ETAPA 04
Planeación de movimientos

Instructor: Marco Antonio Negrete Villanueva

Programa Espacial Universitario, UNAM

Consurso Iberoamericano de Robótica Espacial 2022
https://github.com/mnegretev/CIRE2022

Objetivos:

Objetivo General: Que los participantes comprendan los conceptos básicos para dotar de navegación autónoma a un robot móvil.

Objetivos Específicos:

- Dar una introducción a las tareas del problema de la planeación de movimientos
- Dar un panorama sobre las técnicas de representación del ambiente
- Dar un panorama sobre las técnicas de planeación de rutas
- Dar un panorama sobre las técnicas de localización
- Dar una introducción al problema del mapeo y localización simultáneos

Contenido

Planeación de movimientos

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 lecuras 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 $q \in Q_b \subset Q$.

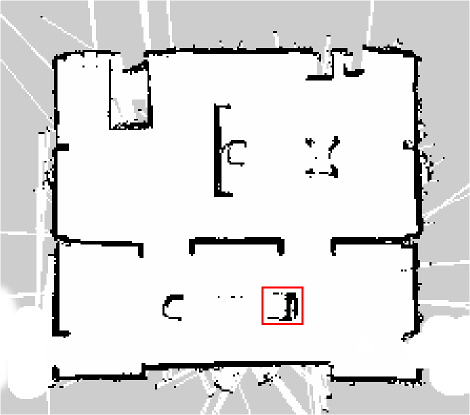
Representación del ambiente

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

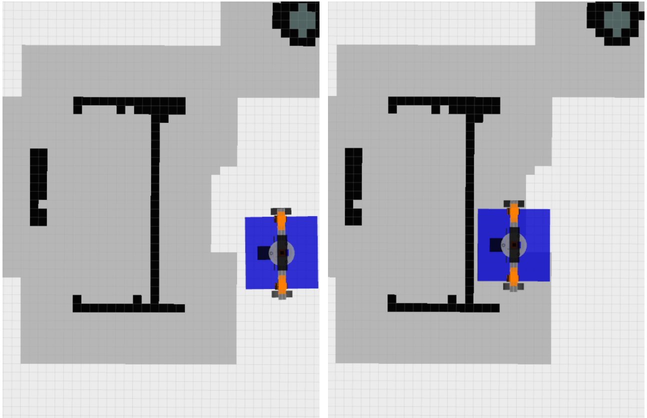
Es un tipo de mapa geométrico. El espacio se discretiza con una resolución determinada y a cada celda se le asigna un número $p \in [0, 1]$ que indica su nivel de ocupación. En un enfoque probabilístico este número se puede interpretar como la certeza que se tiene de que una celda esté ocupada.



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.

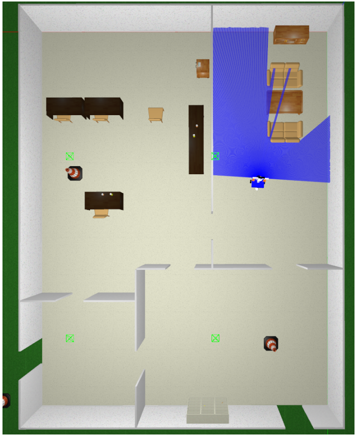
Inflado de celdas de ocupación

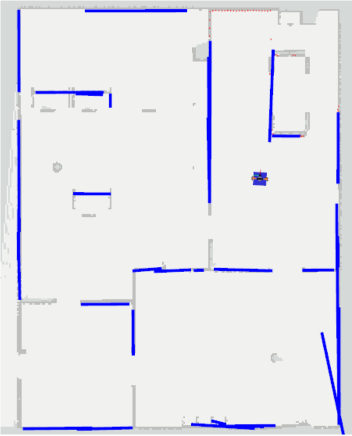
Algoritmo 1: Algoritmo de inflado de mapas

```
Data:
Mapa M de celdas de ocupación
Radio de inflado r_i
Result: Mapa inflado M_inf
M_inf = Copia de M
foreach i ∈ [0, ..., rows) do
    foreach j ∈ [0, ..., 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 ∈ [−r_i, ..., r_i] do
                foreach k_2 ∈ [−r_i, ..., r_i] do
                    M_inf[i + k_1, j + k_2] = 100
                end
            end
        end
    end
end
```

Mapas de líneas

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





Algunos métodos para extraer líneas:

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

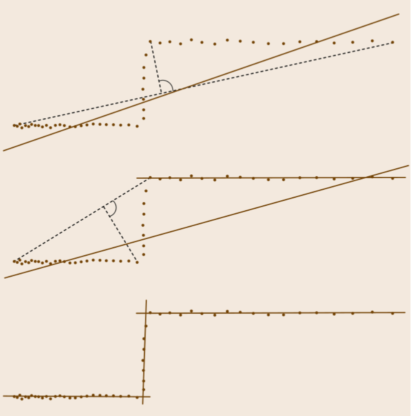
Algoritmo *Split and Merge*

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

Algoritmo 2: *Split and Merge*

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

Ajustar una recta L al conjunto P por mínimos cuadrados
Encontrar el punto p_i más lejano a la recta
if $d(p_i, L) > umbral$ **then**
  Dividir P en dos subconjuntos P_1 y P_2 usando p_i como pivote
  Aplicar este algoritmo recursivamente para P_1 y P_2
  Devolver las rectas de ambos subconjuntos
else
  Devolver la recta L en forma normal (ρ, θ)
end



Mínimos cuadrados

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

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

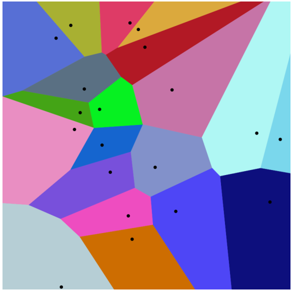
$$
\begin{aligned}
\theta &= \frac{1}{2}\text{atan2}\left(-2\sum_i(\bar{x} - x_i)(\bar{y} - y_i) \quad , \quad \sum_i \left[(\bar{y} - y_i)^2 - (\bar{x} - x_i)^2\right]\right) \\
\rho &= \bar{x}\cos\theta + \bar{y}\sin\theta
\end{aligned}
$$

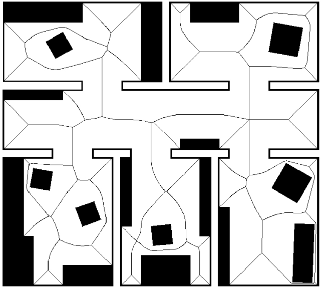
con

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

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 generalizdo (GVD) considera que las semillas pueden ser objetos con dimensiones y no solo puntos.

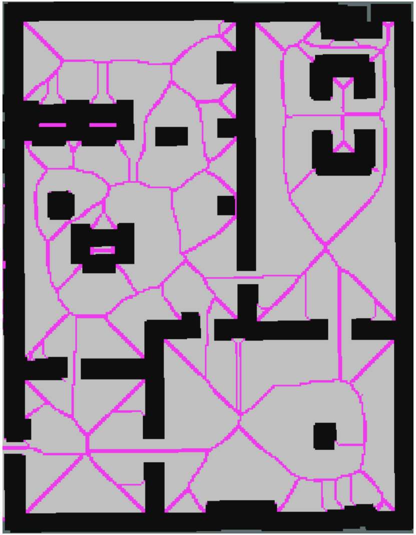




- La forma de las regiones depende de la función de distancia que se utilice.

El algoritmo *Brushfire*

- Obtener un GVD es aún un problema abierto
- Se simplifica el problema si se asume 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.

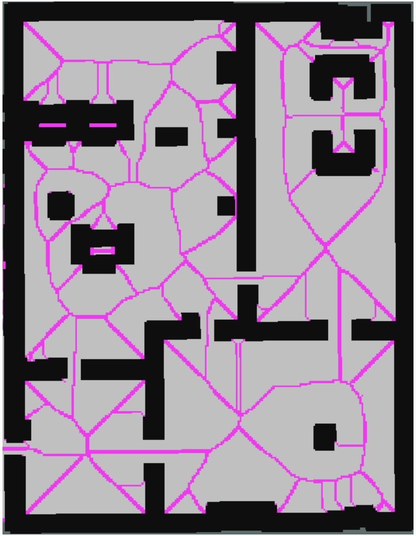


El algoritmo *Brushfire*

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**
            Agregar 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 1 - Inflado de mapas

1. Corra la simulación mediante el comando `roslaunch bring_up stage04.launch`
2. Abra el archivo `catkin_ws/src/exercises/scripts/map_inflation.py` y agregue el siguiente código en la línea 32:

```
32 for i in range(height):
33     for j in range(width):
34         if static_map[i,j] > 50:
35             for k1 in range(-inflation_cells, inflation_cells+1):
36                 for k2 in range(-inflation_cells, inflation_cells+1):
37                     inflated[i+k1, j+k2] = 100
38
```

3. Modifique radio de inflado mediante el comando:

```
rosparam set /path_planning/inflation_radius 0.4
```

4. ¿Qué cambios observa?

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

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:

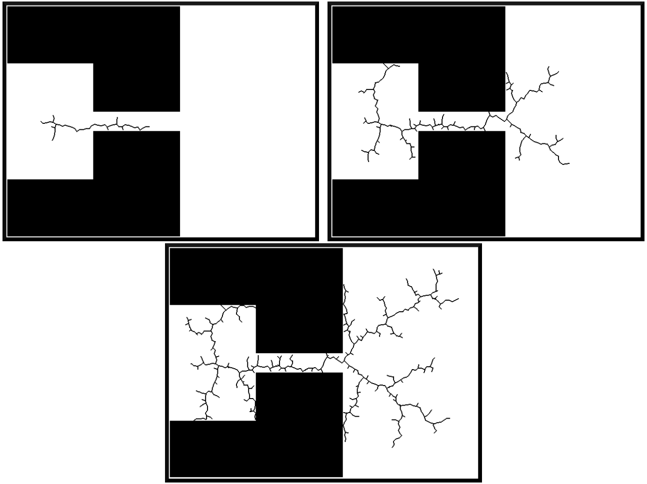
- RRT (Rapidly-exploring Random Trees)
- RRT-Bidireccional
- RRT-Extendido

Rapidly-exploring Random Trees

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

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 = ConfiguracionAleaoria();
    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 algorimos se pueden resumir en:

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

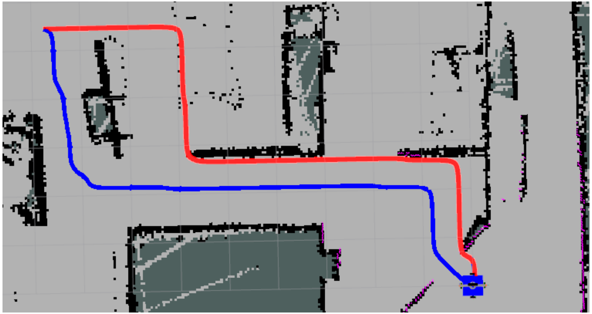
El criterio para seleccinar el siguiente nodo a expandir n_c de la lista abierta, determina el tipo de algoritmo:

- 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.



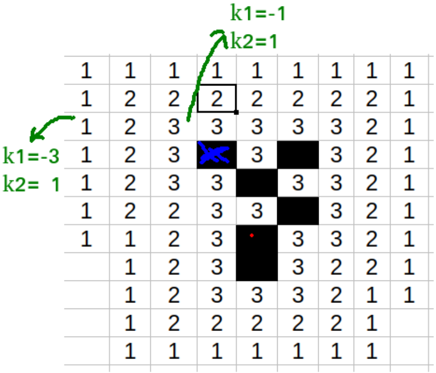
Mapas de costo

- 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 $i \in [0, \dots, rows)$ **do**
  foreach $j \in [0, \dots, cols)$ **do**
    //Si está ocupada, calcular el costo de r_c celdas alrededor.
    if $M[i,j] == 100$ **then**
      foreach $k_1 \in [-r_c, \dots, r_c]$ **do**
        foreach $k_2 \in [-r_c, \dots, r_c]$ **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 $P = [q_s, q_1, q_2, \dots, q_g]$
Lista abierta $OL = \emptyset$ y lista cerrada $CL = \emptyset$
Fijar $f(n_s) = 0$, $g(n_s) = 0$ y $prev(n_s) = NULL$
Agregar n_s a OL y fijar nodo actual $n_c = n_s$
while $OL \neq \emptyset$ *y* $n_c \neq n_g$ **do**
  Remover de OL el nodo n_c con el menor valor f y agregar n_c a CL
  forall n *vecino de* n_c **do**
    $g = g(n_c) + costo(n_c, n)$
    if $g < g(n)$ **then**
      $g(n) = g$
      $f(n) = h(n) + g(n)$
      $prev(n) = n_c$
    end
  end
  Agregar a OL los vecinos de n_c que no estén ya en OL ni en CL
end
if $n_c \neq n_g$ **then**
  Anunciar Falla
end
while $n_c \neq NULL$ **do**
  Insertar al inicio de la ruta P la configuración correspondiente al nodo n_c
  $n_c = prev(n_c)$
end
Devolver ruta óptima P

El algoritmo A*

- La función de costo será el número de celdas más el mapa de costo de la clase anterior.
- Puesto que el mapa está compuesto por celdas de ocupación, los nodos vecinos se pueden obtener usando conectividad 4 o conectividad 8.
- Si se utiliza conectividad 4, la distancia de Manhattan es una buena heurística.
- Si se utiliza conectividad 8, se debe usar la distancia 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.
- La obtención de las coordenadas (x, y) a partir de los nodos n se puede hacer con:

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

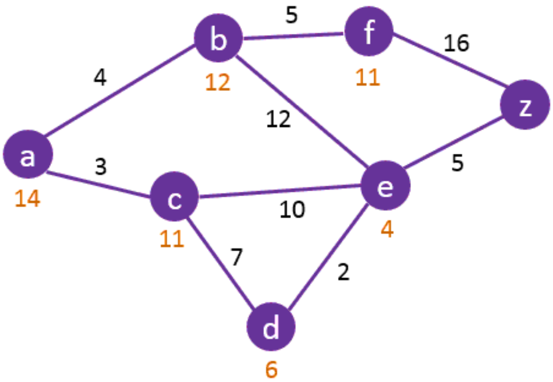
- La obtención del renglón-columna (r, c) del nodo n a partir de (x, y), se puede obtener con:

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

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

El algoritmo A*

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



El algoritmo A*

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

A g,f,p	B g,f,p	C g,f,p	D g,f,p	E g,f,p	F g,f,p	Z g,f,p
0,0,NULL	∞, ∞,NULL	∞, ∞,NULL	∞, ∞,NULL	∞, ∞,NULL	∞, ∞,NULL	∞, ∞,NULL
0,0,NULL	4, 16, A	3, 14, A	∞, ∞,NULL	∞, ∞,NULL	∞, ∞,NULL	∞, ∞,NULL
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	13, 17, C	∞, ∞,NULL	∞, ∞,NULL
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	13, 17, C	9, 20, B	∞, ∞,NULL
0,0,NULL	4, 16, A	3, 14, A	10, 16, C	**12, 16, D**	9, 20, B	∞, ∞,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

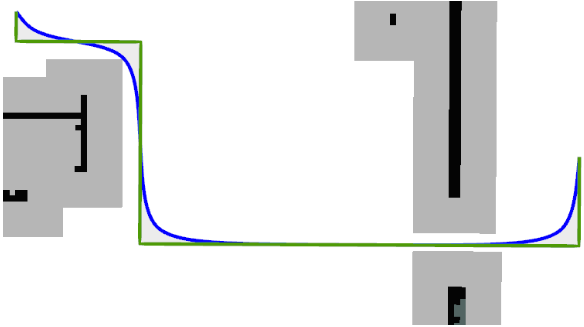
Ejercicio 2 - Planeación de rutas

Realice lo siguiente:

1. Corra la simulación mediante el comando `roslaunch bring_up stage04.launch`
2. Corra los nodos de inflado de mapas, mapa de costo y A* (`map_inflation.py`, `cost_map.py` y `a_star.py`).
3. Modifique el radio de costo mediante el comando:
   `rosparam set /path_planning/cost_radius 0.4`
4. Calcule una ruta a un punto meta mediante el comando:
   `rosservice call /path_planning/a_star_search`
   (Autocomplete el comando y utilice los campos *goal-¿pose-¿position-¿x* y *goal-¿pose-¿position-¿y*
5. En el archivo `a_star.py`, cambie la función de distancia de Manhattan por distancia Euclideana, y cambie la conectividad 4 por conectividad 8 y vea qué sucede.
6. Modifique el código para que h sea siempre cero y vea qué sucede con el número de pasos. Pruebe con varias rutas.

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.



Existen varias formas de suavizar la ruta generada:

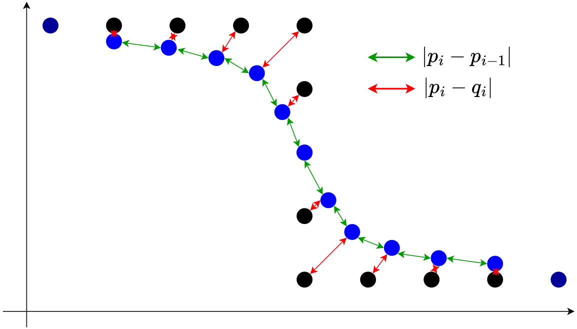
- Splines
- Descenso del gradiente

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**
    $p \leftarrow p - \epsilon \nabla J(p)$ //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:

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

Suavizado mediante descenso del gradiente

Para no variar los puntos inicial y final de la ruta, la primer y última componentes de ∇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 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 $\|\nabla J(p_i)\| > tol$ **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

Práctica 4 - Suavizado de rutas

Realice lo siguiente:

1. Abra el archivo `catkin_ws/src/students/scripts/practice04.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 los nodos de inflado de mapas, mapa de costo, A* y suavizado de rutas (practice02.py, practice03a.py, practice03b.py y practice04.py)
3. Mediante la GUI, modifique los parámetros α y β del suavizado y observe qué sucede con el cálculo de rutas.

Práctica 4 - Suavizado de rutas

Entregables:

- Código modificado en la rama correspondiente del repositorio en línea.
- Documento escrito con los siguientes puntos:
  - Introducción (el problema del suavizado de rutas y el algoritmo del descenso del gradiente)
  - Objetivo: Implementar el descenso del gradiente para suavizar una ruta calculada con A*
  - Descripción del descenso del gradiente
  - Resultados del inciso anterior
  - Conclusiones
  - Referencias

Deadline: 2022-09-27 al inicio de la clase.

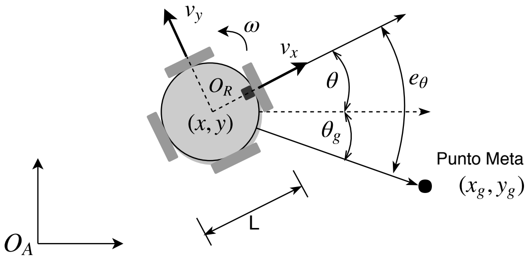
Seguimiento de rutas

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

Modelo cinemático

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



El modelo cinemático está dado por

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

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

Control de posición

- Las leyes de control se diseñarán considerando una base diferencial
- Es mejor mover al robot así, pues lo 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

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

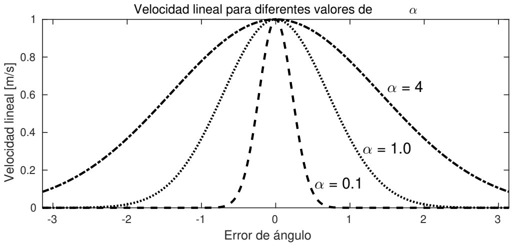
El error de ángulo e_θ debe estar siempre en el intervalo $(-\pi, \pi]$. Si la diferencia resulta en un valor fuera de este ángulo, se puede acotar mediante:

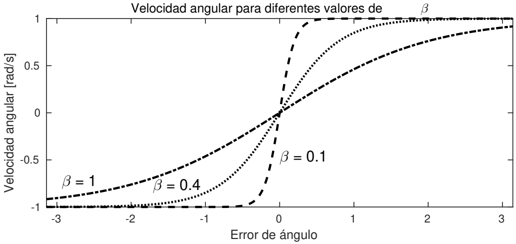
$$e_\theta \leftarrow \left(e_\theta + \pi\right) \%(2\pi) - \pi$$

donde % denota el operador módulo (residuo).

Control de posición

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





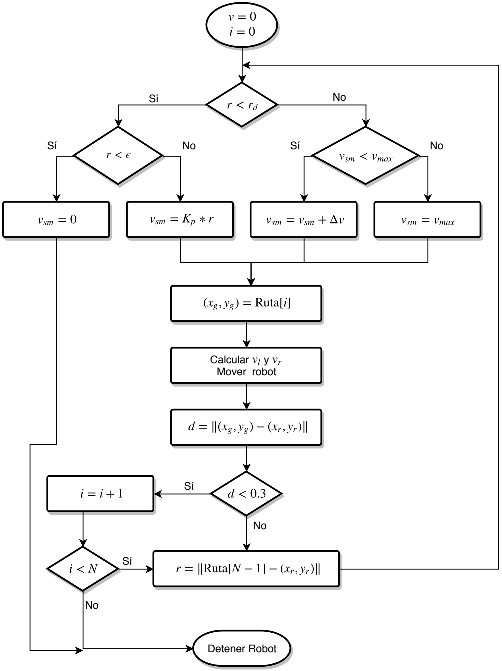
Seguimiento de rutas

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

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

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

Perfil de velocidad



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

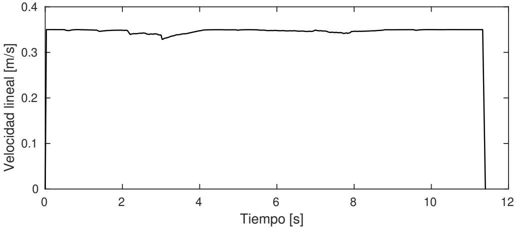
$$\begin{aligned} v &= v_{sm} e^{-\frac{e_\theta^2}{\alpha}} \\ \omega &= \omega_{max} \left(\frac{2}{1 + e^{-\frac{e_\theta}{\beta}}} - 1 \right) \end{aligned}$$

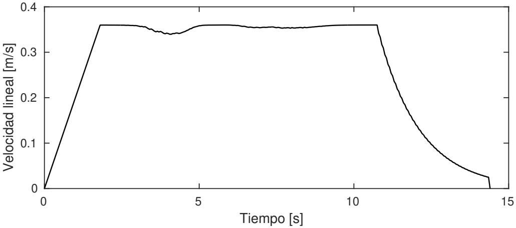
con

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

Perfil de velocidad

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





Práctica 5 - Seguimiento de rutas

- Código modificado en la rama correspondiente del repositorio en línea.
- Documento escrito con los siguientes puntos:
  - Introducción (el problema del suavizado de rutas y el algoritmo del descenso del gradiente)
  - Objetivo: Implementar el descenso del gradiente para suavizar una ruta calculada con A*
  - Descripción del descenso del gradiente
  - Resultados del inciso anterior
  - Conclusiones
  - Referencias

Evasión de obstáculos

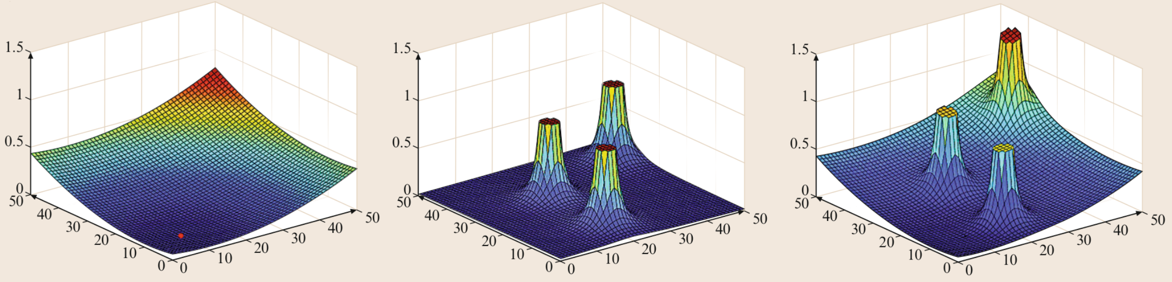
- Hasta el momento se tiene una manera de representar el ambiente, planear una ruta y seguirla
- ¿Qué pasa si en el ambiente hay un obstáculo que no estaba en el mapa?
- Se requiere de una técnica reactiva para evadir obstáculos
- Una posible solución es el uso de campos potenciales artificiales

Campos potenciales artificiales

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

- El gradiente $\nabla U(q) = \left[\frac{\partial U}{\partial q_1}, \dots, \frac{\partial U}{\partial q_n}\right]$ es una fuerza.
- Se debe diseñar de modo que tenga un mínimo global en el punto meta y máximos locales en cada obstáculo.
- Si el robot se mueve siempre en sentido contrario al gradiente ∇U llegará al punto meta siguiendo una ruta alejada de los obstáculos.
- 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:

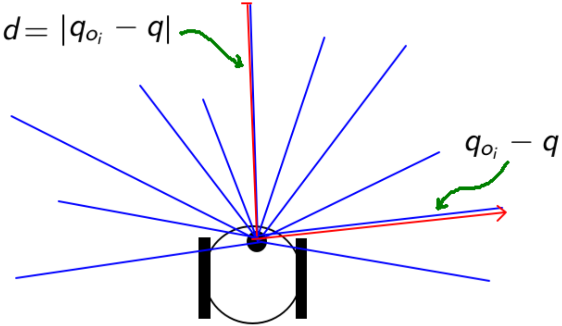
$$\begin{aligned} F_{att} &= \zeta\frac{(q - q_g)}{\|q - q_g\|}, \qquad \zeta > 0 \\ F_{rej} &= \begin{cases} \eta\left(\sqrt{\dfrac{1}{d} - \dfrac{1}{d_0}}\right)\dfrac{q_{o_i} - q}{d} & \text{si} \quad d < d_0 \\ 0 & \text{en otro caso} \end{cases} \end{aligned}$$

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.



Evasión de obstáculos por campos potenciales

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

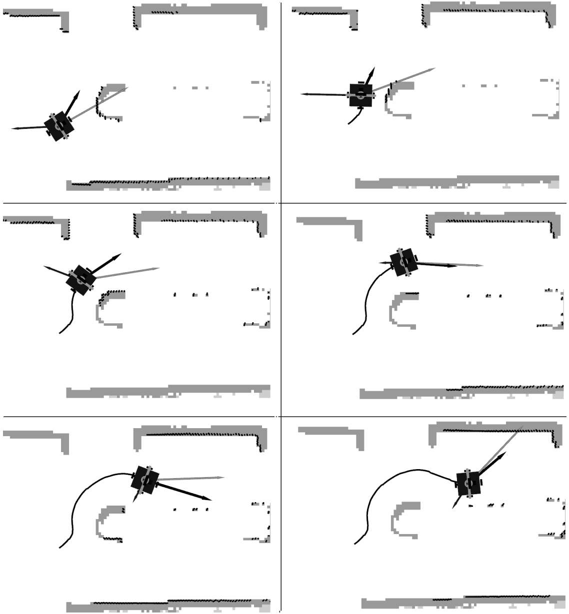
Algoritmo 8: Descenso del gradiente para mover al robot a través de un campo potencial.

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

$q \leftarrow q_s$
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:



Práctica 6 - Evasión de obstáculos

Entregables:

- Código modificado en la rama correspondiente del repositorio en línea.
- Documento escrito con los siguientes puntos:
  - Introducción (el problema del suavizado de rutas y el algoritmo del descenso del gradiente)
  - Objetivo: Implementar el descenso del gradiente para suavizar una ruta calculada con A*
  - Descripción del descenso del gradiente
  - Resultados del inciso anterior
  - Conclusiones
  - Referencias

Deadline: 2022-09-27 al inicio de la clase.

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. 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.

Filtros de partículas

Son un método de Monte Carlo, es decir, se repite un muestreo aleatorio hasta obtener resultados numéricos. La idea de los filtros de partículas es muy sencilla:

1. Inicializar un número grande N de partículas (robots simulados) con configuraciones aleatorias (x_i, y_i, θ_i).
2. Para cada partícula, simular las lecturas del sensor (generalmente un lidar) dado un mapa.
3. Comparar cada lectura simulada con la lectura del sensor real, y asignar a cada partícula una medida de similitud.
4. **Realizar un muestreo aleatorio con reemplazo de N partículas usando las similitudes como distribución de probabilidad**.
5. Agregar ruido a cada partícula muestreada.
6. Mover cada partícula una cantidad igual al desplazamiento del robot, más un pequeño ruido.
7. Volver al paso 2.

Práctica 7 - Localización

Entregables:

- Código modificado en la rama correspondiente del repositorio en línea.
- Documento escrito con los siguientes puntos:
    - Introducción (el problema del suavizado de rutas y el algoritmo del descenso del gradiente)
    - Objetivo: Implementar el descenso del gradiente para suavizar una ruta calculada con A*
    - Descripción del descenso del gradiente
    - Resultados del inciso anterior
    - Conclusiones
    - Referencias

Deadline: 2022-09-27 al inicio de la clase.

Localización y mapeo simultáneos

Su misión, si deciden aceptarla...

Para más información...

Gracias

Contacto

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

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