

# Tareas Básicas en Robots de Servicio Doméstico

Marco Negrete

Facultad de Ingeniería, UNAM

2o Workshop de Sistemas Ciberfísicos  
Grupo Académico MINA

## Objetivos:

- ▶ Aprender los conceptos básicos para operar un robot móvil autónomo
- ▶ Implementar dichos conceptos en un ambiente simulado
- ▶ Familiarizar al estudiante con la plataforma ROS

## 1. Introducción y generalidades

- ▶ Componentes básicos de un robot móvil
- ▶ Herramientas de software para el desarrollo de robots móviles

## 2. Planeación de movimientos

- ▶ El problema de la planeación de movimientos
- ▶ Mapas geométricos y topológicos
- ▶ Celdas de ocupación y diagramas de Voronoi
- ▶ Planeación de rutas mediante búsqueda en grafos
- ▶ Modelos cinemáticos: diferencial, omnidireccional, Ackermann
- ▶ Control de posición y seguimiento de trayectorias
- ▶ Campos potenciales artificiales

## 3. Mapeo y localización

- ▶ Localización mediante filtro de Kalman extendido
- ▶ Localización mediante filtros de partículas
- ▶ Creación de mapas mediante agrupamiento
- ▶ Localización y mapeo simultáneos

## 4. Conceptos básicos de visión artificial

- ▶ Imágenes y espacios de color
- ▶ Operadores morfológicos
- ▶ Extracción de características geométricas
- ▶ Reconocimiento mediante redes neuronales artificiales
- ▶ Nubes de puntos

## 5. Conceptos básicos de manipulación

- ▶ Movimiento de cuerpo rígido
- ▶ Cinemática directa
- ▶ Cinemática inversa por métodos numéricos
- ▶ Planeación y seguimiento de trayectorias

## 6. Herramientas para la interacción humano-robot

- ▶ Síntesis de voz con la biblioteca Festival
- ▶ Reconocimiento de voz con la biblioteca CMU Sphinx
- ▶ Reconocimiento de gestos con la biblioteca OpenPose

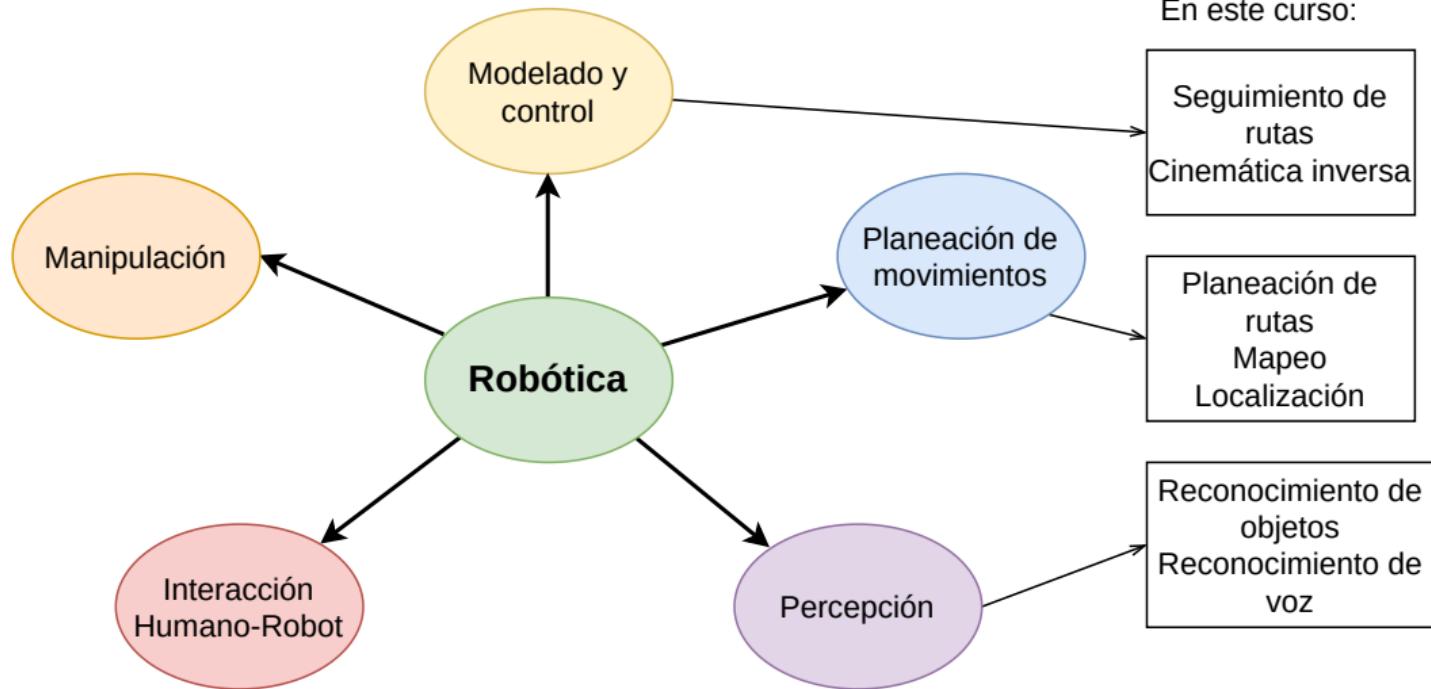
## Bibliografía recomendada:

- ▶ <https://drive.google.com/drive/folders/1gb7VQJG5eUkCvCginRHHGn5lez6VASBJ?usp=sharing>
- ▶ <https://drive.google.com/drive/folders/1Epl2b51xEJzCvzfugBD1i7xGdKYdJucy?usp=sharing>

# Contexto y definiciones

- ▶ La palabra *robot* tiene su origen en la obra *Rossum's Universal Robots* del escritor checo Karel Čapek, publicada en 1921, y su significado es “trabajo duro”.
- ▶ Latombe (1991) define un robot como un dispositivo mecánico versátil equipado con sensores y actuadores bajo el control de un sistema de cómputo [?].
- ▶ Arkin (1998) propone que un robot inteligente es una máquina capaz de extraer información de su ambiente y usar el conocimiento acerca de su mundo para moverse de manera segura y significativa, con un propósito específico [?].
- ▶ Robótica es la ciencia que estudia la conexión inteligente entre la percepción y la acción.

# Áreas de la Robótica



# Componentes de un robot

## Robot

### Sensores:

Son transductores que obtienen información del ambiente útil para la toma de decisiones.

**Propioceptivos:** sensan el estado interno del robot.

**Exteroceptivos:** sensan el ambiente externo.

**Activos:** emiten energía para realizar la medición.

**Pasivos:** no emiten energía.

Ejemplos de Sensores:

- Cámaras (RGB, RGB-D)
- Micrófonos
- Lidar
- Encoders
- Sensores de batería

### Procesadores:

Es el hardware que se utiliza para procesar información. Reciben información de los sensores y envían comandos a los actuadores.

Ejemplos de procesadores:

- CPUs
- GPUs
- FPGA
- Microcontrolador
- DSP

### Actuadores:

Son dispositivos que realizan alguna modificación al ambiente. Se pueden clasificar según su principio de actuación:

- Eléctricos
- Hidráulicos
- Neumáticos

Ejemplos de actuadores

- Motores (DC, Brushless)
- Bocinas
- Pistones
- Manipuladores

- ▶ **Configuración:** es la descripción de la posición en el espacio de todos los puntos del robot. Se denota con  $q$ .
- ▶ **Espacio de configuraciones:** es el conjunto  $Q$  de todas las posibles configuraciones.
- ▶ **Grados de libertad:** número mínimo de variables independientes para describir una configuración. En este curso, la base móvil del robot tiene 3 GdL, la cabeza tiene 2 GDL y cada brazo tiene 7 GDL más 1 GdL para el gripper. En total, el robot tiene 21 GdL.

## Propiedades del robot:

- ▶ **Holonómico:** el robot puede moverse instantáneamente en cualquier dirección del espacio de configuraciones. Comunmente se logra mediante ruedas de tipo *Mecanum* u *Omnidireccionales*.
- ▶ **No holonómico:** existen restricciones de movimiento en velocidad pero no en posición. Son restricciones que solo se pueden expresar en términos de la velocidad pero no pueden integrarse para obtener una restricción en términos de posición. Ejemplo: un coche sólo puede moverse en la dirección que apuntan las llantas delanteras, sin embargo, a través de maniobras puede alcanzar cualquier posición y orientación. El robot de este curso es no holonómico.

## Propiedades de los algoritmos:

- ▶ **Complejidad:** cuánta memoria y cuánto tiempo se requiere para ejecutar un algoritmo, en función del número de datos de entrada (número de grados libertad, número de lecturas de un sensor, entre otros).
- ▶ **Optimalidad:** un algoritmo es óptimo cuando encuentra una solución que minimiza una función de costo.
- ▶ **Completitud:** un algoritmo es completo cuando garantiza encontrar la solución siempre que ésta exista. Si la solución no existe, indica falla en tiempo finito.
  - ▶ Completitud de resolución: la solución existe cuando se tiene una discretización.
  - ▶ Completitud probabilística: la probabilidad de encontrar la solución tiende a 1.0 cuando el tiempo tiende a infinito.

Una explicación más detallada se puede encontrar en el Cap. 3 de [?].



**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.

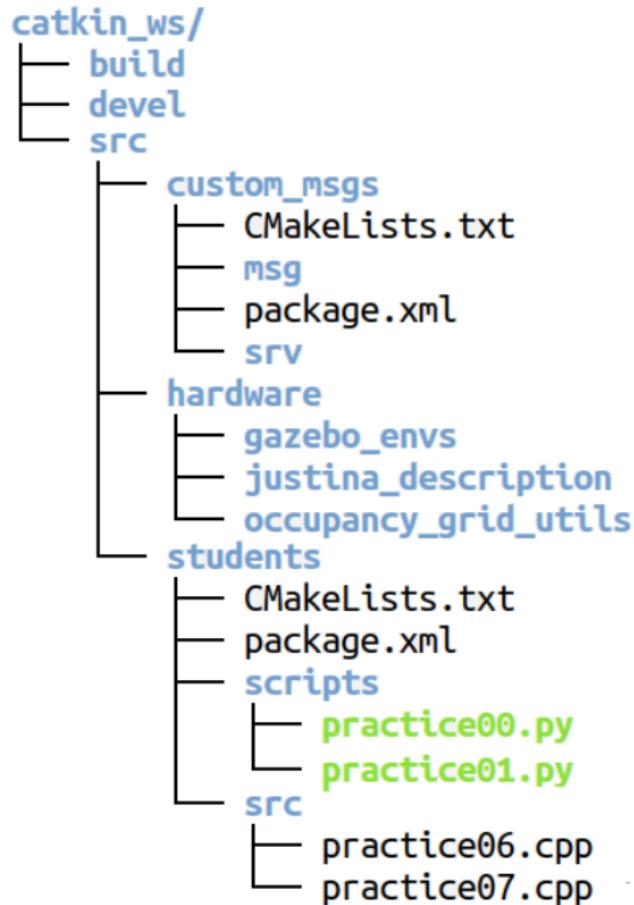
ROS se puede entender en dos grandes niveles conceptuales:

- ▶ **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:

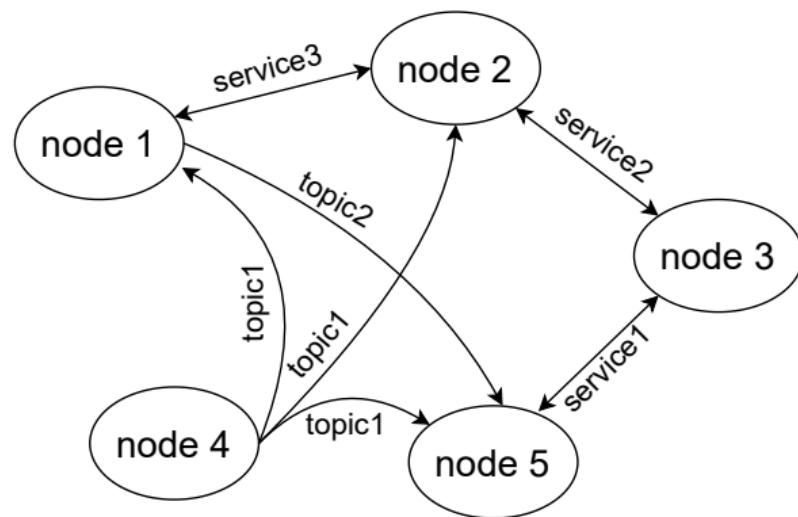
- ▶ **Workspace:** carpeta que contiene los paquetes desarrollados
- ▶ **Paquetes:** Principal unidad de organización del software en ROS (concepto heredado de Linux)
- ▶ **Manifiesto:** (`package.xml`) provee metadatos sobre el paquete (dependencias, banderas de compilación, información del desarrollador)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

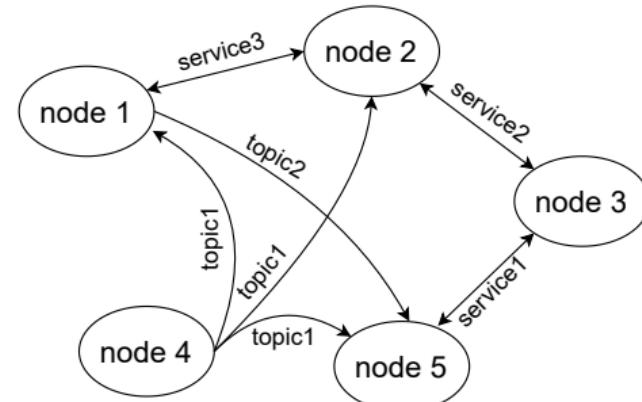
Los nodos (procesos) en ROS intercambian información a través de dos grandes patrones:

## ► 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.

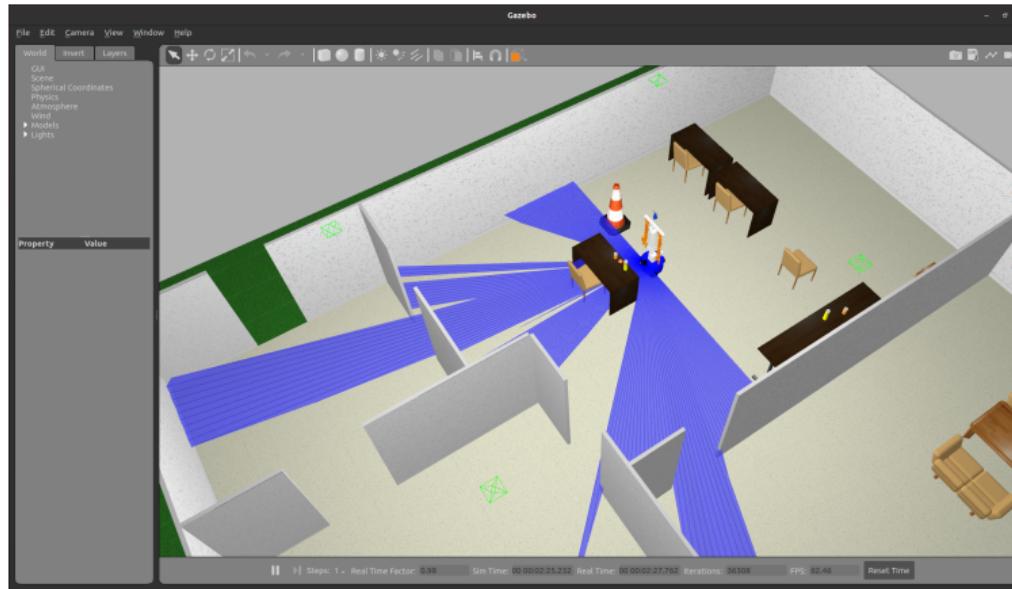


Para mayor información:

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

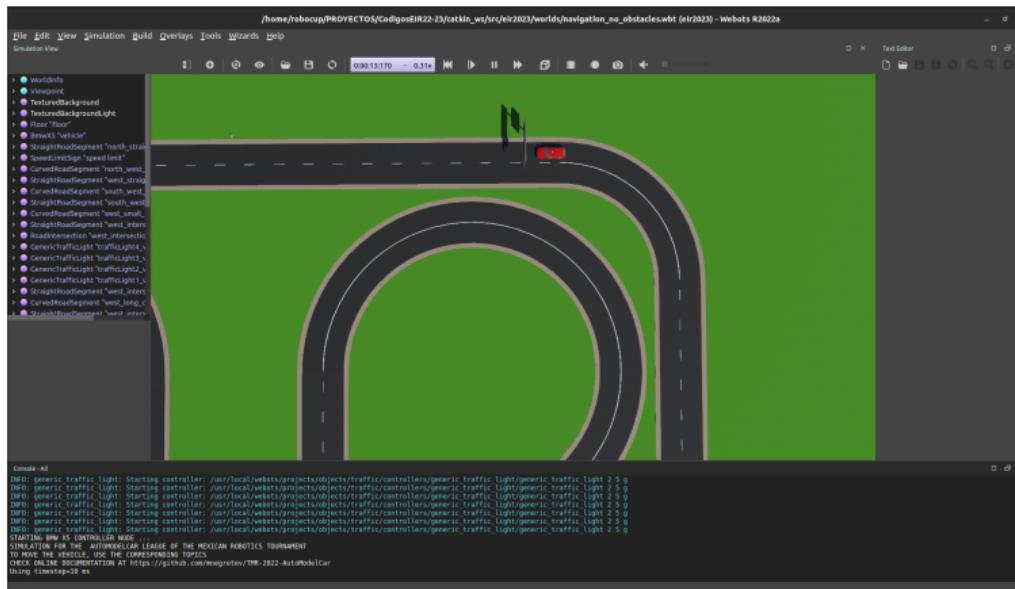
# El simulador Gazebo

- ▶ Acceso a múltiples motores de físicas de alto rendimiento como: ODE, Bullet, Simbody y DART
- ▶ Existen *plugins* para los sensores y actuadores más comunes en robots móviles (lidar, cámara RGB-D, motores, base diferencial, entre otros)
- ▶ Los componentes básicos son los *modelos* (archivos \*.sdf) y los *mundos* (archivos \*.world)
- ▶ Fácil integración con ROS



## El simulador Webots

- ▶ Motor de físicas ODE y OpenGL como motor de renderizado
  - ▶ Es multiplataforma y disponible para sistemas operativos Linux, Windows y macOS
  - ▶ Programación de robots en diferentes lenguajes de programación: C, C++, Python, Java, MATLAB e integración con la plataforma ROS
  - ▶ Amplia variedad de robots, sensores, actuadores, objetos y materiales



# Ejercicio 1

1. Abra el archivo `catkin_ws/src/students/scripts/assignment01.py` y agregue el siguiente código en la línea 25:

```
25 n = int((msg.angle_max - msg.angle_min)/msg.angle_increment/2)
26 obstacle_detected = msg.ranges[n] < 1.0
27
```

En el mismo archivo, en la línea 45, agregue el siguiente código:

```
45 msg_cmd_vel = Twist()
46 msg_cmd_vel.linear.x = 0 if obstacle_detected else 0.3
47 pub_cmd_vel.publish(msg_cmd_vel)
48
```

2. Abra una terminal y corra la simulación con el comando (revise las instrucciones en el README del repositorio):

```
roslaunch bring_up path_planning.launch
```

3. En otra terminal, corra el ejercicio 1 con el comando:

```
rosrun students assignment01.py
```

4. Observe el comportamiento y describa lo que sucede.

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  $q \in Q_b \subset Q$ .

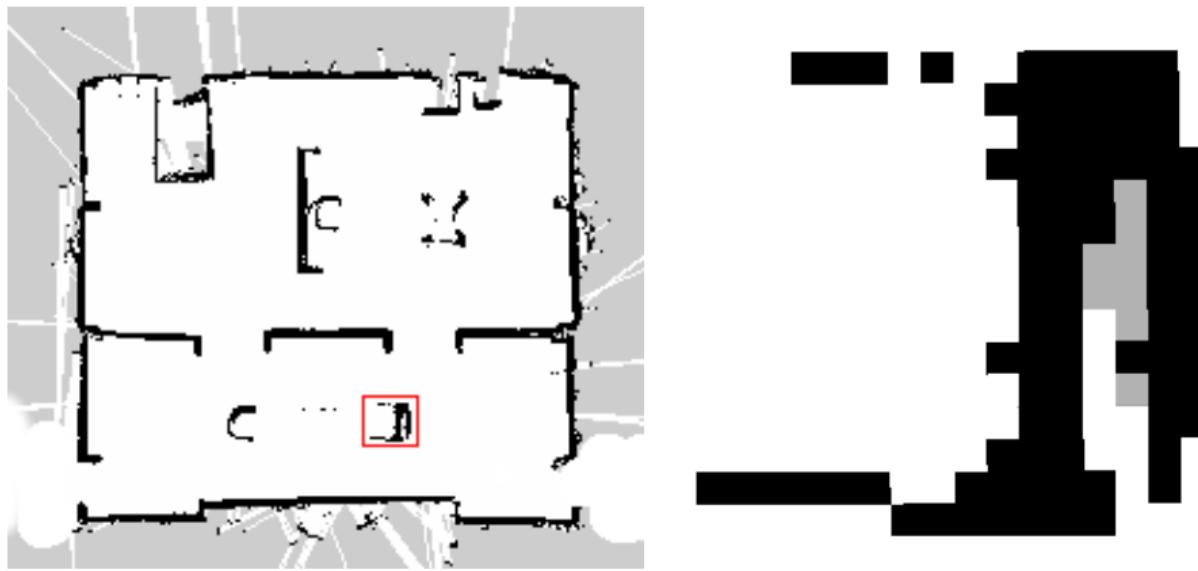
# Representación del ambiente

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

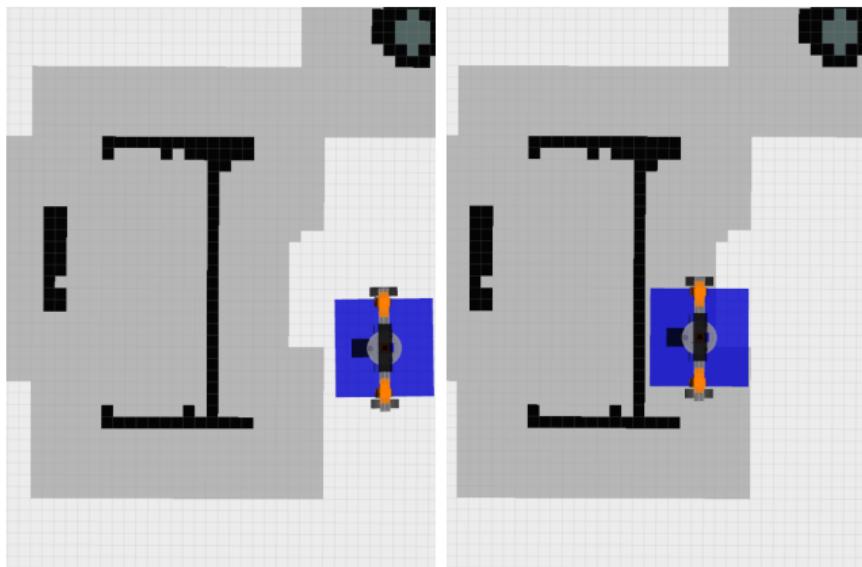
Es un tipo de mapa geométrico. El espacio se discretiza con una resolución determinada y a cada celda se le asigna un número  $p \in [0, 1]$  que indica su nivel de ocupación. En un enfoque probabilístico este número se puede interpretar como la certeza que se tiene de que una celda esté ocupada.



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

## 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**

---

# 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*

**Data:** Conjunto de puntos  $P$

**Result:** Conjunto de líneas en forma normal  $(\rho, \theta)$

Ajustar una recta  $L$  al conjunto  $P$  por mínimos cuadrados

Encontrar el punto  $p_i$  más lejano a la recta

**if**  $d(p_i, L) > d_0$  **then**

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

    Aplicar este algoritmo recursivamente para  $P_1$  y  $P_2$

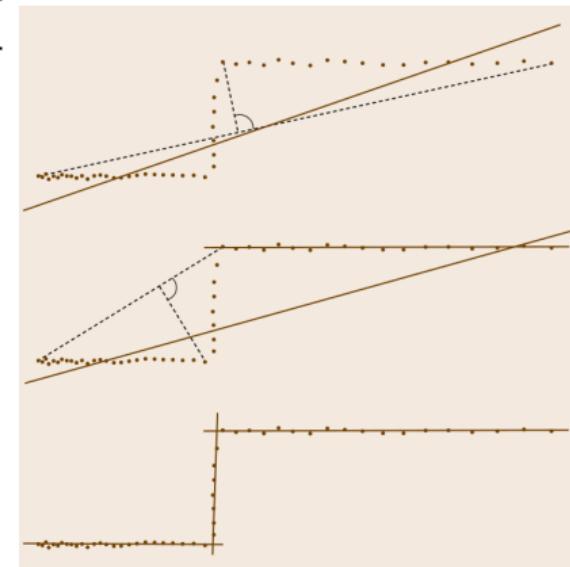
    Devolver las rectas de ambos subconjuntos

**else**

    Devolver la recta  $L$  en forma normal  $(\rho, \theta)$

**end**

---



## Algoritmo *Split and Merge*

El algoritmo anterior realiza la partición (*split*) del conjunto de puntos en posibles rectas y obtiene la ecuación para cada conjunto por mínimos cuadrados. La etapa de mezcla (*merge*) consiste simplemente en considerar como una sola recta, dos segmentos con parámetros  $(\rho, \theta)$  similares. Los parámetros a sintonizar en este algoritmo son:

- ▶  $d_0$  : umbral de distancia entre un punto  $p_i$  y la recta  $L$
- ▶  $N_{min}$  : número mínimo de puntos para considerar que un subconjunto puede ser una recta.
- ▶  $\rho_{tol}$  : valor máximo de diferencia entre dos rectas con  $\rho_1$  y  $\rho_2$  para considerarlas como una sola recta.
- ▶  $\theta_{tol}$  : valor máximo de diferencia entre dos rectas con  $\theta_1$  y  $\theta_2$  para considerarlas como una sola recta.

## 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  $(\rho, \theta)$ .

Dado un conjunto de puntos  $(x_i, y_i)$ , la recta  $(\rho, \theta)$  que mejor se ajusta se puede obtener con:

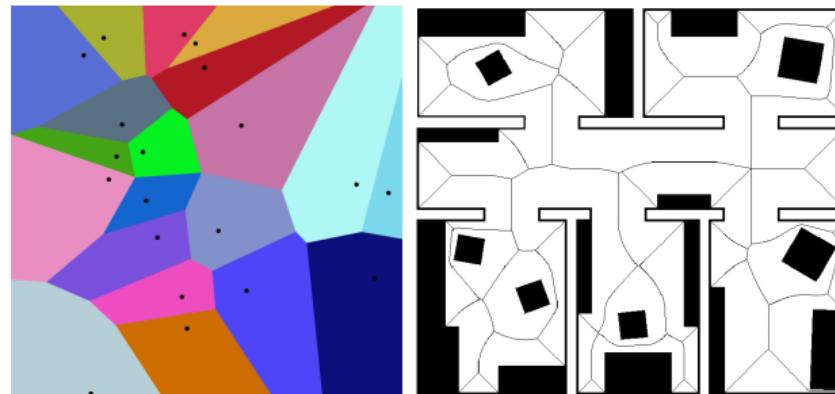
$$\begin{aligned}\theta &= \frac{1}{2} \operatorname{atan2} \left( -2 \sum_i (\bar{x} - x_i)(\bar{y} - y_i) , \sum_i [(\bar{y} - y_i)^2 - (\bar{x} - x_i)^2] \right) \\ \rho &= \bar{x} \cos \theta + \bar{y} \sin \theta\end{aligned}$$

con

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

# Diagrama de Voronoi Generalizado

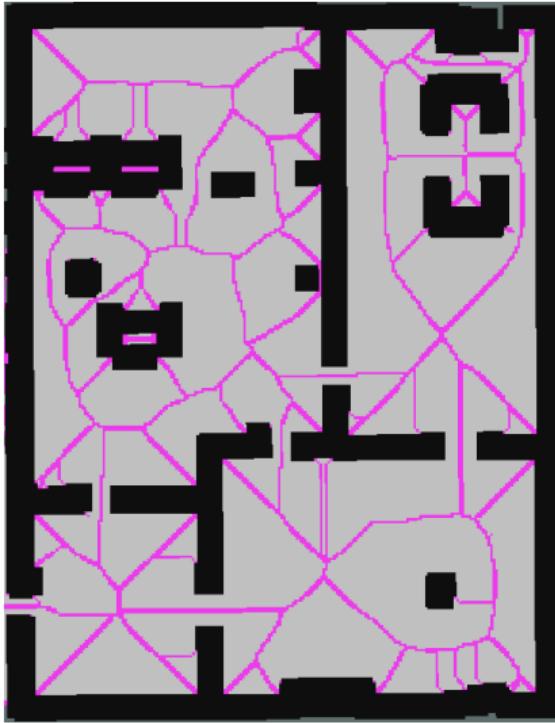
- ▶ A diferencia de los mapas geométricos, donde se busca reflejar la forma exacta del ambiente, los **mapas topológicos** buscan representar solo las relaciones espaciales de los puntos de interés.
- ▶ Los Diagramas de Voronoi dividen el espacio en regiones. Cada región está asociada a un punto llamado 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  $d(p) = x + d(p, x)$

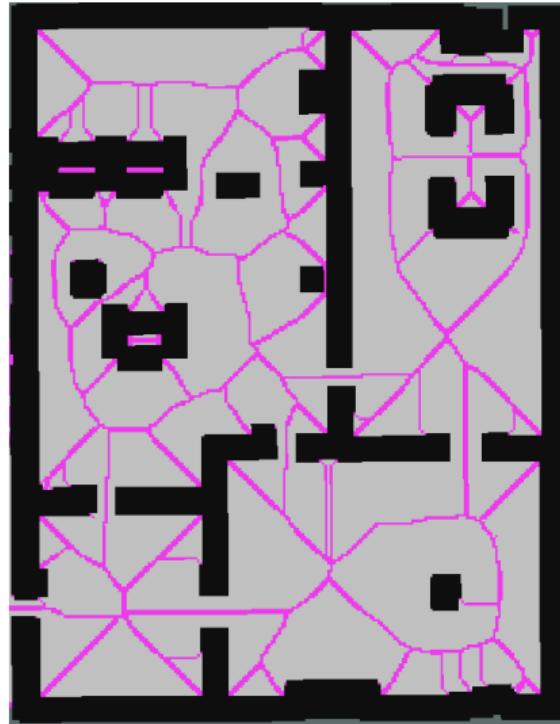
**else**

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

**end**

**end**

**end**



## Ejercicio 2 - Representación del ambiente

1. Abra el archivo `catkin_ws/src/students/scripts/assignment02a.py` e implemente el algoritmo de inflado de mapas (algoritmo 1, diapositiva 23)
2. Abra una terminal y corra la simulación con el comando (revise las instrucciones en el README del repositorio):

```
roslaunch bring_up path_planning.launch
```

3. En otra terminal, corra el ejercicio 2a con el comando:

```
rosrun students assignment02a.py _inflation_radius:=0.2
```

4. Observe lo que sucede en el visualizador RViz.

5. Detenga la práctica y ejecútela de nuevo con radios de inflado diferentes, por ejemplo:

```
rosrun students assignment02a.py _inflation_radius:=0.3
```

Observe qué sucede. ¿Qué pasa si el radio de inflado es muy grande o muy pequeño?

6. En otra terminal, corra el algoritmo *split and merge* con el comando:

```
rosrun students assignment02b.py _dist:=0.01 _points:=3 _rho:=0.05 _theta:=0.05
```

7. Detenga la detección de líneas y ejecútela nuevamente con diferentes parámetros de sintonización. Con la GUI, mueva el robot a diferentes puntos del mapa para verificar la correcta sintonización de dicho parámetros.

## Ejercicio 2 - Representación del ambiente

8. En otra terminal, corra el algoritmo *Brushfire* para obtener un Diagrama de Voronoi Generalizado a partir del mapa inflado (el ejercicio 02a debe estar corriendo):  
`rosrun students assignment02c.py`
9. Detenga el GVD y abra el archivo `catkin_ws/src/students/scripts/assignment02c.py` y en las líneas 54 y 57 cambie la distancia Euclídea por distancia de Manhattan (revise los comentarios arriba de cada línea).
10. Ejecute nuevamente la obtención del GVD y observe lo que sucede 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();

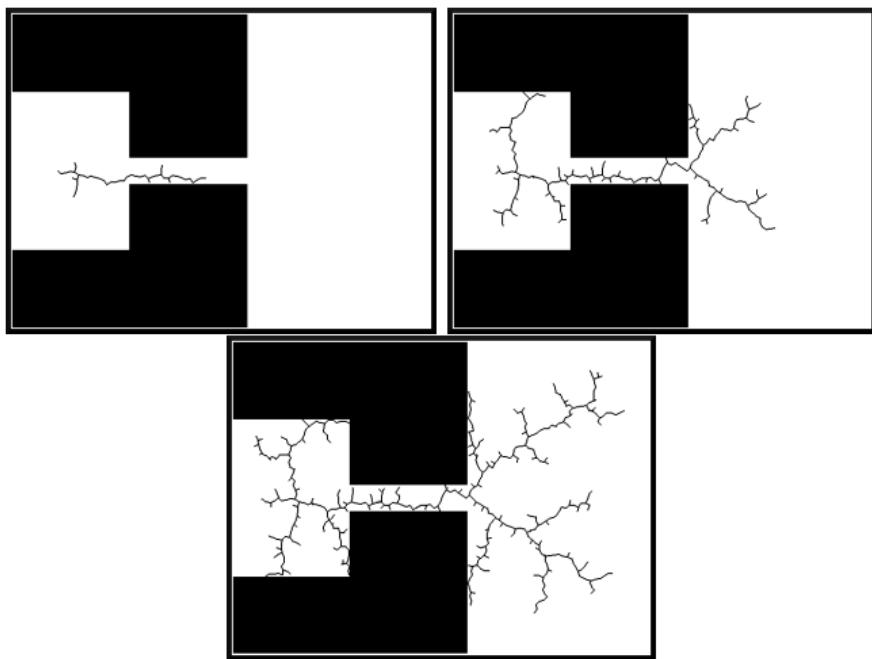
    Extiende(Árbol,  $q_r$ );

$k++$ ;

**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

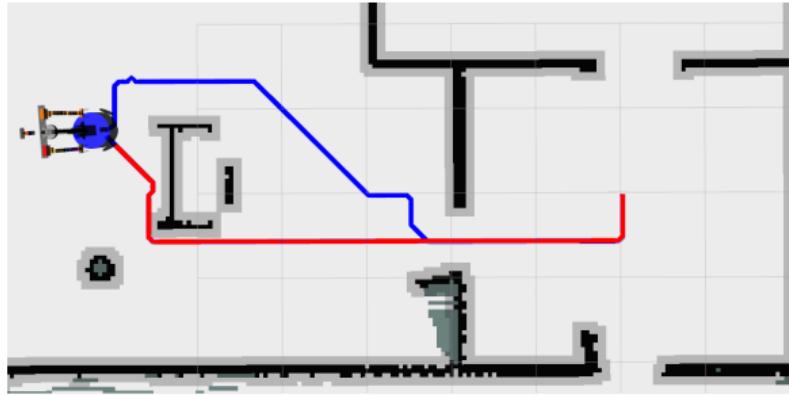
El criterio para seleccionar el siguiente nodo a expandir  $n_c$  de la lista abierta, determina el tipo de algoritmo:

- ▶ 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:

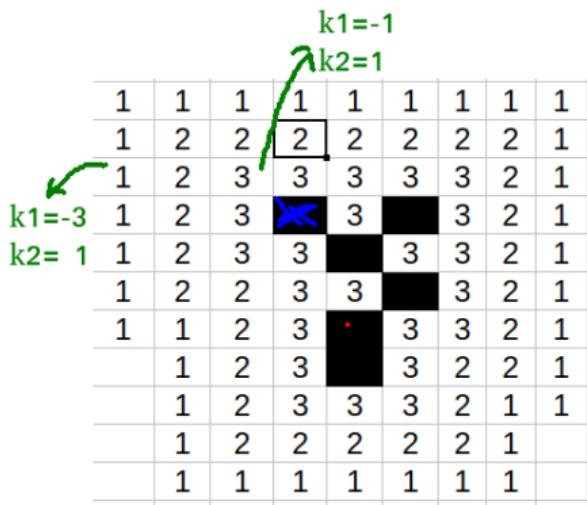
Mapa  $M$  de celdas de ocupación

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 obtenido anteriormente.
- ▶ 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:

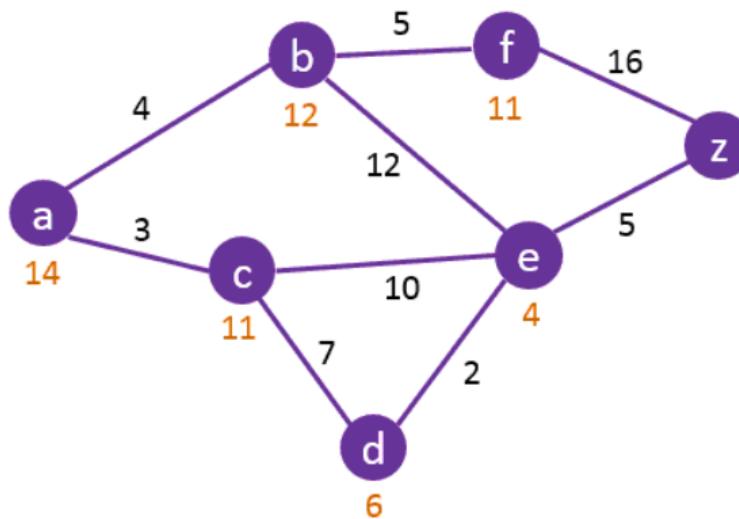
$$\begin{aligned}r &= \text{int}((y - M_{oy})/\delta) \\c &= \text{int}((x - M_{ox})/\delta)\end{aligned}$$

donde

- ▶  $(M_{ox}, M_{oy})$  es el origen del mapa, es decir, las coordenadas cartesianas de la celda  $(0,0)$ .
- ▶  $\delta$  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	<b>12, 16, D</b>	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 03 - Planeación de rutas

Realice lo siguiente:

1. Abra el archivo `catkin_ws/src/students/scripts/assignment03a.py` y agregue el siguiente código en la línea 45:

```
45 for i in range(height):
46     for j in range(width):
47         if static_map[i,j] > 50:
48             for k1 in range(-cost_radius, cost_radius+1):
49                 for k2 in range(-cost_radius, cost_radius+1):
50                     cost = cost_radius - max(abs(k1),abs(k2)) + 1
51                     cost_map[i+k1,j+k2] = max(cost, cost_map[i+k1,j+k2])
52
```

2. Corra el simulador con el comando

```
roslaunch bring_up path_planning.launch
```

3. Corra los nodos de inflado de mapas, mapa de costo y A\*, para ello, en tres terminales diferentes, corra los comandos:

```
rosrun students assignment02a.py _inflation_radius:=0.25
rosrun students assignment03a.py _cost_radius:=0.3
rosrun students assignment03b.py
```

## Ejercicio 03 - Planeación de rutas

4. Calcule una ruta utilizando los campos Start Pose y Goal Pose de la GUI:

The image shows a user interface for a robot navigation application. It consists of three rows of input fields and buttons. The first row has a label "Start Pose:" followed by a text input field containing "Robot" and a button labeled "Calc Path". The second row has a label "Goal Pose:" followed by a text input field containing "0 0" and a button labeled "Exec Path". The third row has a label "Simple Move:" followed by an empty text input field.

Start Pose:	<input type="text" value="Robot"/>	<button>Calc Path</button>
Goal Pose:	<input type="text" value="0 0"/>	<button>Exec Path</button>
Simple Move:	<input type="text"/>	

5. Detenga y ejecute de nuevo el inflado de mapas con radios entre 0.1 y 0.5 m y vea qué sucede con el cálculo de rutas.
6. Detenga y ejecute de nuevo el cálculo del mapa de costo con radios de entre 0.05 y 0.5 m y vea qué sucede con las rutas.
7. En el archivo `assignment03b.py`, cambie la función de distancia de Manhattan por distancia Euclídea, y cambie la conectividad 4 por conectividad 8 (líneas 44, 68 y 69) y vea qué sucede.
8. Modifique el código para que  $h$  sea siempre cero (línea 69) y vea qué sucede con el número de pasos. Pruebe con varias rutas.

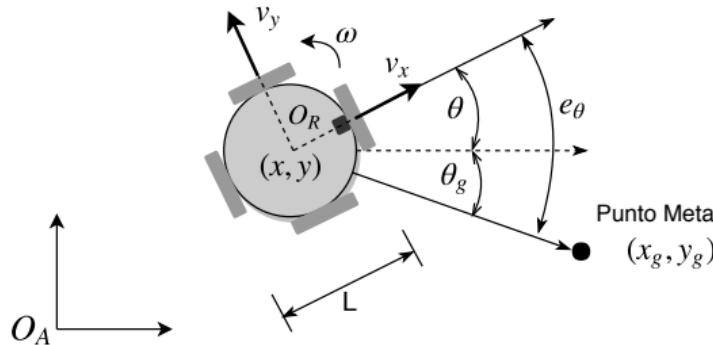
# Seguimiento de rutas

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  $q = (x, y, \theta)$ .



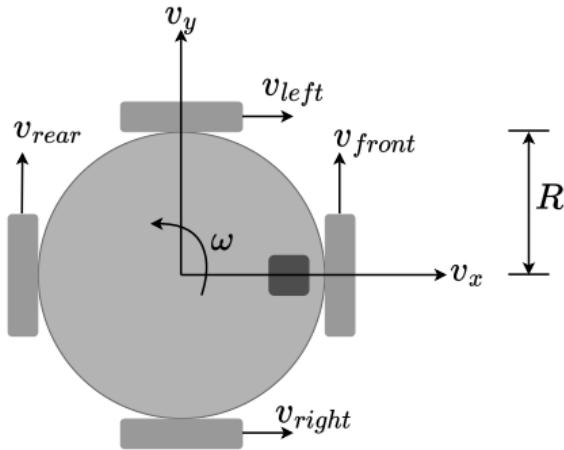
El modelo cinemático está dado por

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

- ▶  $(v_x, v_y, \omega)$  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  $(v_x, v_y, \omega)$  a velocidades de cada motor varía dependiendo del número de motores y de su posición.

## Base omnidireccional de 4 ruedas

Dada una base omnidireccional con las ruedas colocadas como se muestra en la figura, las velocidades de cada llanta se pueden obtener como:



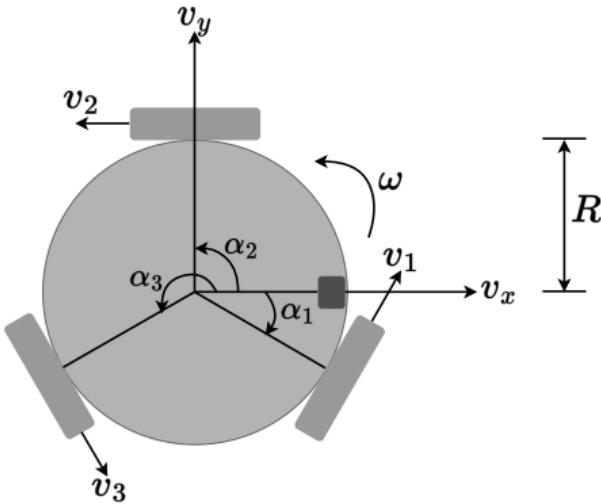
$$\begin{bmatrix} v_{left} \\ v_{right} \\ v_{front} \\ v_{rear} \end{bmatrix} = \begin{bmatrix} 1 & 0 & -R \\ 1 & 0 & +R \\ 0 & 1 & +R \\ 0 & 1 & -R \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}$$

- ▶ Las velocidades de las llantas son lineales.
- ▶ Para obtener las velocidades angulares, basta con dividir entre el radio de las llantas.

- ▶ Como se puede observar, la matriz anterior no tiene inversa.
- ▶ Se tienen cuatro velocidades de llantas en función de tres variables ( $v_x, v_y, \omega$ )
- ▶ Esto significa que dadas tres velocidades de llantas, **la cuarta no puede ser cualquier valor**

## Base omnidireccional de 3 ruedas

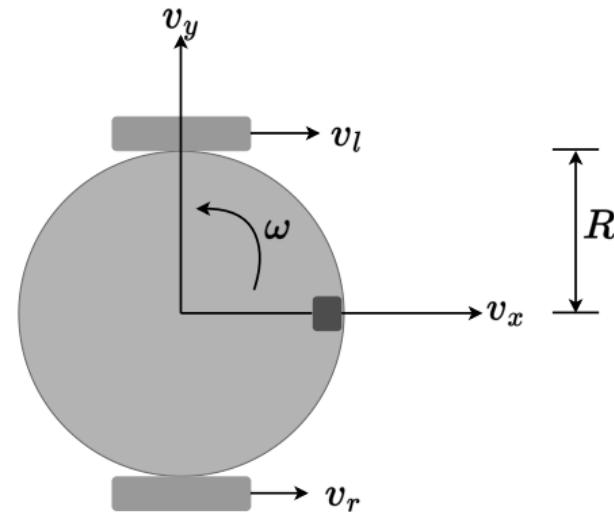
- ▶ La base de 4 ruedas omnidireccionales tiene la ventaja de tener mejor tracción y de lograr movimientos rectos más fácilmente.
- ▶ Tiene la desventaja de que las velocidades pueden indeterminarse si no están bien calculadas.
- ▶ Una base omnidireccional también puede lograrse con 3 ruedas:



$$\begin{bmatrix} v_1 \\ v_2 \\ v_3 \end{bmatrix} = \begin{bmatrix} -\sin \alpha_1 & \cos \alpha_1 & R \\ -\sin \alpha_2 & \cos \alpha_2 & R \\ -\sin \alpha_3 & \cos \alpha_3 & R \end{bmatrix} \begin{bmatrix} v_x \\ v_y \\ \omega \end{bmatrix}$$

# Base diferencial

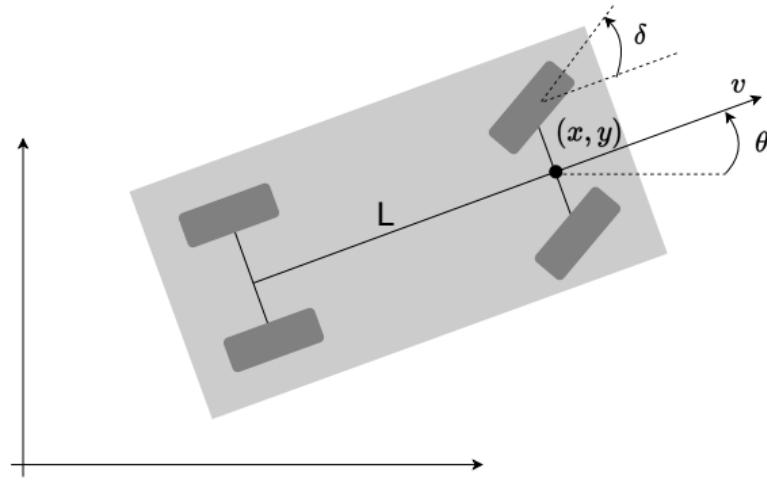
- ▶ Con una base diferencial ya no se tiene movimiento omnidireccional, es decir, se tiene movimiento no holonómico.
- ▶ El robot ya solo puede tener velocidad frontal  $v_x$  pero no velocidad lateral  $v_y$ .



$$v_l = v_x - R\omega$$

$$v_r = v_x + R\omega$$

Esta es la configuración más fácil de lograr debido a la simplicidad del hardware necesario.



$$\begin{aligned}\dot{x} &= v \cos(\theta + \delta) \\ \dot{y} &= v \sin(\theta + \delta) \\ \dot{\theta} &= \frac{v}{L} \sin(\delta)\end{aligned}$$

# 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

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

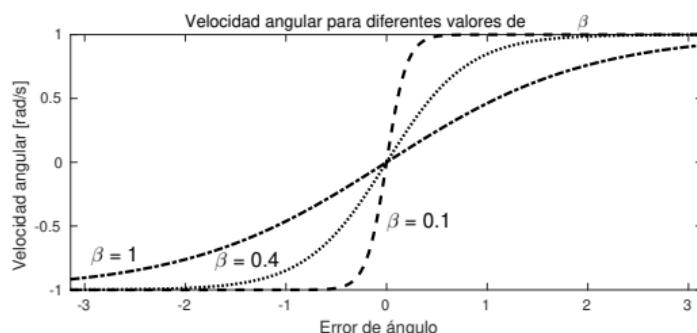
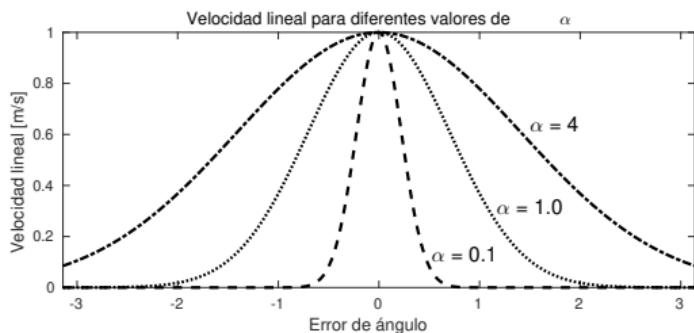
El error de ángulo  $e_\theta$  debe estar siempre en el intervalo  $(-\pi, \pi]$ . Si la diferencia resulta en un valor fuera de este ángulo, se puede acotar mediante:

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

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

# Control de posición

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



## Seguimiento de rutas

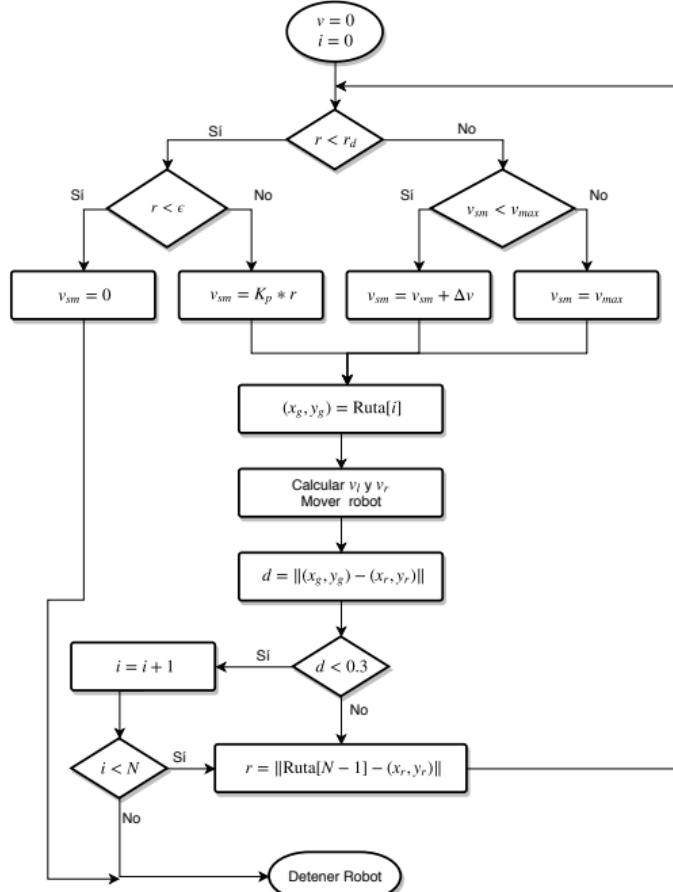
- ▶ Hasta el momento se ha planteado cómo alcanzar una posición, pero, ¿para una ruta?
- ▶ Las rutas son secuencias de puntos. Esta secuencia se podría parametrizar con respecto al tiempo para tener una trayectoria, sin embargo esto resulta muy complicado debido a la complejidad de las rutas.
- ▶ Una solución más sencilla es aplicar el control de posición para cada punto hasta recorrer toda la ruta.
- ▶ Las leyes de control solo dependen de  $e_\theta$  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  $\epsilon$ .

- ▶ En este caso, el robot se detendrá abruptamente cuando el error de distancia sea menor que  $\epsilon$ , 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:

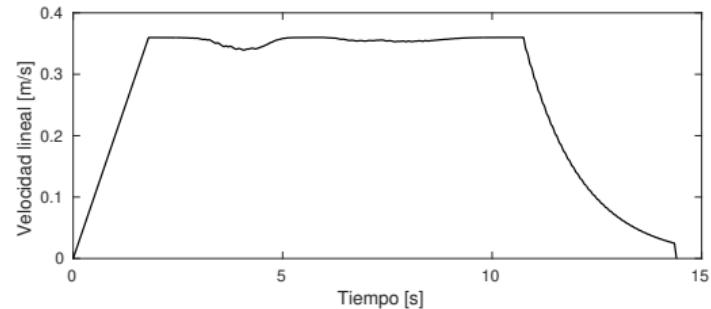
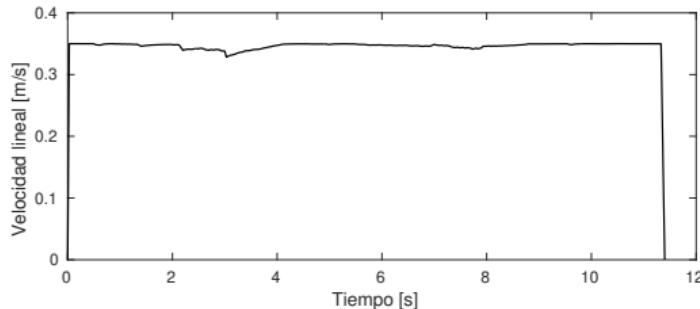
$$v = v_{sm} e^{-\frac{e^2}{\alpha}}$$
$$\omega = \omega_{max} \left( \frac{2}{1 + e^{-\frac{e^2}{\beta}}} - 1 \right)$$

con

- ▶  $r$ : Distancia a la meta global
- ▶  $\epsilon$ : Distancia a la que se considera que el robot alcanzó la meta global
- ▶  $r_d$ : Distancia a la meta global para desacelerar
- ▶  $\Delta v$ : 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).



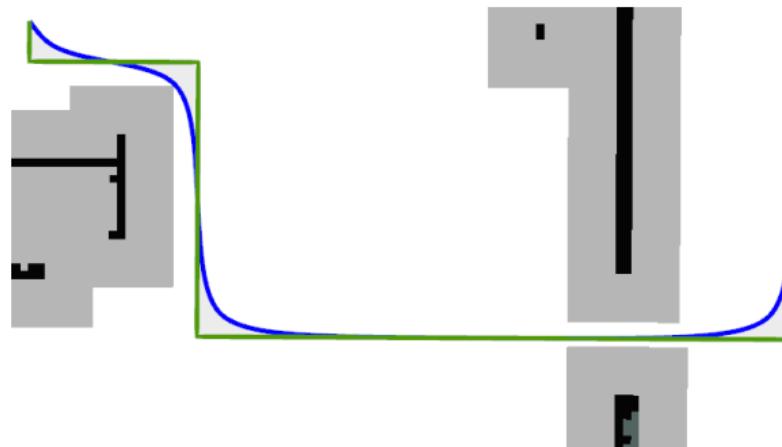
## Ejercicio 04 - Seguimiento de rutas

Realice lo siguiente:

1. Abra el archivo `catkin_ws/src/students/scripts/assignment05.py` e implemente las leyes de control para calcular  $v$  y  $\omega$  dentro de la función `calculate_speeds`. Siga las instrucciones de los comentarios del código.
2. Ejecute la simulación igual que en los ejercicios anteriores.
3. Ejecute el inflado de obstáculos, mapa de costo, algoritmo A\* y seguimiento de rutas (`assignment02a.py`, `assignment03a.py`, `assignment03b.py` y `assignment04.py`)
4. Con el botón *2D Nav Goal* del visualizador *RViz*, seleccione un punto meta en el mapa y observe qué sucede.
5. Pruebe con diferentes valores de  $\alpha$  y  $\beta$  hasta obtener un movimiento satisfactorio.

## Suavizado de rutas

- ▶ Puesto que las rutas se calcularon a partir de celdas de ocupación, están compuestas de esquinas.
- ▶ Las esquinas no son deseables, pues suelen generar cambios bruscos en las señales de control.
- ▶ La ruta verde de la imagen es una muestra de una ruta calculada por A\*.
- ▶ Es preferible una ruta como la azul.



Existen varias formas de suavizar la ruta generada:

- ▶ Splines
- ▶ Descenso del gradiente

## Suavizado mediante splines

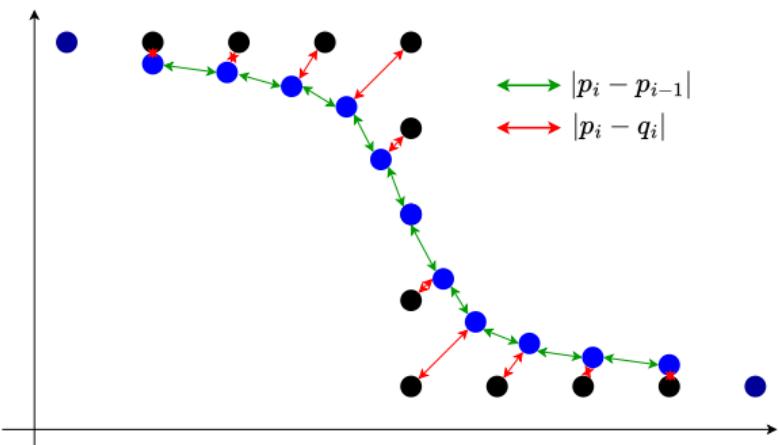
- ▶ Un *spline* es una función definida a tramos por polinomios.
- ▶ La forma más común son los splines de tercer grado o *cubic splines*
- ▶ Se ajusta un polinomio de tercer grado por cada par de puntos
- ▶ La derivada al final de un tramo debe ser igual a la derivada al inicio del siguiente tramo.
- ▶ Aplicando estas condiciones para cada par

# Suavizado mediante descenso del gradiente

Otra forma de suavizar la ruta es planteando una función de costo y encontrando el mínimo. Los puntos negros representan la ruta de A\* compuesta por los puntos  $Q = \{q_0, q_1, \dots, q_n\}$  y los puntos azules representan una ruta suave  $P = \{p_0, p_1, \dots, p_n\}$ .

Considere la función de costo:

$$J = \alpha \frac{1}{2} \sum_{i=1}^{n-1} (p_i - p_{i-1})^2 + \beta \frac{1}{2} \sum_{i=1}^{n-1} (p_i - q_i)^2$$



- ▶  $J$  es la suma de distancias entre un punto y otro de la ruta suavizada, y entre la ruta suavizada y la original.
- ▶ Si la ruta es muy suave,  $J$  es grande.
- ▶ Si la ruta es muy parecida a la original,  $J$  también es grande.
- ▶ Una ruta ni muy suave ni muy parecida a la original, logrará minimizar  $J$ .

# Suavizado mediante descenso del gradiente

- ▶ Una forma de encontrar el mínimo es resolviendo  $\nabla J(p) = 0$ , y luego evaluando la matriz Hessiana para determinar si el punto crítico  $p_c$  es un mínimo.
- ▶ Esto se puede complicar debido al alto número de variables en  $p$ .
- ▶ Una forma más sencilla, es mediante el descenso del gradiente.

---

## Algoritmo 6: Descenso del gradiente

---

**Data:** Función  $J(p) : \mathbb{R}^n \rightarrow \mathbb{R}$  a minimizar

**Result:** Vector  $p$  que minimiza la función  $J$

$p \leftarrow p_{init}$  //Fijar una estimación inicial

**while**  $|\nabla J(p)| > tol$  **do**

$| \quad p \leftarrow p - \epsilon \nabla J(p)$  // $p$  se modifica un poco en sentido contrario al gradiente.

**end**

Devolver  $p$

---

El descenso del gradiente devuelve el mínimo local más cercano a la condición inicial  $p_0$ . Pero la función de costo  $J$  tiene solo un mínimo global. El gradiente de la función de costo  $J$  se calcula como:

$$\left[ \underbrace{\alpha(p_0 - p_1) + \beta(p_0 - q_0), \dots, \alpha(2p_i - p_{i-1} - p_{i+1}) + \beta(p_i - q_i), \dots, \alpha(p_{n-1} - p_{n-2}) + \beta(p_{n-1} - q_{n-1})}_{\frac{\partial J}{\partial p_0}}, \underbrace{\dots, \frac{\partial J}{\partial p_i}, \dots, \frac{\partial J}{\partial p_{n-1}}} \right]$$

# Suavizado mediante descenso del gradiente

Para no variar los puntos inicial y final de la ruta, la primer y última componentes de  $\nabla J$  se dejarán en cero. El algoritmo de descenso del gradiente queda como:

---

## Algoritmo 7: Suavizado de rutas mediante descenso del gradiente

---

**Data:** Conjunto de puntos  $Q = \{q_0 \dots q_i \dots q_{n-1}\}$  de la ruta original, parámetros  $\alpha$  y  $\beta$ , ganancia  $\epsilon$  y tolerancia  $tol$

**Result:** Conjunto de puntos  $P = \{p_0 \dots p_i \dots p_{n-1}\}$  de la ruta suavizada

$P \leftarrow Q$

$\nabla J_0 \leftarrow 0$

$\nabla J_{n-1} \leftarrow 0$

**while**  $\|\nabla J(p_i)\| > tol$  **do**

**foreach**  $i \in [1, n - 1]$  **do**

$\nabla J_i \leftarrow \alpha(2p_i - p_{i-1} - p_{i+1}) + \beta(p_i - q_i)$

**end**

$P \leftarrow P - \epsilon \nabla J$

**end**

regresar  $P$

---

# Ejercicio 05 - Planeación y seguimiento de rutas

Realice lo siguiente:

1. Abra el archivo `catkin_ws/src/students/scripts/assignment05.py` y agregue el siguiente código en la línea 39:

```
39 nabla[0], nabla[-1] = 0, 0
40 while numpy.linalg.norm(nabla) > tol*len(P) and steps < 100000:
41     for i in range(1, len(Q)-1):
42         nabla[i] = alpha*(2*P[i] - P[i-1] - P[i+1]) + beta*(P[i] - Q[i])
43         P = P - epsilon*nabla
44         steps += 1
45
```

2. Corra la simulación igual en los ejercicios anteriores anteriores.
3. Corra los nodos de inflado de mapas, mapa de costo, A\* y seguimiento de rutas.
4. Corra el suavizado de rutas mediante el comando:  
`rosrun students assignment05.py _alpha:=0.9 _beta:=0.1`
5. En el visualizador, observe la diferencia entre la ruta original calculada con A\* (en azul) y la ruta suavizada (en verde).

## Ejercicio 05 - Planeación y seguimiento de rutas

6. Detenga el suavizado de rutas y vuelva a ejecutar con diferentes valores de  $\alpha$  y  $\beta$ .
7. Detenga el suavizado de rutas y sintonice las constantes  $\alpha$  y  $\beta$  del control de posición (ejercicio 04) hasta que el robot siga muy de cerca la ruta calculada (no importa que tenga cambios abruptos de velocidad).
8. Ejecute el suavizado de rutas y el control de posición con las constantes sintonizadas.
9. Comente los resultados.

## Evasión de obstáculos

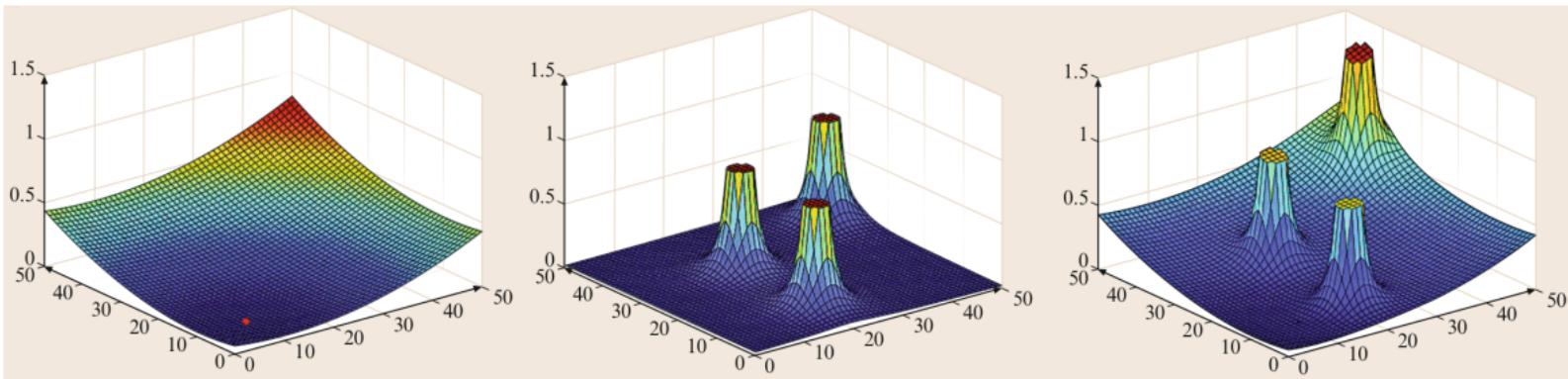
- ▶ Hasta el momento se tiene una manera de representar el ambiente, planear una ruta y seguirla
- ▶ ¿Qué pasa si en el ambiente hay un obstáculo que no estaba en el mapa?
- ▶ Se requiere de una técnica reactiva para evadir obstáculos
- ▶ Una posible solución es el uso de campos potenciales artificiales

# 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  $\nabla U$  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:

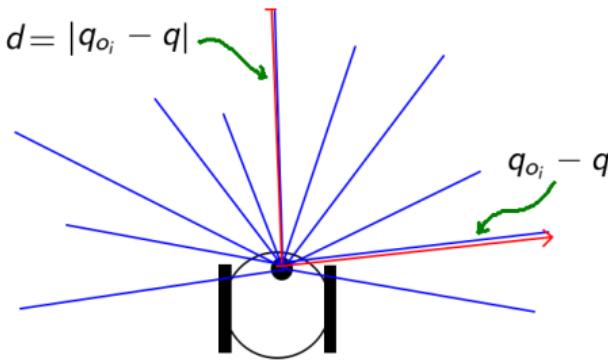
$$F_{att} = \zeta \frac{(q - q_g)}{\|q - q_g\|}, \quad \zeta > 0$$
$$F_{rej} = \begin{cases} \eta \left( \sqrt{\frac{1}{d} - \frac{1}{d_0}} \right) \frac{q_{o_i} - q}{d} & \text{si } d < d_0 \\ 0 & \text{en otro caso} \end{cases}$$

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
- ▶  $\zeta$  y  $\eta$ , 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  $(d_i, \theta_i)$  expresados con respecto al robot, por lo que, si se conoce la posición del robot  $(x_r, y_r, \theta_r)$ , 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 8:** 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  $tol$

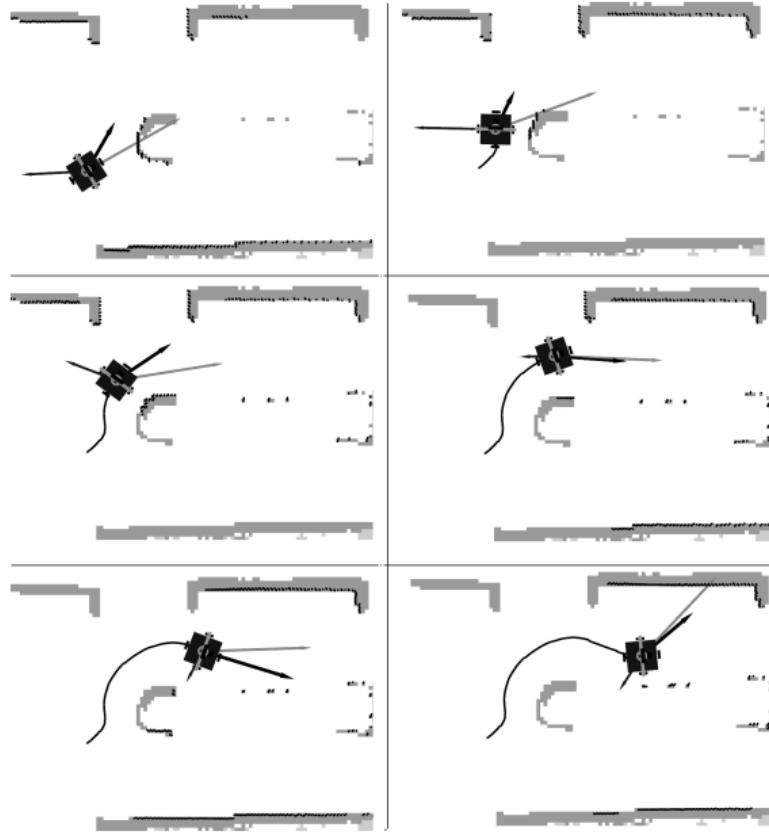
**Result:** Secuencia de puntos  $\{q_0, q_1, q_2, \dots\}$  para evadir obstáculos y alcanzar el punto meta

```
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 06 - Evasión de obstáculos

Realice lo siguiente:

1. Abra el archivo `catkin_ws/src/students/scripts/assignment06.py` y realice lo siguiente:
  - 1.1 En la función `calculate_control` implemente el control de posición utilizado en la práctica 03
  - 1.2 Implemente el cálculo de la fuerza atractiva en la función `attraction_force`
  - 1.3 En la función `rejection_force`, implemente el cálculo de la fuerza repulsiva total

$$F_{rej} = \frac{1}{N} \sum_{i=1}^N F_{rej_i}(q)$$

Considere que dada lectura del láser es un obstáculo, como se explicó anteriormente.

- 1.4 En la función `callback_pot_fields_goal` implemente el descenso del gradiente para mover el robot hacia el punto meta mediante campos potenciales artificiales. Revise los comentarios en el archivo.

2. Ejecute la simulación con el comando

```
roslaunch surge_et_ambula obstacle_avoidance.launch
```

3. Ejecute la evasión por campos potenciales mediante el comando

```
rosrun students assignment06.py
```

4. Con la opción *2D Nav Goal* del visualizador *RViz*, seleccione un punto meta en el mapa.

5. Observe qué sucede.

6. Pruebe con diferentes constantes de sintonización y observe los cambios 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.

# El Filtro de Kalman Extendido

# Algoritmo de estimación del EKF

# Tarea 07 - Filtro de Kalman Extendido

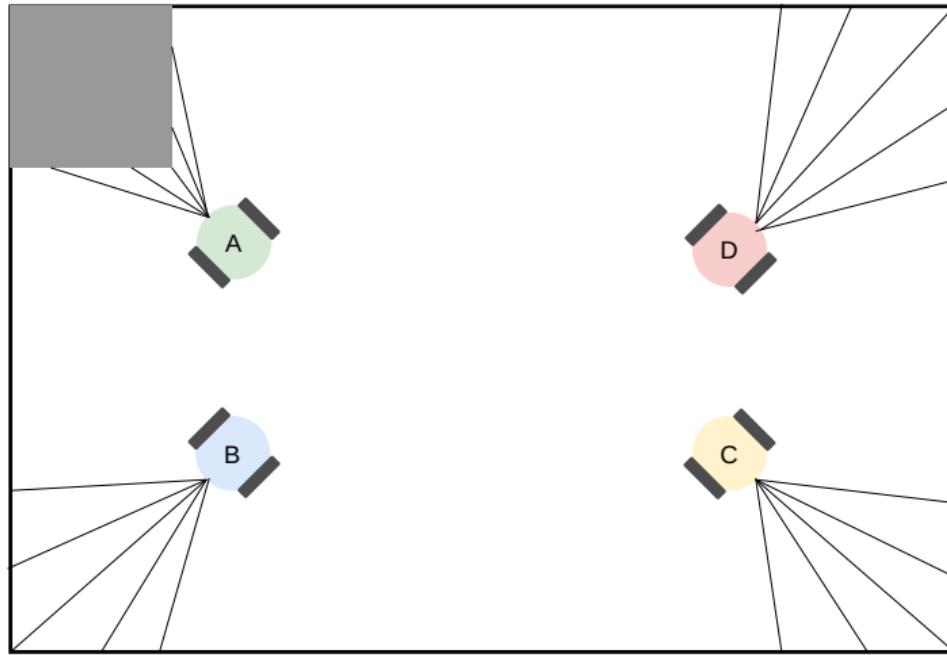
# Modelo cinemático discreto

# Modelo de observación

# Extracción de marcas

# Obtención de las matrices $Q$ y $R$

# Filtros de Partículas



## Tarea 08 - Conceptos de filtros de partículas

- ▶ Ejercicio de remuestreo con reemplazo
- ▶ Transformación de posición par desplazamiento de las partículas

# Simulación del sensor

# Comparación de simulación con real

# Remuestreo con reemplazo

# Desplazamiento de partículas

## Tarea 09 - Ejercicios de filtros de partículas

- ▶ Ejercicio de remuestreo con reemplazo
- ▶ Transformación de posición par desplazamiento de las partículas

# Repaso de C++

# La biblioteca *random\_numbers*

# Práctica 05 - Localización por filtros de partículas

Realice lo siguiente:

1. Abra el archivo `catkin_ws/src/students/src/practice05.cpp` y realice lo siguiente:
  - 1.1 En la función `get_initial_distribution` genere  $N$  partículas con pose  $(x, y, \theta)$  aleatoria con distribución uniforme.
  - 1.2 Complete la función `simulate_particle_scans` para generar lecturas del láser simuladas dado un mapa y la posición de las partículas.
  - 1.3 En la función `calculate_particle_similarities`, calcule la similitud vista en clase para cada partícula y normalice los valores para sumen 1.
  - 1.4 Complete la función `random_choice` para devolver un índice aleatorio  $i$  dada una distribución de probabilidad cualquiera determinada por un arreglo de probabilidades  $p$ .
  - 1.5 En la función `resample_particles` implemente el remuestreo con reemplazo visto en clase, dado un conjunto de partículas y una distribución de probabilidad.
  - 1.6 En la función `move_particles` implemente el movimiento de partículas dado un desplazamiento.
  - 1.7 Complete la función `main` de acuerdo con las instrucciones dadas en los comentarios del código.
2. Compile el código con el comando `catkin_make` (el directorio de trabajo debe ser `catkin_ws`)
3. Ejecute la simulación con el comando `roslaunch bring_up localization.launch`
4. Corra la práctica con el comando `rosrun students practice05`
5. Mueva el robot con la GUI y observe lo que sucede en el visualizador.
6. El código debe recompilarse con cada cambio.

# Práctica 05 - Localización por filtros de partículas

## Entregables:

- ▶ Código modificado en la rama correspondiente del repositorio en línea.
- ▶ Documento escrito con los puntos indicados al inicio del semestre

**Deadline:** 2023-05-09 al inicio de la clase.

# ¿Qué es la visión computacional?

- ▶ **Visión Humana:** Se puede concebir como una tarea de procesamiento de información, que obtiene significado a partir de los estímulos percibidos por los ojos.
- ▶ **Visión Computacional:** Desarrollo de programas de computadora que puedan *interpretar* imágenes. Es decir, realizar la visión humana por medios computacionales.



"Visión es un proceso que produce, a partir de imágenes del mundo externo, una descripción que es útil para el observador y que está libre de información irrelevante." (Marr, 1976).

El fenómeno de la visión lo podemos considerar como el producto de un sistema de procesamiento de información.

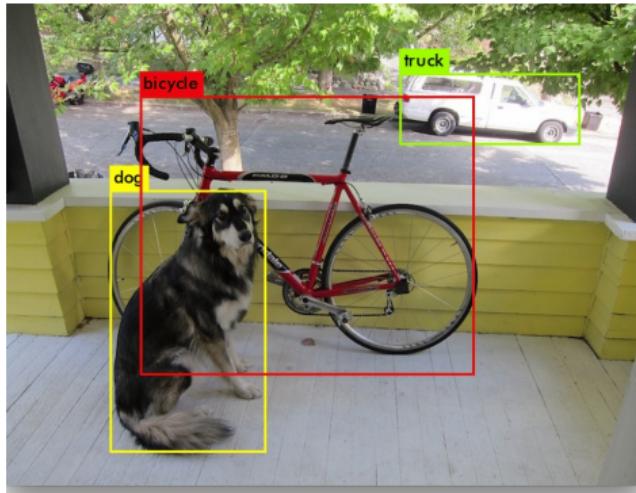
Marr propone los siguientes tres niveles de construcción de un sistema de procesamiento de información:

1. Teoría Computacional (¿Cuál es el problema por resolver?)
2. Representación y algoritmos (Estrategia usada para resolverlo)
3. Implementación (Realización física, software y hardware)

Es decir, la visión computacional sería un proceso parecido a la visión humana, similar en los niveles computacionales y de algoritmos, pero implementado de forma diferente: en hardware de procesamiento con sensores de visión.

# Visión Computacional

Por lo tanto, la tarea de la Visión por computadora es la construcción de descriptores de la escena con base en características relevantes contenidas en una imagen:



- ▶ Objetos
- ▶ Formas de Superficies
- ▶ Colores
- ▶ Texturas
- ▶ Movimientos
- ▶ Iluminación
- ▶ Reflejos

# Vision Computacional vs Proc de Imágenes

1. Procesamiento de imágenes: Es cualquier forma de procesamiento de señales donde la entrada es una imagen, la salida puede ser otra imagen o un conjunto de características o parámetros relacionados con la misma.
2. Visión Computacional: Estudio y aplicación de métodos que permiten a las computadoras “entender” el contenido de una imagen.
3. Visión Máquina: Es la aplicación de la visión por computadora en la industria y procesos de manufactura.

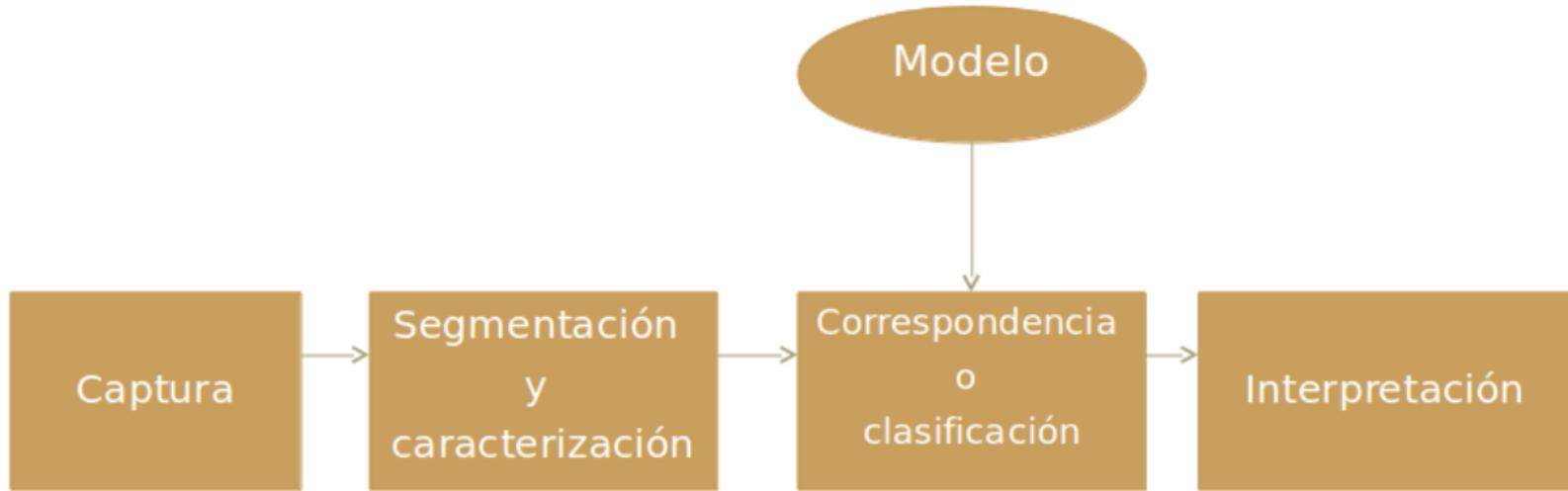
Tareas que se pueden hacer con visión computacional (con aplicaciones a la robótica):

- ▶ OCR (Optical Character Recognition)
- ▶ Detección e identificación de rostros
- ▶ Reconocimiento de objetos
- ▶ Percepción para vehículos sin conductor
- ▶ Reconocimiento de gestos

Otras aplicaciones:

- ▶ Vigilancia
- ▶ Imagenología médica
- ▶ Consultas a bases de datos de imágenes.
- ▶ Percepción remota

# Esquema de Visión



# Dificultades

El entorno real tiene una gran cantidad de variaciones en las imágenes de entrada.



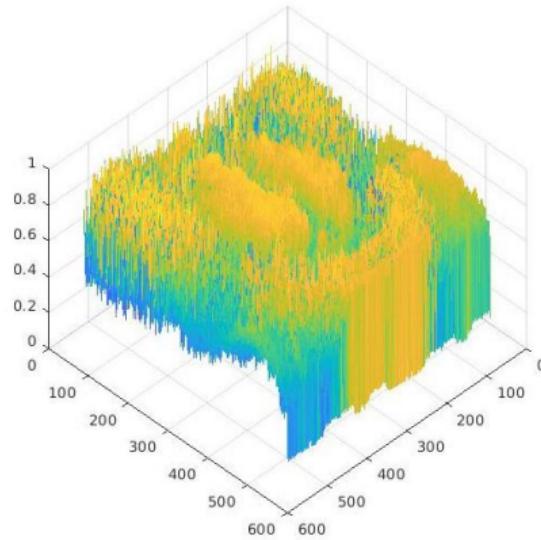
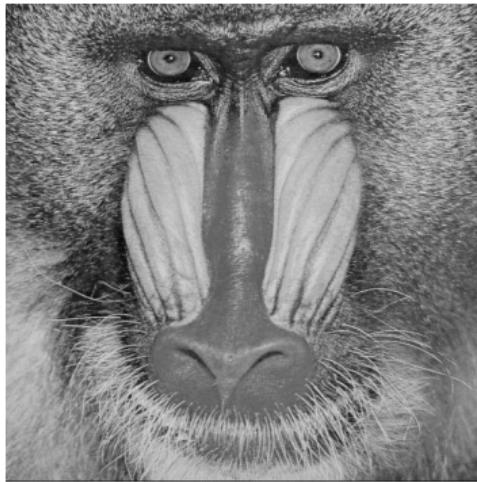
1. Iluminación
2. Orientación
3. Oclusión
4. Escala
5. Ruido
6. Desenfoque

La computación con imágenes tiene mas de 30 años, sin embargo, en los últimos años, se ha incrementado considerablemente su desarrollo debido a:

1. Decremento en los precios
2. Memoria con gran capacidad
3. Procesadores de propósito general de alta velocidad.
4. Existen scanners o camaras digitales que pueden ser utilizados para procesar imágenes propias.
5. Existen bibliotecas de software que contienen subrutinas de procesamiento de imágenes (opencv).

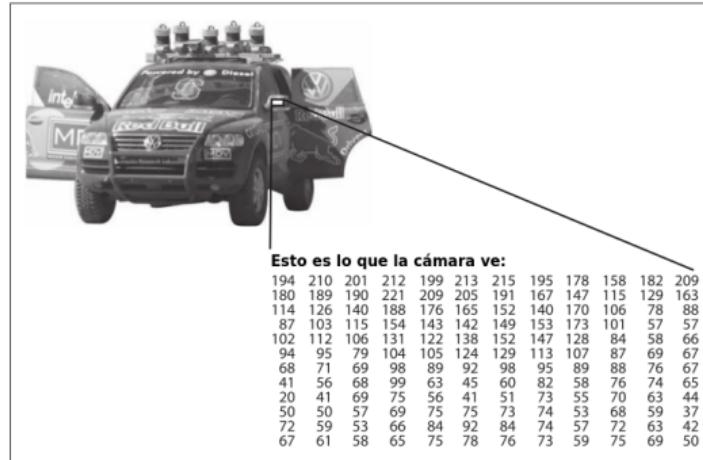
# Imágenes como funciones

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



# Las imágenes como funciones

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



# Operaciones básicas

- ▶ Desfase
- ▶ Escalamiento
- ▶ Inversión en  $x, y$
- ▶ Suma y Resta
- ▶ Multiplicación

Ver ejercicios en Python.

# Tipos de ruido

El ruido es una señal aleatoria  $\eta(x, y)$ , es decir, no sabemos cuánto vale para un punto determinado  $(x, y)$  pero sí podemos caracterizarla.

$$I_n(x, y) = I(x, y) + \eta(x, y)$$

Existen varios tipos de ruido:

- ▶ Sal y pimienta: aleatoriamente aparecen puntos ya sea blancos o negros
- ▶ Ruido de impulso: aleatoriamente aparecen puntos blancos
- ▶ Ruido gausiano:  $\eta(x, y)$  se distribuye

Ver ejercicios en Python

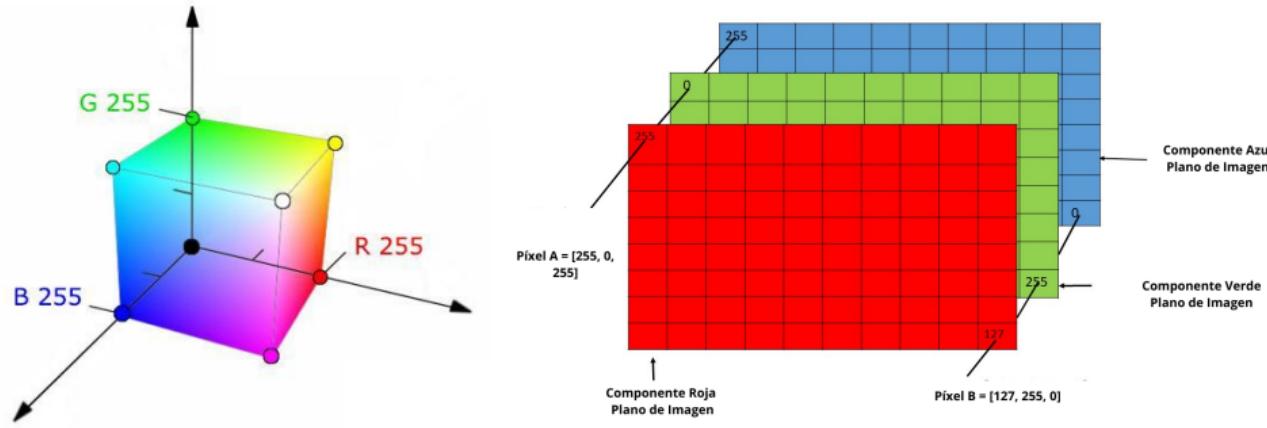
# Espacios de color

Un espacio de color o modelo de color es una representación del color mediante un conjunto numérico de valores, generalmente tres valores. Existen varios espacios de color:

- ▶ Aditivos:
  - ▶ RGB
  - ▶ HSV
  - ▶ YCrCb
- ▶ Sustractivos
  - ▶ MCYK
- ▶ Lineales:
  - ▶ RGB
  - ▶ CIE XYZ
- ▶ No lineales
  - ▶ HSV
  - ▶ HSI

# El espacio RGB

En este espacio cada color se forma mediante la suma de tres colores primarios: rojo, verde y azul.



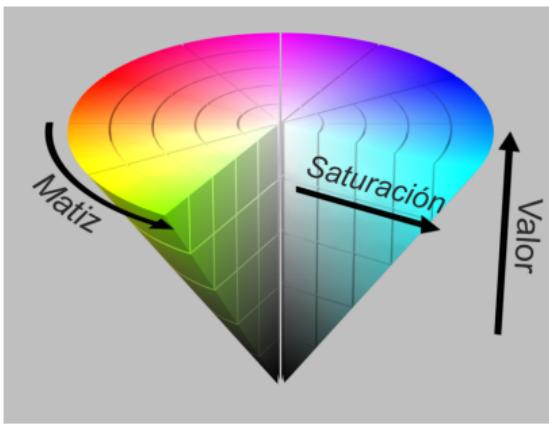
En memoria, las imágenes RGB se suelen representar con arreglos de  $H \times W \times 3$  donde  $H$  y  $W$  son el alto y ancho de la imagen respectivamente.

# El espacio HSV

Es un espacio de color diseñado para representar el color de una forma más similar a como lo percibe el ojo humano. El color se representa con tres valores:

- ▶ **Hue:** (Matiz) Es el atributo del color que hace que un estímulo se perciba como similar a alguno de los colores que el ojo puede percibir: rojo, amarillo, verde, azul, violeta, o una combinación de ellos.
- ▶ **Saturation:** (Saturación) Es el atributo que indica qué tan colorido es un estímulo con respecto a su propio brillo.
- ▶ **Value:** (Valor) Atributo que indica qué tanta luz emite un estímulo.

Para obtenerlo a partir de RGB:



$$M = \max(R, G, B) \quad m = \min(R, G, B) \quad C = M - m$$

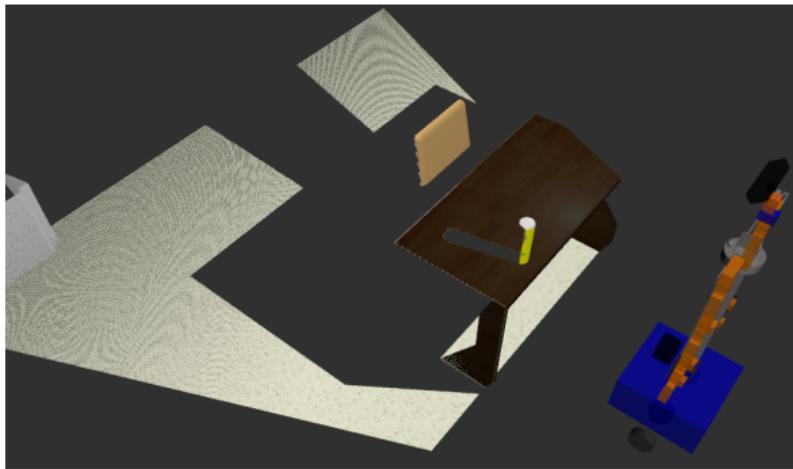
$$H = \begin{cases} \text{Indeterminado} & \text{si } C=0 \\ \frac{G-B}{C} \times 60 & \text{si } M = R \\ \frac{B-R}{C} \times 60 & \text{si } M = G \\ \frac{R-G}{C} \times 60 & \text{si } M = B \end{cases}$$

$$S = \begin{cases} 0 & \text{si } V = 0 \\ \frac{C}{V} & \text{en otro caso} \end{cases}$$

$$V = M$$

## 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. Obtención del centroide de la región (funciones findNonZero y mean)
4. Si se dispone de una nube de puntos, se puede obtener la posición ( $x, y, z$ ) del centroide de la región segmentada.

## Práctica 06 - Segmentación por color

1. En el archivo `catkin_ws/src/students/scripts/practice06.py`, en la función `segment_by_color`, realice lo siguiente:
  - 1.1 Defina dos límites superiores y dos límites inferiores, en el espacio de color HSV, para segmentar las latas que se encuentran sobre el escritorio simulado.
  - 1.2 Transforme la imagen del espacio BGR al espacio HSV mediante la función `cvtColor` de OpenCV.
  - 1.3 Determine los pixeles de la imagen que pertenecen al rango de color elegido mediante la función `inRange` de OpenCV.
  - 1.4 Encuentre los índices de los pixeles que pertenecen al rango de color con la función `findNonZero` de OpenCV.
  - 1.5 Utilizando los índices anteriores y la nube de puntos, determine el centroide del objeto con el color seleccionado.
2. Ejecute la simulación con el comando `roslaunch bring_up color_segmentation.launch`
3. Ejecute la práctica con el comando `rosrun students practice06.py`
4. En la pestaña *Simple Tasks* de la GUI, en el campo *Find Object* teclee `pringles` o `drink`
5. En el visualizador debe dibujarse una esfera morada sobre el centro del objeto seleccionado.

# Práctica 06 - Segmentación por color

## Entregables:

- ▶ Código modificado en la rama correspondiente del repositorio en línea.
- ▶ Documento escrito con los puntos indicados al inicio del semestre

**Deadline:** 2023-05-16 al inicio de la clase.

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.

# 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:

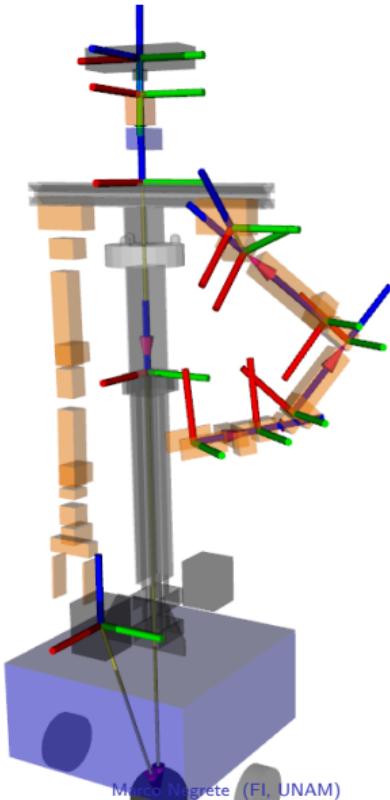
- ▶ Asociatividad:  $(T_1 T_2) T_3 = T_1 (T_2 T_3)$
- ▶ Inversa:

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

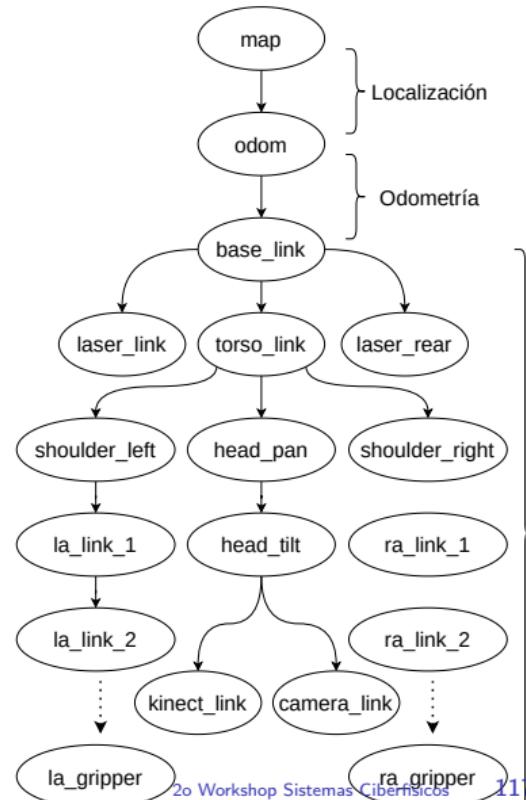
- ▶ Cancelación de índices:  $T_{ab} = T_{ac} T_{cb}$

# 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   <!-- Define la forma y tamano de la base movil-->
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 tamano 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 posicion del laser con respecto a la base movil-->
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

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>
```

## Tarea 12 - Transformaciones Homogéneas

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. Despues ejecute el comando:

```
1 rosrun bringup 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.

- ▶ Observe en el visualizador qué sucede con las lecturas del sensor láser.
- ▶ Deshaga los cambios.

## 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”.

# Tarea 13 - Cinemática directa

# 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

$$f(x) = 0$$

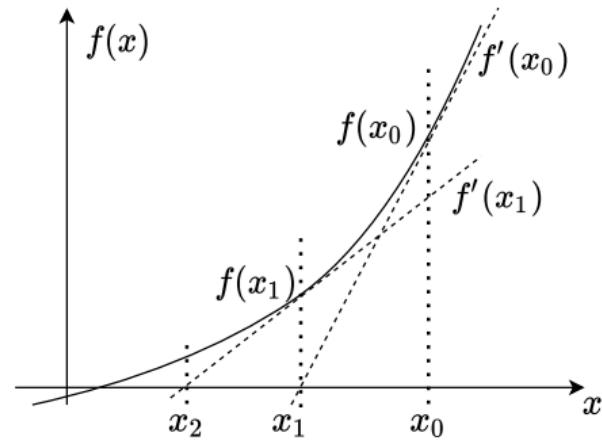
El algoritmo es el siguiente:

---

---

```
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  $\Delta q$ , 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  $p - p_d$

# Práctica 10 - Cinemática Inversa

# Modelo dinámico de un manipulador

# Tarea 14 - Modelo dinámico del brazo

# 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  $\tau$  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 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.

# Práctica 11 - Control del manipulador

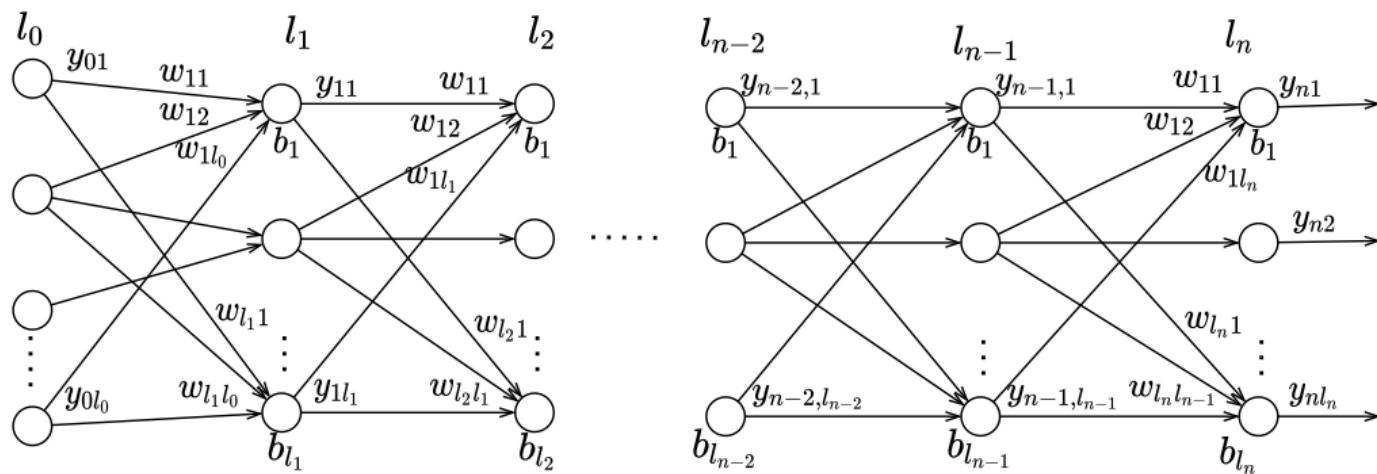
# Modelo de una neurona

# Entrenamiento de una neurona

# Tarea 15 - Entrenamiento de una neurona

# Backpropagation

La red neuronal está definida por la cantidad de neuronas en cada capa  $[l_0, l_1, \dots, l_{n-1}, l_n]$ :



$$W_1 \in R^{l_1 \times l_0}$$

$$W_2 \in R^{l_2 \times l_1}$$

$$B_1 \in R^{l_1}$$

$$B_2 \in R^{l_2}$$

$$W_n \in R^{l_{n-1} \times l_{n-2}} \quad W_n \in R^{l_n \times l_{n-1}}$$

$$B_n \in R^{l_{n-1}}$$

$$B_n \in R^{l_n}$$

El conjunto de pesos  $w$  se puede agrupar en un conjunto de  $n$  matrices  $W = [W_1, W_2, \dots, W_n]$  con los órdenes indicados en la figura. El conjunto de biases se puede agrupar en  $n$  vectores  $B = [B_1, B_2, \dots, B_n]$ .

# Backpropagation

Para la capa salida, el gradiente con respecto a cada uno de los pesos  $w \in W_n \in \mathbb{R}^{l_n \times l_{n-1}}$  es también una matriz  $\nabla y_n \in \mathbb{R}^{l_n \times l_{n-1}}$ :

$$\begin{bmatrix} (y_{n1} - t_1)(y_{n1} - y_{n1}^2)y_{n-1,1} & (y_{n1} - t_1)(y_{n1} - y_{n1}^2)y_{n-1,2} & \dots & (y_{n1} - t_1)(y_{n1} - y_{n1}^2)y_{n-1,l_{n-1}} \\ (y_{n2} - t_2)(y_{n2} - y_{n2}^2)y_{n-1,1} & (y_{n2} - t_2)(y_{n2} - y_{n2}^2)y_{n-1,2} & \dots & (y_{n2} - t_2)(y_{n2} - y_{n2}^2)y_{n-1,l_{n-1}} \end{bmatrix}$$

# Práctica 12 - Redes neuronales

# La biblioteca Festival

# La biblioteca Sphinx

# Tarea 16 - Reconocimiento de voz

# Proyecto final: robot de servicio

## Contacto

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

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