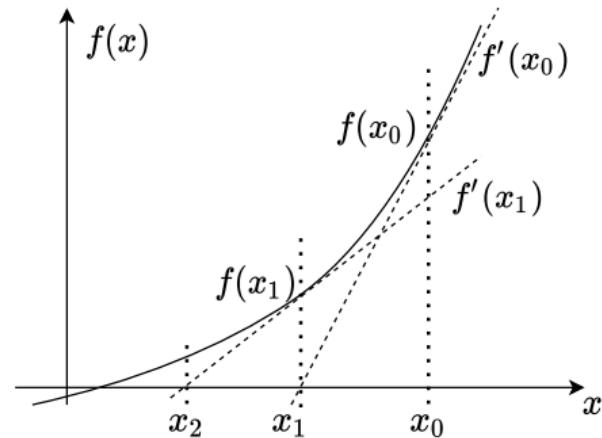
El método Newton-Raphson

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:

```
xi ← x0
while |f(x)| > ε do
    |   xi+1 ← xi -  $\frac{f(x_i)}{f'(x_i)}$ 
end
```



El Jacobiano

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 approximando las derivadas parciales con diferencias finitas:

$$J = \left[\frac{FK(q_+^1) - FK(q_-^1)}{2\Delta q} \quad \dots \quad \frac{FK(q_+^7) - FK(q_-^7)}{2\Delta q} \right]$$

$$\begin{aligned} q_+^i &= [q_1 \quad \dots \quad q_i + \Delta q \quad \dots \quad q_7] \\ q_-^i &= [q_1 \quad \dots \quad q_i - \Delta q \quad \dots \quad q_7] \end{aligned}$$

con Δq , un valor lo suficientemente pequeño para una buena aproximación de la derivada.

Cinemática Inversa por Newton-Raphson

Aplicando Newton-Raphson para resolver la ecuación:

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

Se tiene:

```
qi ← q0 //Una estimación inicial que puede ser la posición articular actual
p ← FK(qi) //La posición cartesiana que tendría el gripper con la estimación inicial
while |p - pd| > ε do
    J ← Jacobiano(q)
    qi+1 ← qi - J†(p - pd)
    p ← FK(qi)
end
```

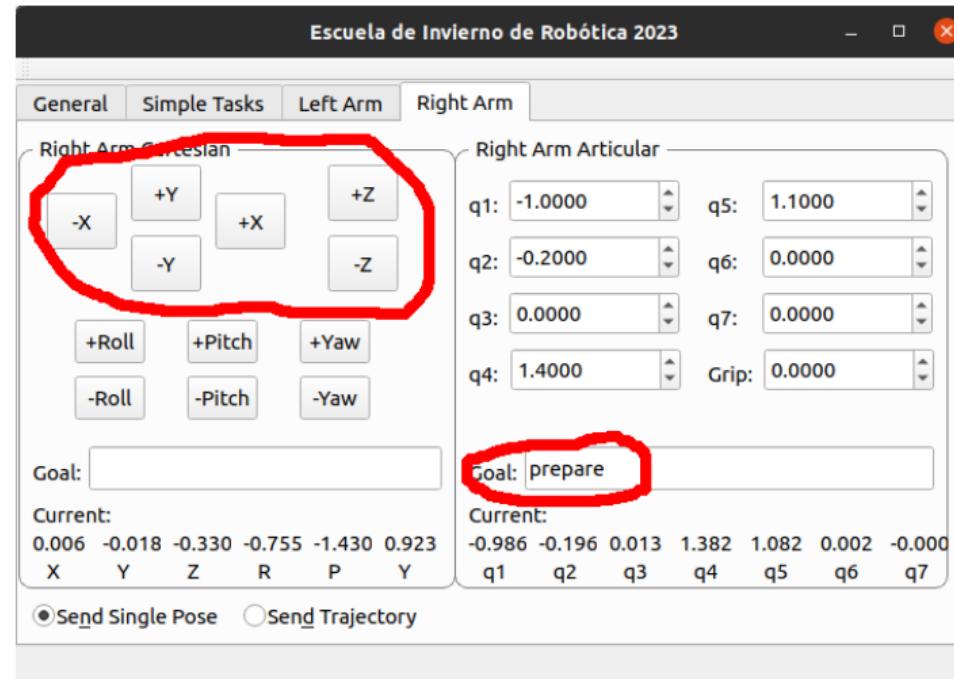
Puesto que el Jacobiano J no es una matriz cuadrada, no tiene inversa, por lo que se utiliza la matriz pseudoinversa J^\dagger .

- ▶ 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 $p - p_d$

Ejercicio 12 - Cinemática inversa

- ▶ Ejecute la simulación con el comando `roslaunch bring_up inverse_kinematics.launch`
- ▶ Ejecute el cálculo de la cinemática inversa: `rosrun exercises inverse_kinematics.py`

En la GUI, en el campo *goal* del bloque *articular*, escriba “prepare” (en cualquiera de los dos brazos):



Utilice los controles del bloque *cartesian* para mover el manipulador.

El control PID

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 τ 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 Δt , 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 sobreceso.

Los *stacks* ros_control y ros_controllers

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.

Ejercicio 13 - Control PID

Ejecute el comando:

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

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

Síntesis de voz con SoundPlay

- ▶ 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"

Ejercicio 14 - Síntesis de voz

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 synthesized 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/`

Ejercicio 14 - Síntesis de voz

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

Reconocimiento de voz con Pocketsphinx

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/>

Ejercicio 15 - Reconocimiento de voz

1. Verifique el volumen del micrófono
2. Inspeccione el archivo `catkin_ws/src/sprec_pocketsphinx/vocab/final_project.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 /hri/sp_rec/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

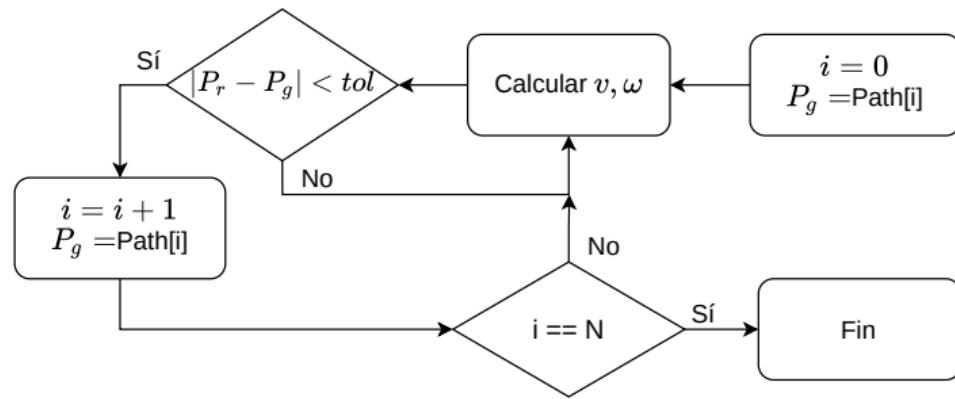
Planeación de acciones

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
- ▶ Σ : 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)

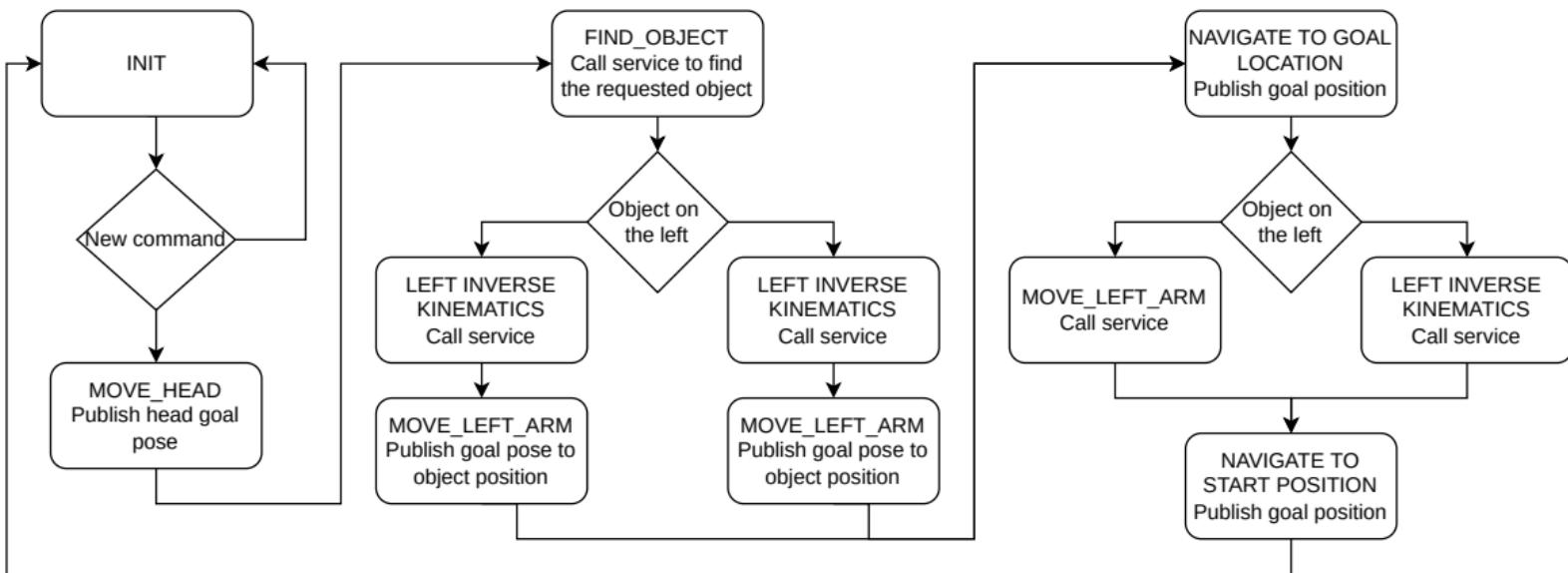
Una forma de representar los estados y transiciones de una FSM es mediante una carta ASM. Ejemplo:



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

FSM para manipular objetos

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



Ejercicio Final

Abra el archivo `catkin_ws/src/exercises/scripts/simple_service_robot.py` e inspeccione los diversos estados. Ejecute el comando:

```
1      roslaunch bring_up final_exercise.launch  
2
```

Y en otra terminal, corra la planificación para un robot de servicio simple:

```
1      rosrun exercises simple_service_robot.py  
2
```

Gracias

Contacto

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

mnegretev.info
marco.negrete@ingenieria.unam.edu