Introducción a la navegación autónoma para robots móviles

Marco Negrete

Facultad de Ingeniería, UNAM

Escuela de Invierno de Robótica 2024-2025, CIC, IPN
https://github.com/RobotJustina/pumas_navigation

Contenido

Algunos conceptos previos

- **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 espacion 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.

Algunos conceptos previos

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 cuándo 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 exite, 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.

Planeación de movimientos

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

- Mapeo: construir una representación del ambiente a partir de las lecuras de los sensores y la trayectoria del robot.
- 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}.
- Localización: determinar la configuración q dado un mapa y lecturas de los sensores.
- Barrido: pasar un actuador por todos los puntos $q \in Q_b \subset Q$.

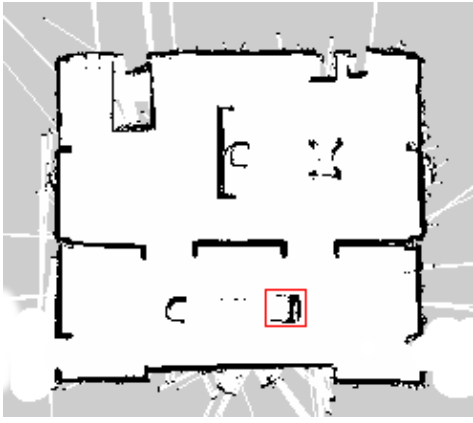
Representación del ambiente

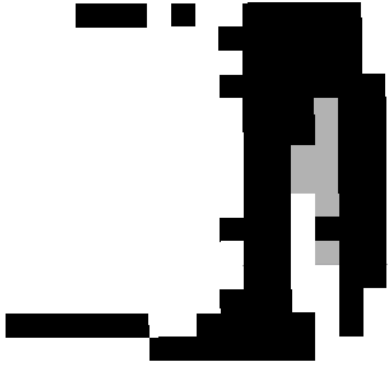
Un mapa es cualquier representación del ambiente útil en la toma de decisiones.

- 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

- Cree un espacio de trabajo para ROS y clone los repositorios:
  - `git clone https://github.com/RobotJustina/pumas_hardware`
  - `git clone https://github.com/RobotJustina/pumas_navigation`
- Compile el espacio de trabajo
- Ejecute el comando

```
1        roslaunch hardware_bring_up justina_hardware_gazebo.launch
2
```

- En otra terminal, ejecute el comando:

```
1        roslaunch navigation_bring_up navigation.launch
2
```

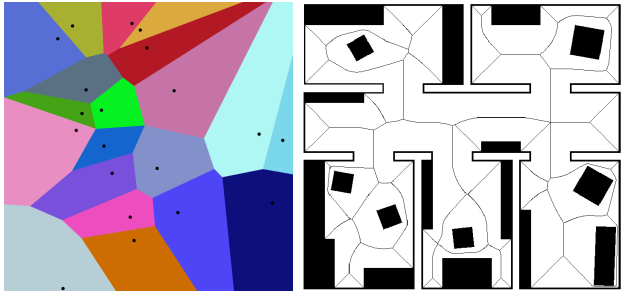
- En otra terminal, ejecute el comando:

```
1      rviz -d src/pumas_navigation/navigation_bring_up/rviz/pumas_navigation
       .rviz
2
```

- Observe el mapa de celdas de ocupación
- Opcional: Modifique que archivo `navigation_bring_up/maps/appartment.pgm`

Diagrama de Voronoi Generalizado

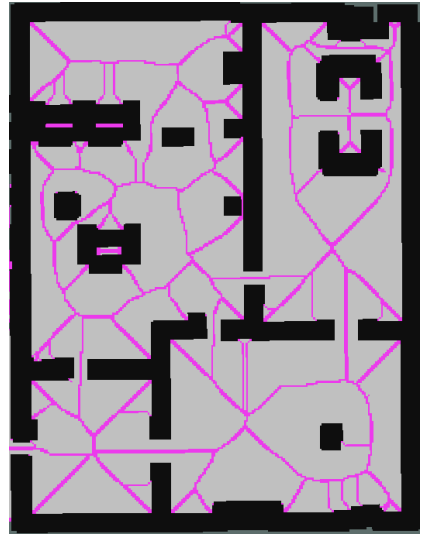
- A diferencia de los mapas geométricos, donde se busca reflejar la forma exacta del ambiente, los **mapas topológicos** buscan representar solo las relaciones espaciales de los puntos de interés.
- Los Diagramas de Voronoi dividen el espacio en regiones. Cada región está asociada a un punto llamado 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 generalizdo (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 un espacio finito y discretizado (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.



Ejercicio 2

- Ejecute el comando

```
roslaunch hardware_bring_up justina_hardware_gazebo.launch
```

- En otra terminal, ejecute el comando:

```
roslaunch navigation_bring_up navigation.launch
```

- En otra terminal, ejecute el comando:

```
rviz -d src/pumas_navigation/navigation_bring_up/rviz/pumas_navigation.rviz
```

- En otra terminal, ejecute el comando:

```
rosrun map_augmenter gvd.py
```

Planeación de rutas

La planeación de rutas consiste en encontrar una secuencia de configuraciones $q \in Q_{free}$ 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

Métodos basados muestreo

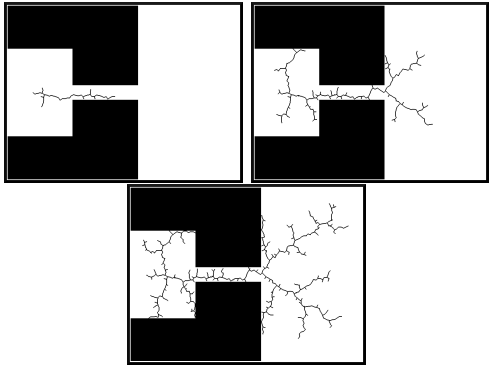
Como su nombre lo indica, consisten en tomar muestras aleatorias del espacio libre. Si es posible llegar en línea recta de la configuración actual al punto muestrado, entonces se agrega a la ruta. Ejemplos:

- 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. Los pasos generales se pueden resumir en:

1. Construir un árbol T con nodo raíz en el punto inicial
2. Elegir una posición aleatoria $p = (x, y)$ dentro del espacio libre
3. Encontrar el nodo n, del árbol T, más cercano al punto p
4. Si la distancia $d(n, p) > \epsilon$, cambiar p por un punto en la misma dirección $\overrightarrow{np}$ pero a una distancia ϵ
5. Si no hay colisión entre n y p, agregar a p como nodo hijo de n
6. Si no hay colisión entre p y el punto meta p_g, agregar p_g como nodo hijo de p, terminar la exploración y anunciar éxito
7. Repertir desde el punto 2 un máximo de N veces.



Ejercicio 3

- Ejecute el comando

```
roslaunch hardware_bring_up justina_hardware_gazebo.launch
```

- En otra terminal, ejecute el comando:

```
roslaunch navigation_bring_up navigation.launch use_rrt:=true
```

- En otra terminal, ejecute el comando:

```
rviz -d src/pumas_navigation/navigation_bring_up/rviz/pumas_navigation.rviz
```

- Con RViz fije un punto meta

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 algorimos se pueden resumir en:

Data: Mapa M de celdas de ocupación, configuración inicial q_{start}, configuración meta q_{goal}
Result: Ruta $P = [q_{start}, q_1, q_2, \ldots, q_{goal}]$
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

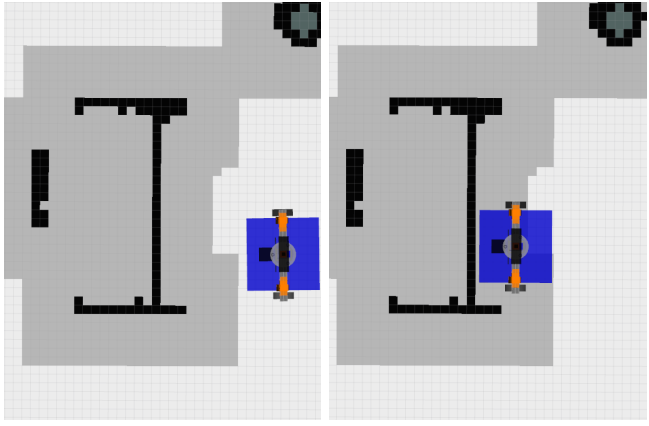
El criterio para seleccionar el siguiente nodo a expandir n_c de la lista abierta, determina el tipo de algoritmo:

- 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 (óptima) pero A* lo hace más rápido.

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:

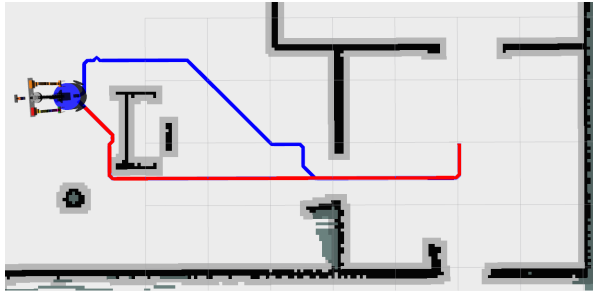


- Celdas blancas: espacio libre.
- Celdas negras: espacio con obstáculos.
- Celdas grises: espacio sin obstáculos donde el robot no puede estar debido a su tamaño.

- Un mapa de celdas de ocupación debe *inflarse* antes de usarse para planear rutas.
- Esta operación se conoce como *dilatación* y es tipo de *operador morfológico*
- El inflado se usa para planeación de rutas, no para localización.

Mapas de costo

- Los métodos como Dijkstra y A* minimizan una función de costo. Esta función podría ser distancia, tiempo de recorrido, número de giros, 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.
- Ejemplo: la ruta azul es más larga pero más segura.



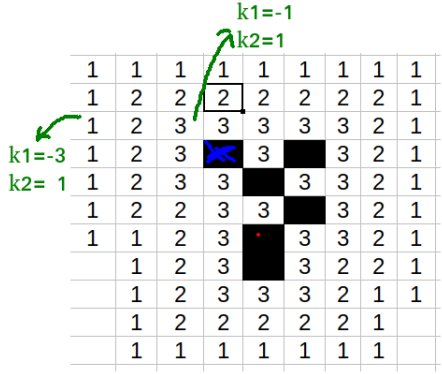
Mapas de costo

- 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 1: 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 $i \in [0, \ldots, rows)$ **do**
  foreach $j \in [0, \ldots, cols)$ **do**
    //Si está ocupada, calcular el costo de r_c celdas alrededor.
    if $M[i,j] == 100$ **then**
      foreach $k_1 \in [-r_c, \ldots, r_c]$ **do**
        foreach $k_2 \in [-r_c, \ldots, r_c]$ **do**
          $C = r_c - max(|k1|, |k2|) + 1$
          $M_c[i + k1, j + k2] = max(C, M_c[i + k1, j + k2]$
        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 $P = [q_s, q_1, q_2, \ldots, q_g]$
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 Euclideana.
- 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.

Ejercicio 4

- Ejecute el comando

```
roslaunch hardware_bring_up justina_hardware_gazebo.launch
```

- En otra terminal, ejecute el comando:

```
roslaunch navigation_bring_up navigation.launch
```

- En otra terminal, ejecute el comando:

```
rviz -d src/pumas_navigation/navigation_bring_up/rviz/pumas_navigation.rviz
```

- Detenga la navegación y cambien las constantes α y β

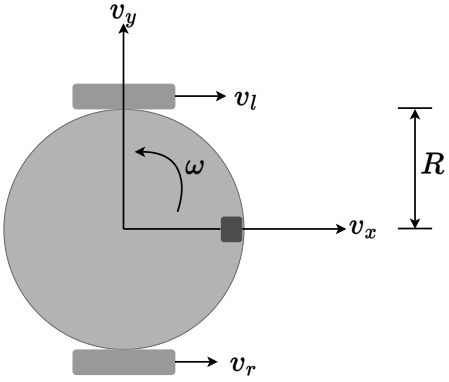
Seguimiento de rutas

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 abodará 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

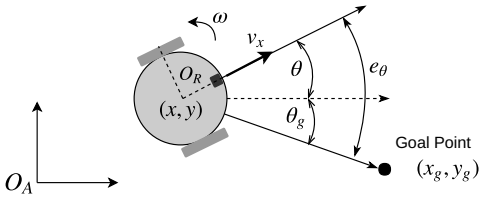
Base diferencial

- Es la configuración más sencilla debido a la simplicidad del hardware requerido
- Aunque implica restricciones no holonómicas, es suficiente para la mayoría de los movimientos requeridos. Además, suponer este tipo de base es algo útil debido a una razón sencilla: los sensores suelen estar al frente del robot.
- Se diseñarán leyes de control suponiendo esta configuración.



$$
\begin{aligned}
v_l &= v_x - R\omega \\
v_r &= v_x + R\omega
\end{aligned}
$$

Control de posición



Considere una base diferencial y el punto meta (x_g, y_g), las siguientes leyes de control permiten al robot 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

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

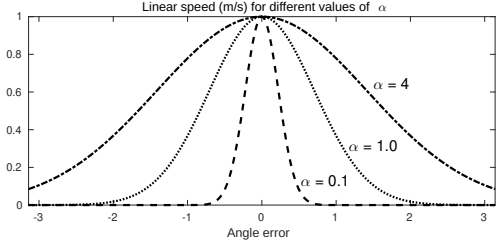
El error de ángulo e_θ debe estar siempre en $(-\pi, \pi]$. Esto se puede asegurar con:

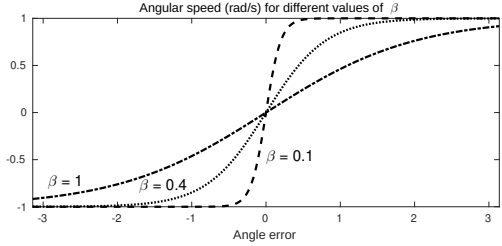
$$e_\theta \leftarrow \left(e_\theta + \pi\right) \%(2\pi) - \pi$$

donde % denota el operador módulo.

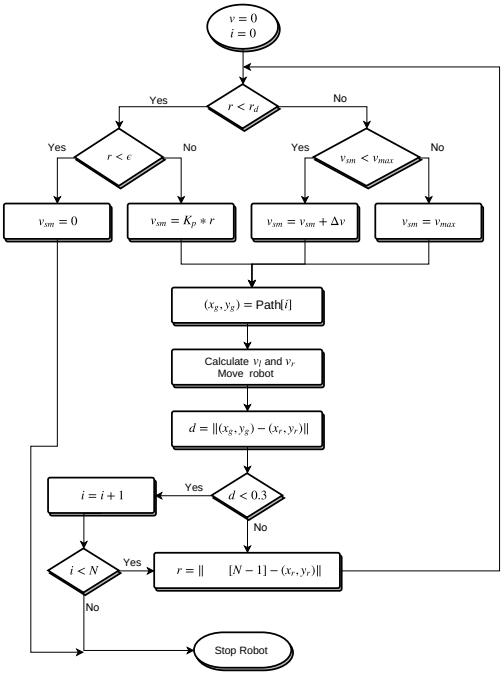
Control de posición

- v_{max} y ω_{max} son las velocides lineal y angular máximas.
- α y β determinan qué tan rápido cambian las velocidades cuando cambia el error de posición.
- En general, valores pequeños de α y β hacen que el robot siga la ruta más de cerca pero valores muy pequeños pueden producir *chattering*.
- Valores grandes de α y β producen un movimiento más suave pero puede resultar en curvas muy extendidas.





Perfil de velocidad



Considere una máquina de estados finita para calcular v_{max}. Sea v_{sm} la nueva velocidad lineal máxima, tal que, ahora el control está dado por:

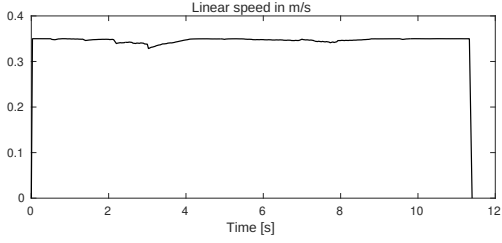
$$
\begin{aligned}
v &= v_{sm} e^{-\frac{e_\theta^2}{\alpha}} \\
\omega &= \omega_{max} \left(\frac{2}{1 + e^{-\frac{e_\theta}{\beta}}} - 1 \right)
\end{aligned}
$$

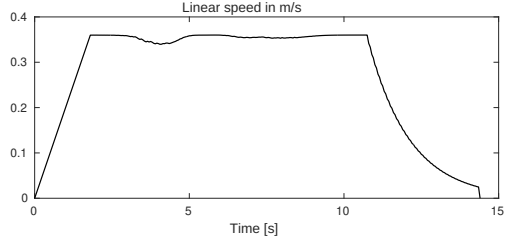
con

- r: Distancia la punto meta
- ϵ: Tolerancia para considerar que se alcanzó el punto meta
- r_d: Distancia al punto meta para empezar a desacelerar
- Δv: Aceleración deseada

Perfil de velocidad

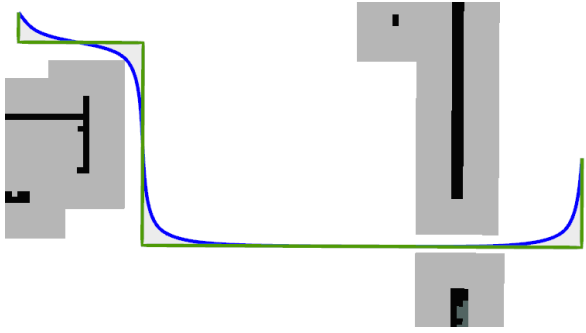
La figura muestra un ejemplo de una ruta planeada y el perfil de velocidad generada usando solo las leyes de control y usando la AFSM para generar un perfil de velocidad.





Suavizado de rutas

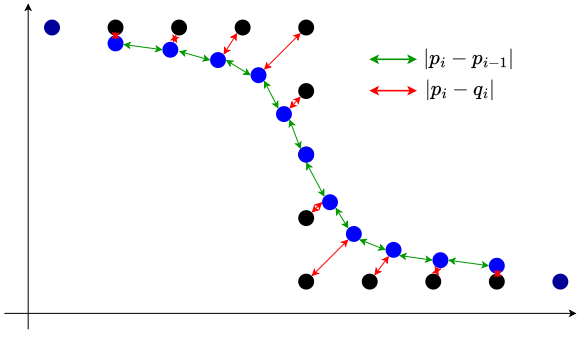
- Puesto que las rutas se calcularon a partir de celdas de ocupación, están compuestas de esquinas.
- La esquinas no son deseables, pues suelen generar cambios bruscos en las señales de control.
- La ruta verde de la imagen es una muestra de una ruta calculada por A*.
- Es preferible una ruta como la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



Usaremos un enfoque de optimización para suavizar la ruta generada por A*

Suavizado mediante descenso del gradiente

Se plantea una función de costo tal que, minimizar dicha función, resulte en una ruta suavizada. Los puntos negros representan la ruta de A* compuesta por los puntos $Q = \{q_0, q_1, \ldots, q_n\}$ y los puntos azules representan una ruta suave $P = \{p_0, p_1, \ldots, p_n\}$.



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 2: Descenso del gradiente

Data: Función $J(p) : \mathbb{R}^n \to \mathbb{R}$ a minimizar
Result: Vector p que minimiza la función J
$p \leftarrow p_{init}$ //Fijar una estimación inicial
while $|\nabla J(p)| > tol$ **do**
  $p \leftarrow p - \epsilon \nabla J(p)$ //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:

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

Suavizado mediante descenso del gradiente

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

Algoritmo 3: 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 α y β, ganancia ϵ 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 $\|\nabla J(p_i)\| > tol \wedge steps \text{ ¡ } max\_steps$ **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

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

  $steps \leftarrow steps + 1$

end

regresar P

Localización

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.

Localización

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.

Localización probabilística

- En la localización probabilística, en lugar de llevar una sola estimación 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

- Es un tipo de filtro Bayesiano que supone que la distribución de probabilidad de la posición del robot es una distribución normal:

$$x_{k+1} = f(x_k, u_k) + n_p$$

con:

  - $x_{k+1} = f(x_k, u_k)$: modelo de movimiento del robot (indica la media de la posición, que se distribuye normalmente)
  - n_p : ruido de proceso, que se distribuye normalmente con media cero y covarianza conocida Q

- Supone que las mediciones son varibles aleatorias también con una distribución normal

$$y_k = h(x_k) + n_m$$

con:

  - $y_k = h(x_k)$: medición esperada en función de la posición del robot
  - n_p : ruido de medición, que se distribuye normalmente con media cero y covarianza conocida R

- Si el sistema es lineal, el Filtro de Kalman converge para cualquier condición inicial
- Si el sistema es no lineal (el caso del robot), el Filtro de Kalman converge solo si la estimación inicial es *cercana* a la posición inicial real.
- El Filtro de Kalman es un método de localización local

El Filtro de Kalman Extendido

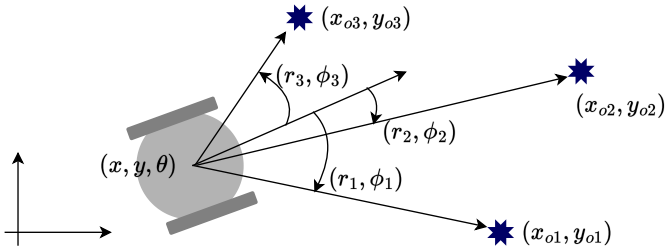
- El EKF pondera dos fuentes de información:

$$x_{k+1} = f(x_k, u_k) + n_p$$

$$y_k = h(x_k) + n_m$$

  - La posición que predice el modelo de movimiento
  - La posición que mide el sensor
- Cada fuente de información tiene una incertidumbre caracterizada por las covarianzas Q y R
- El EKF pondera ambas fuentes dándole más peso a la fuente con menos incertidumbre

El Filtro de Kalman Extendido



Modelo de transición de estados:

$$\left.\begin{array}{rcll} x_{k+1} & = & x_k + (\Delta t) v \cos(\theta_k) & +n_{p1} \\ y_{k+1} & = & y_k + (\Delta t) v \sin(\theta_k) & +n_{p2} \\ \theta_{k+1} & = & \theta_k + (\Delta t)\omega & +n_{p3} \end{array}\right\} f(x)$$

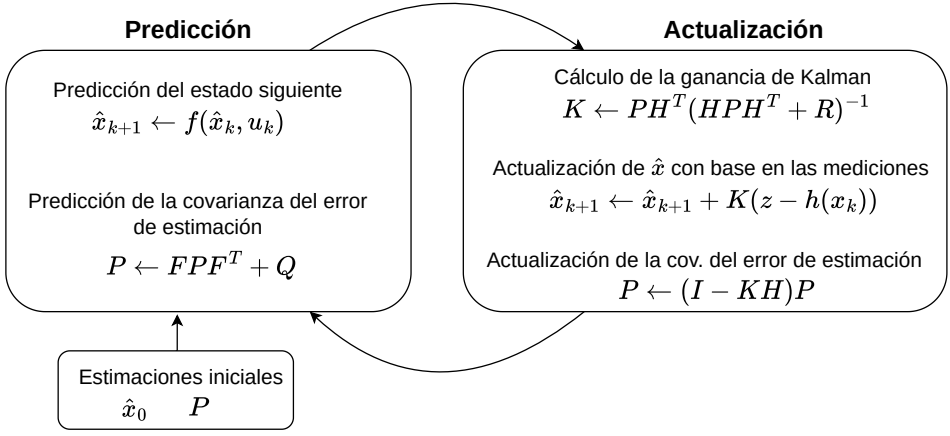
donde $n_p \in \mathbb{R}^3$ es ruido de proceso que es distribuye normalmente con media cero y matriz de covarianza conocida $Q \in \mathbb{R}^{3\times 3}$

Modelo de observación:

$$\left.\begin{array}{rcll} & \vdots & & \\ r_i & = & \sqrt{(x_k - x_{oi})^2 + (y_k - y_{oi})^2} & +n_{m1} \\ \phi_i & = & atan2(y_k - y_{oi}, x_k - x_{oi}) - \theta_k & +n_{m2} \\ & \vdots & & \end{array}\right\} h(x)$$

donde M es el número de marcas observadas, $n_m \in \mathbb{R}^{2M}$ es ruido de medición que se distribuye normalmente con media cero y matriz de covarianza conocida $R \in \mathbb{R}^{2M\times 2M}$

Algoritmo de estimación del EKF



con:

- P : Matriz de covarianza del error de estimación
- F : Jacobiano de $f(x)$
- H : Jacobiano de $h(x)$
- z : Mediciones de los sensores

Filtros de partículas

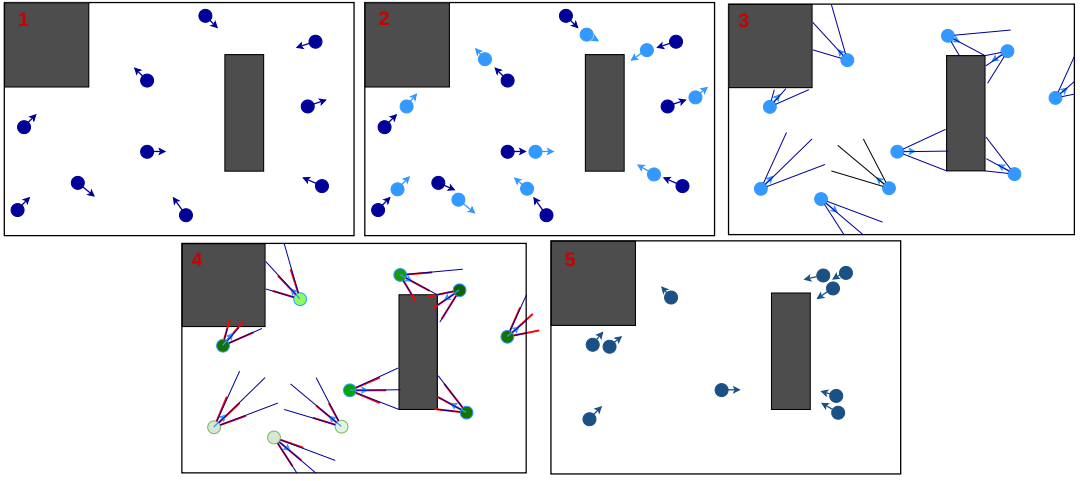
- También son un filtro Bayesiano
- A diferencia del EKF que supone una distribución normal, los filtros de partículas suponen una distribución no paramétrica.
- Se consideran N suposiciones de la posición del robot. A cada suposición (x, y, θ) se le llama partícula.
- La distribución *a priori* se calcula realizando simulaciones de movimiento para cada partícula.
- El modelo de observación también se realiza mediante simulaciones de los sensores para cada partículas.
- Para obtener la distribución *a posteriori* se comparan las lecturas simuladas con la lectura del sensor real y se realiza un muestreo aleatorio con reemplazo. Cada partícula tiene una probabilidad de ser muestreada, proporcional a la similitud de su lectura simulada con el sensor real.

Filtros de Partículas

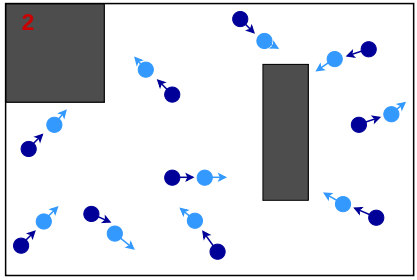
Un filtro de partículas para localización consta de 5 pasos generales:

1. Generar N partículas con (x, y, θ) aleatorias con distribución uniforme (distribución inicial equivalente a $\hat{x}_0$ en el EKF)
2. Mover cada partícula un desplazamiento $(\Delta x, \Delta y, \Delta\theta)$ más ruido Gaussiano (equivalente a estimar $\hat{x}_{k+1} = f(x_k, u_k)$ en el EKF)
3. Simular las lecturas del sensor láser para cada partícula (equivalente a predecir las salidas $h(x_k)$ en el EKF)
4. Comparar cada lectura simulada con la lectura del sensor real. Las similitudes representan una distribución de probabilidad
5. Remuestrear N partículas con reemplazo usando la distribución anterior, y agregar ruido Gaussiano (equivalente a la actualización de la estimación de $\hat{x}$ en el EKF)

Filtros de Partículas



Desplazamiento de partículas



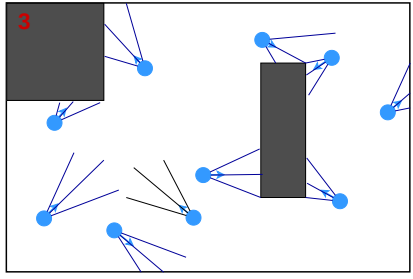
- Se puede suponer que todas las partículas se desplazan una cantidad $(\Delta x, \Delta y, \Delta\theta)$ (obtenida mediante odometría) con respecto al sistema del robot
- Para calcular la nueva pose de cada partícula, basta con transformar los desplazamientos al sistema de referencia (giro sobre Z de un ángulo θ)
- Para modelar la incertidumbre del movimiento, a cada partícula se le suma ruido Gaussiano n con media cero y varianza σ_m^2:

Para cada partícula, hacer:

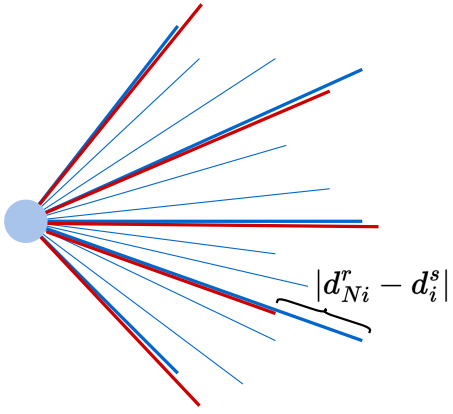
$$\begin{aligned} x_{k+1} &= x_k + \Delta x \cos\theta_k - \Delta y \sin\theta_k + n_x \\ y_{k+1} &= y_k + \Delta x \sin\theta_k + \Delta y \cos\theta_k + n_y \\ \theta_{k+1} &= \theta_k + \Delta\theta + n_\theta \end{aligned}$$

Simulación del sensor

- Se puede simular con técnicas de *ray tracing* a partir de un mapa y una pose del sensor
- Se puede utilizar el paquete `occupancy_grid_utils`. Tiene una biblioteca que simula las lecturas del láser dado:
  - Un mapa de celdas de ocupación (`OccupancyGrid`)
  - Una posición y orientación del sensor (`Pose`). Esta posición y orientación corresponde con el (x, y, θ) de cada partícula.
  - Especificaciones del sensor a simular. Esto lo obtiene de un mensaje `LaserScan`
- Puesto que esta simulación es computacionalmente costosa, solo se simulan **1 de cada** N lecturas del sensor real.



Comparación de simulación con real



Para obtener una medida de comparación se obtiene una media de diferencias en las distancias:

$$\delta = \sum_{i=0}^{M_s - 1} |d^r_{Ni} - d^s_i|$$

$$s = e^{-\delta^2/\sigma^2}$$

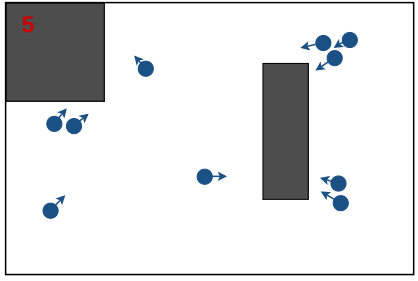
con:

- d^r_{Ni}: es la (Ni)-ésima lectura de distancia del sensor real
- d^s_i: es la i-ésima lectura de distancia del sensor simulado (de una partícula)
- δ: media de los valores absolutos de las diferencias
- σ^2: Varianza del sensor (medida de incertidumbre)
- s: medida de similitud entre el sensor real y el sensor simulado
- M_s: número de distancias en el sensor simulado ($1/N$ del número de distancias en el sensor real).

Recuerde que se simularon solo 1 de cada N lecturas del sensor real

Remuestreo con reemplazo

- Las similitudes obtenidas en el paso anterior pueden usarse como distribución de probabilidad si se normalizan para que sumen 1
- La distribución de probabilidad anterior determina la probabilidad de que una partícula sea remuestreada
- Para evitar tener partículas repetidas, a cada partícula remuestreada se le suma ruido Gaussiano



Evasión de obstáculos

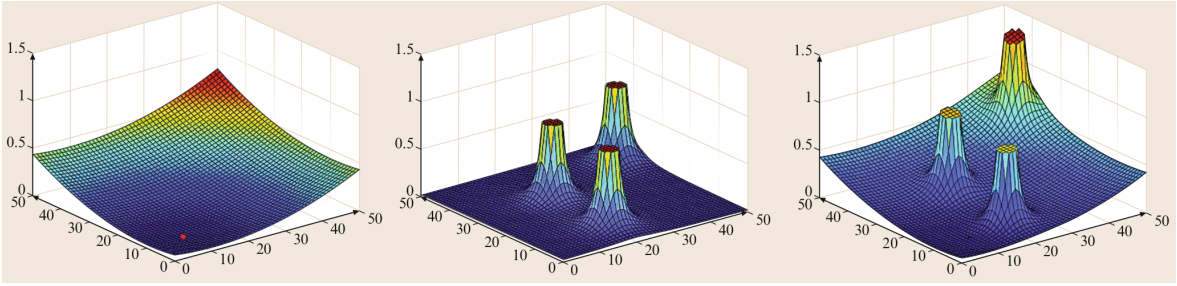
- Hasta el momento se tiene una manera de representar el ambiente, planear una ruta y seguirla
- ¿Qué pasa si en el ambiente hay un obstáculo que no estaba en el mapa?
- Se requiere de una técnica reactiva para evadir obstáculos
- Una posible solución es el uso de campos potenciales artificiales

Campos potenciales artificiales

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

- El gradiente $\nabla U(q) = \left[\frac{\partial U}{\partial q_1}, \ldots, \frac{\partial U}{\partial q_n}\right]$ 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.
- Ha 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:

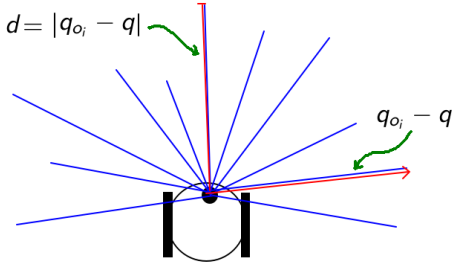
$$\begin{aligned} F_{att} &= \eta \frac{(q - q_g)}{\|q - q_g\|}, \qquad \zeta > 0 \\ F_{rej} &= \begin{cases} \zeta \left(\sqrt{\frac{1}{d} - \frac{1}{d_0}} \right) \frac{q_{o_i} - q}{d} & \text{si} \quad d < d_0 \\ 0 & \text{en otro caso} \end{cases} \end{aligned}$$

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_{o_i}, en realidad ésta aparece siempre en la diferencia $q_{o_i} - q$, 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:

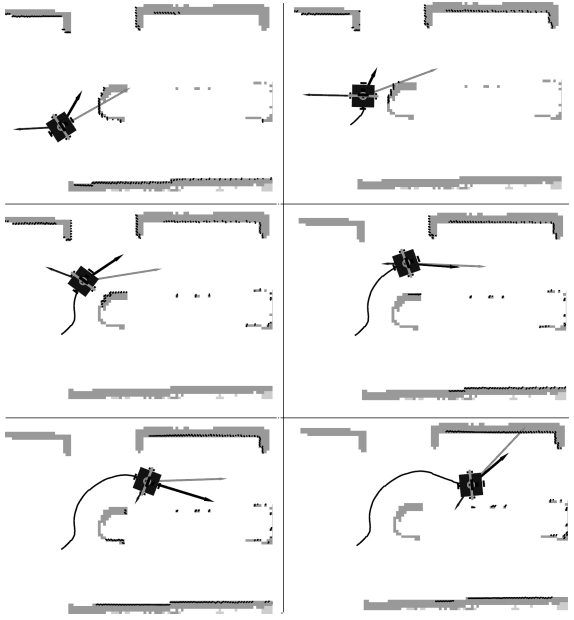
Algoritmo 4: Descenso del gradiente para mover al robot a través de un campo potencial.

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

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

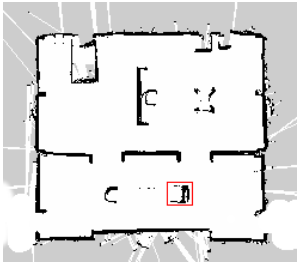
Evasión de obstáculos por campos potenciales

Ejemplo de movimiento:

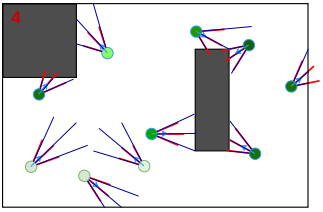


Navegación: el enfoque tradicional

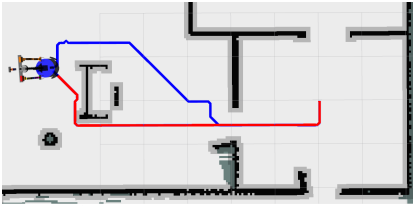
Mapas geométricos (los más comunes son las celdas de ocupación) para representar el ambiente



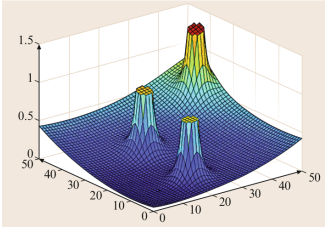
Filtros de partículas (también EKF) para localización y mapeo



Planeadores de rutas basados en grafos o basados en muestreo



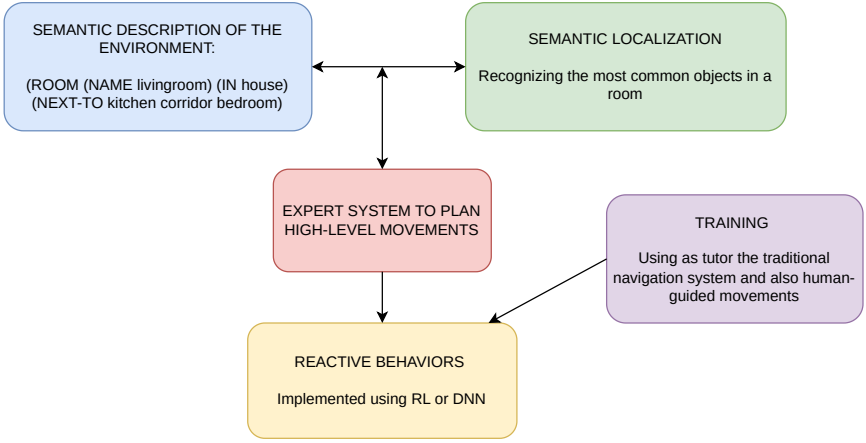
Evasión de obstáculos mediante campos potenciales



Desventaja: Se requiere un mapa geométrico. No es posible aprender a navegar y luego usar el conocimiento para navegar en un ambiente diferente.

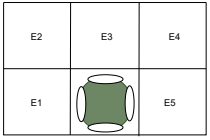
Navegación: ¿qué sigue?

Navegación sin mapa (geométrico)

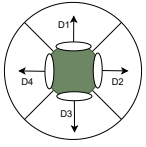


Navegación reactiva usando DNN

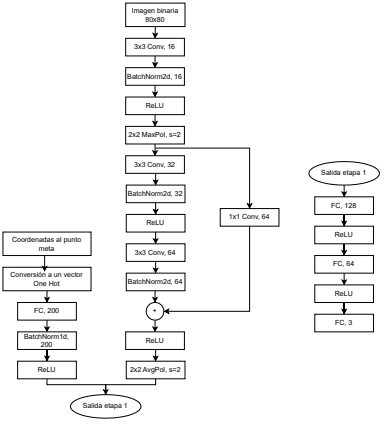
Discretización del espacio alrededor del robot:



Discretización de los comandos de velocidad como acciones::

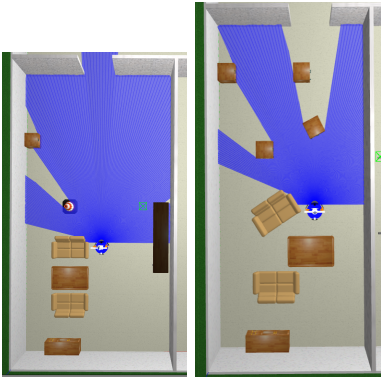


DNN para generar comandos de velocidad:



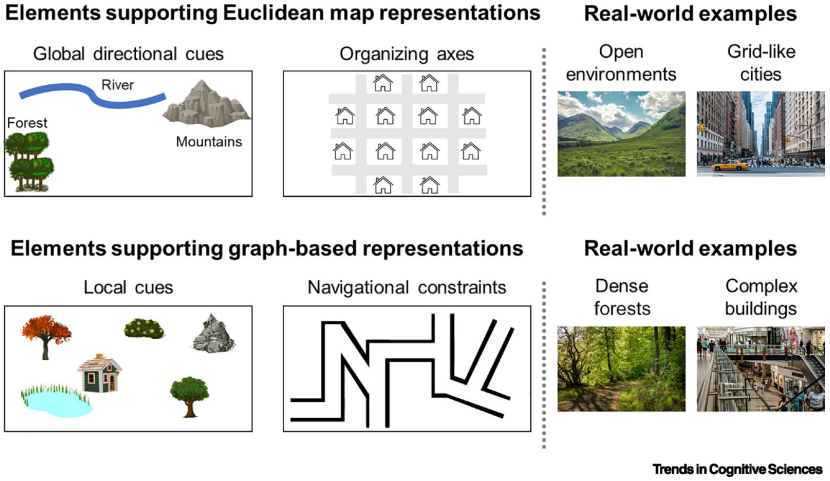
Entrenamiento usando datos generados por humanos y datos generados usando campos potenciales

Pruebas en ambientes diferentes al de entrenamiento:



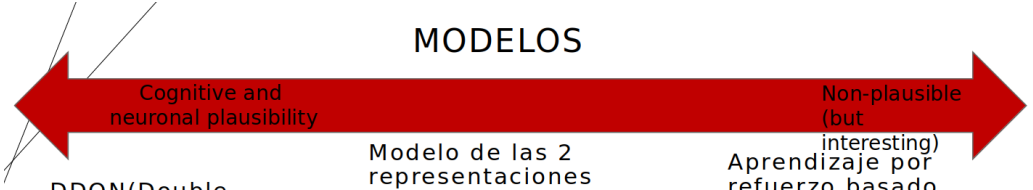
En el ambiente conocido (izq), la navegación basada en grafos y los campos potenciales superaron a las DNN, pero en el ambiente desconocido (derecha), la DNN superó a ambos métodos.

Modelos psicológicos sobre navegación



Se ha encontrado evidencia de como navegamos por el mundo, identificando principalmente 2 mecanismos cognitivos, navegación alocentrica y egócentrica.

Modelos psicológicos sobre navegación



DDQN(Double Deep Q Networks)

Redes neuonales dentro de algoritmos de aprendizaje por refuerzo, se ha demostrado que las neuronas adquieren roles parecidos a las vistas en place cells, wall cells, etc

Modelo de las 2 representaciones en paralelo con meta decisiones

Tener la representación alócentrica y egocentrica en un mismo modelo, que pueda elegir una sobre otra en determinado momento de acuerdo a las circunstancias.

Aprendizaje por refuerzo basado en modelos

Algoritmos de aprendizaje por refuerzo basado en modelos, que pueden variar en la cantidad de información del ambiente que se da en un inicio y los sesgos inductivos que se programen. La major oportunidad de usar Bayes para cuantificar como entiende el entorno y como se comporta en el.

Este curso no solo no hubiera sido nada sin ustedes, sino con toda la gente que ha asistido a la EIR desde el comienzo, algunos, siguen hasta hoy...

¡Gracias! ¡Totales!

Contacto

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

marco.negrete@ingenieria.unam.edu