**Práctica 4. Planificación y seguimiento de rutas**

Javier Ahumada, Rodrigo Covas, Javier Escobar

Contenido

[1. Aquí sí, aquí no 2](#_Toc193714771)

[2. Conectando los puntos 3](#_Toc193714772)

[3. ¿Cuándo llegamos? ¿Falta mucho? 3](#_Toc193714773)

[4. Debes encontrar el camino 4](#_Toc193714774)

[5. El rally de Montecarlo 5](#_Toc193714775)

[6. Me encanta que los planes salgan bien 6](#_Toc193714776)

[7. Análisis de sensibilidad del PRM 7](#_Toc193714777)

[8. ¿Quién conoce la ruta? 7](#_Toc193714778)

[9. Encontrando el rastro 7](#_Toc193714779)

[10. ¡Tierra a la vista! 7](#_Toc193714780)

[11. ¡Que empiece la carrera! 7](#_Toc193714781)

[12. Sensibilidad de la persecución pura 7](#_Toc193714782)

[13. Azul, rosa, azul, rosa... 7](#_Toc193714783)

# 1. Aquí sí, aquí no

En esta sección implementaremos el método “generate\_nodes” que cree puntos de paso válidos para construir posteriormente un grafo de navegación. El método tendrá dos formas de uso: generar puntos en una rejilla de puntos equidistantes o generar puntos aleatorios muestreando de una distribución uniforme. Tendremos que verificar que los puntos generados son válidos para el robot, por lo que solo se guardarán los puntos generados dentro del mapa (utilizando paredes engrosadas, puesto que el robot es más ancho que los puntos de la simulación).

graph: dict[tuple[float, float], list[tuple[float, float]]] = {}

x\_min, y\_min, x\_max, y\_max = self.\_map.bounds()

if use\_grid:

x = x\_min

while x <= x\_max:

    y = y\_min

while y <= y\_max:

rounded\_x = round(x, 3)

rounded\_y = round(y, 3)

if self.\_map.contains((rounded\_x, rounded\_y)):

graph[(rounded\_x, rounded\_y)] = []

y += grid\_size

x += grid\_size

else:

x\_min, y\_min, x\_max, y\_max = self.\_map.bounds()

for i in range(node\_count):

x: float

y: float

valid = False

while not valid:

x = round(random.uniform(x\_min, x\_max), 3)

y = round(random.uniform(y\_min, y\_max), 3)

valid = self.\_map.contains((x, y))

graph[(x, y)] = []

return graph

# 2. Conectando los puntos

En esta sección implementaremos el método “connect\_nodes”. Este se encargará de conectar cada uno de los nodos generados en el apartado anterior con todos aquellos nodos conectables con una línea recta que estén como máximo a una distancia umbral. Utilizaremos el método “crosses” de la clase “Map” para comprobar que a los nodos les une una línea recta que no traspasa ninguna de las paredes del mapa.

def calculate\_dis(pos\_1, pos\_2):

return math.sqrt((pos\_1[0] - pos\_2[0]) \*\* 2 + (pos\_1[1] - pos\_2[1]) \*\* 2)

def get\_nodes(node\_pos):

return [

target\_pos

for target\_pos in graph.keys()

if (node\_pos != target\_pos)

and (calculate\_dis(node\_pos, target\_pos) <= connection\_distance)

and (not self.\_map.crosses([target\_pos, node\_pos]))

            ]

graph = {node\_pos: get\_nodes(node\_pos) for node\_pos in graph.keys()}

return graph

# 3. ¿Cuándo llegamos? ¿Falta mucho?

En este apartado vamos a implementar un algoritmo A\* para obtener el camino más corto entre dos puntos del mapa utilizando el grafo que hemos construido. Para este algoritmo utilizaremos como heurística la distancia Euclídea al nodo objetivo. La función de coste será la distancia recorrida por el robot.

En cada paso del algoritmo elegiremos el nodo de la lista abierta con menor valor f (el coste más la heurística) y añadiremos a la lista abierta todos los nodos que conectan con él en el grafo que no se encuentren ya en la lista abierta. Si algún nodo se encuentra ya en la lista abierta pero su coste es menor accediendo desde el nodo que estamos explorando, actualizamos su ancestro al nodo actual.

path: list[tuple[float, float]] = []

def calculate\_dis(node, pos):

return math.sqrt((node[0] - pos[0]) \*\* 2 + (node[1] - pos[1]) \*\* 2)

initial\_node = min(self.\_graph, key=lambda node: calculate\_dis(node, start))

final\_node = min(self.\_graph, key=lambda node: calculate\_dis(node, goal))

open\_list: dict[tuple[float, float], tuple[float, float]] = {}

close\_list: set[tuple[float, float]] = set()

open\_list[initial\_node] = (calculate\_dis(initial\_node, final\_node), 0)

node = initial\_node

while open\_list != {} and node != final\_node:

node = min(open\_list, key=lambda k: open\_list.get(k)[0])

f, g = open\_list[node]

for neighbour in self.\_graph[node]:

h\_new = calculate\_dis(neighbour, final\_node)

g\_new = g + calculate\_dis(neighbour, node)

f\_new = h\_new + g\_new

if neighbour not in close\_list:

if neighbour not in open\_list

or open\_list[neighbour][1] > g\_new:

open\_list[neighbour] = (f\_new, g\_new)

ancestors[neighbour] = node

close\_list.add(node)

open\_list.pop(node)

if final\_node in close\_list:

if initial\_node != start:

ancestors[initial\_node] = start

if final\_node != goal:

ancestors[goal] = final\_node

path = self.\_reconstruct\_path(start, goal, ancestors)

return path

else:

raise PathNotFound("Path not found")

# 4. Debes encontrar el camino

En este apartado implementaremos la función “reconstruct\_path”, que recibe el inicio y el final del camino junto con la lista de ancestros y devuelve el camino encontrado en formato de lista de tuplas.

path = [goal]

node = goal

while node != start:

next\_node = ancestors[node]

path.insert(0, next\_node)

node = next\_node

return path

# 5. El rally de Montecarlo

En este apartado implementamos la función “smooth\_path”. Esta se encarga de suavizar el camino para facilitar su recorrido por robots no holonómicos como el nuestro.

En primer lugar, añadimos una cierta cantidad de puntos entre un nodo y otro del camino, añadiendo cada uno a una distancia equidistante entre el anterior y el siguiente.

Después realizamos descenso gradiente para minimizar dos condiciones: la distancia entre los puntos de la ruta, y la diferencia con la ruta original. Calcularemos cuánto ha cambiado la ruta entre una iteración y otra del descenso de gradiente. Cuando el cambio se encuentre bajo un determinado umbral, decidiremos que la ruta ya no va a cambiar pararemos el descenso de gradiente.

Con esto habremos terminado con la lógica del probabilistic roadmap.

extended\_path: list[tuple[float, float]] = path.copy()

if additional\_smoothing\_points > 0:

for i, point in enumerate(path[:-1]):

x\_start, y\_start = point

x\_end, y\_end = path[i + 1]

x\_diff = (x\_end - x\_start) / (additional\_smoothing\_points + 1)

y\_diff = (y\_end - y\_start) / (additional\_smoothing\_points + 1)

for j in range(1, additional\_smoothing\_points + 1):

extended\_path.insert(

i \* (additional\_smoothing\_points + 1) + j,

(x\_start + j \* x\_diff, y\_start + j \* y\_diff),

)

smoothed\_path: list[tuple[float, float]] = extended\_path.copy()

change = float("inf")

while change >= tolerance:

change = 0.0

new\_smoothed\_path = smoothed\_path.copy()

for i in range(1, len(smoothed\_path) - 1):

x = smoothed\_path[i][0]

x += data\_weight \* (extended\_path[i][0] - smoothed\_path[i][0])

x += smooth\_weight \* (

smoothed\_path[i - 1][0] + smoothed\_path[i + 1][0] - 2 \* smoothed\_path[i][0]

)

y = smoothed\_path[i][1]

y += data\_weight \* (extended\_path[i][1] - smoothed\_path[i][1])

y += smooth\_weight \* (

smoothed\_path[i - 1][1] + smoothed\_path[i + 1][1] - 2 \* smoothed\_path[i][1]

)

new\_smoothed\_path[i] = (x, y)

change += abs(new\_smoothed\_path[i][0] - smoothed\_path[i][0]) + abs(new\_smoothed\_path[i][1] - smoothed\_path[i][1])

smoothed\_path = new\_smoothed\_path

return smoothed\_path

# 6. Me encanta que los planes salgan bien

En este apartado simplemente crearemos el publicador en el “prm\_node” para publicar el camino encontrado por el probabilistic roadmap. Este se publicará como un mensaje de tipo “Path”, que contiene el camino como una lista de nodos en formato PoseStamped.

self.\_path\_publisher = self.create\_publisher(Path, "/path", 10)

path\_msg = Path()

path\_msg.header.stamp = self.get\_clock().now().to\_msg()

path\_msg.poses = []

for point in path:

pose\_stamped = PoseStamped()

pose\_stamped.pose.position.x = point[0]

pose\_stamped.pose.position.y = point[1]

path\_msg.poses.append(pose\_stamped)

 self.\_path\_publisher.publish(path\_msg)

# 7. Análisis de sensibilidad del PRM

Uno de los hiperparámetros más importantes del PRM es el “use\_grid”, que define si utilizamos una rejilla o una distribución aleatoria para generar los puntos. La primera será más segura y garantizará una cobertura equilibrada de todo el mapa. La segunda tiene la probabilidad de dejar alguna zona del mapa inaccesible, pero a su vez es capaz de generar rutas más óptimas si tenemos suerte o generamos un número muy alto de nodos.

Este es otro hiperparámetro, “node\_count”, que indica cuantos nodos vamos a generar en el caso de que utilicemos la generación aleatoria. Un número mayor de nodos asegurará una mejor cobertura del mapa y permitirá encontrar rutas más óptimas, pero aumentará el coste computacional de generar el grafo.

En el caso de generar puntos con una rejilla, tenemos el parámetro “grid\_size”, que indica la distancia entre puntos consecutivos en la cuadrícula. Cuanto más bajo sea, más puntos generaremos, lo que tendrá las mismas consecuencias que aumentar “node\_count” en la generación aleatoria.

Otro parámetro es el de “connection\_distance”, que define hasta qué distancia los nodos pueden conectarse con una línea recta. Un valor bajo generará un grafo poco conectado, por lo que pueden aparecer zonas desconectadas. Un valor alto, además de evitar que haya zonas desconectadas, permite encontrar rutas más óptimas al haber más caminos disponibles.

También tenemos “obstacle\_safety\_distance”, que expande los obstáculos en el mapa para evitar que el robot planee rutas muy cerca de ellos. Un valor bajo permite rutas más ajustadas y óptimas, pero aumenta el riesgo de chocarse con un obstáculo. Un valor más alto generará rutas más seguras, pero puede hacer que ciertas rutas sean imposibles si reducimos demasiado el espacio navegable.

Finalmente, tenemos los hiperparámetros de suavizado de rutas: “data\_weight”, que indica cuánto peso tiene la condición de que la ruta se parezca a la original; “smooth\_weight”, que indica cuando peso tiene la condición de que la ruta esté más suavizada (los puntos estén más cerca unos de otros); y “additional\_smoothing\_points”, que indica cuántos puntos intermedios se añaden entre nodos. Cuanto más suavizado, las trayectorias generadas tendrán menos cambios bruscos de dirección. Sin embargo, si suavizamos demasiado podemos alejarnos mucho de la ruta óptima encontrada por el algoritmo.

# 8. ¿Quién conoce la ruta?

En el “pure\_pursuit\_node” crearemos un suscriptor que reciba este camino suavizado que estamos publicando. Al recibirlo convertirá el camino en una lista de tuplas para trabajar con los nodos más fácilmente.

path = []

for pose in path\_msg.poses:

x = pose.pose.position.x

y = pose.pose.position.y

path.append((x, y))

self.\_pure\_pursuit.path = path

# 9. Encontrando el rastro

La función “find\_closest\_point” se encargará de encontrar el nodo de la ruta mas cercano a la posición estimada del robot por el filtro de partículas.

point = (x, y)

closest\_idx = min(

range(len(self.\_path)), key=lambda i: self.calculate\_dis(point, self.\_path[i])

)

closest\_xy = self.\_path[closest\_idx]

return closest\_xy, closest\_idx

# 10. ¡Tierra a la vista!

En este apartado implementaremos el método “find\_target\_point” para obtener el punto de la ruta que se encuentra a la distancia de anticipación (lookahead). En caso de no encontrar ningún punto a esa distancia, elegiremos el siguiente más cercano que esté a más distancia que la lookahead.

path\_cut = self.\_path[origin\_idx:]

points\_outside\_radius = [

point

for point in path\_cut

if (self.calculate\_dis(origin\_xy, point) - self.\_lookahead\_distance) >= 0

]

if points\_outside\_radius:

target\_xy: tuple[float, float] = points\_outside\_radius[0]

else:

target\_xy = self.\_path[-1]

if origin\_xy == target\_xy:

return None

return target\_xy

# 

# 11. ¡Que empiece la carrera!

En este apartado calcularemos las velocidades lineal y angular que deberá tener el robot para seguir la ruta según las fórmulas de persecución pura. En este caso, hemos decidido fijar la velocidad a un valor constante y ajustar solamente la velocidad angular en base a la fórmula. Para evitar ángulos demasiado grandes, giraremos el robot en el sitio hasta que su ángulo respecto al siguiente nodo objetivo sea menor a un umbral.

if not self.\_path:

return 0.0, 0.0

closest\_point, closest\_point\_idx = self.\_find\_closest\_point(x, y)

destination = self.\_find\_target\_point(closest\_point, closest\_point\_idx)

if destination:

point = (x, y)

real\_l = self.calculate\_dis(point, destination)

if real\_l == 0:

real\_l += 1e-6

dx = destination[0] - x

dy = destination[1] - y

beta = math.atan2(dy, dx)

alpha = beta - theta

if alpha > math.pi / 4:

v = 0.0

w = -0.5

elif alpha < - math.pi / 4:

v = 0.0

w = 0.5

else:

v = 0.15

w = 2 \* v \* math.sin(alpha) / real\_l

else:

v, w = 0.0, 0.0

return v, w

# 12. Sensibilidad de la persecución pura

En este apartado analizaremos el efecto del valor de la distancia de anticipación en la persecución pura. Esta es la distancia a la que buscaremos el siguiente punto de nuestra ruta, lo que afectará también al cálculo de la velocidad angular de nuestro robot. Una distancia de anticipación pequeña tiene la desventaja de que el robot va a oscilar mucho al seguir la trayectoria, mientras que una distancia más larga suavizará la ruta. Sin embargo, como hemos explicado en el apartado 7, suavizar demasiado la ruta puede hacer que nos desviemos demasiado de la ruta óptima encontrada por el algoritmo. En general, querremos una distancia de anticipación grande cuando vayamos a una velocidad lineal alta, para evitar oscilaciones muy bruscas. A bajas velocidades podemos permitirnos una distancia de anticipación más baja para mantener precisión en el seguimiento de la ruta óptima.

# 13. Azul, rosa, azul, rosa...

En este último apartado simplemente crearemos un suscriptor en la clase “wall\_follower\_node” que se suscriba al tema “/pose” si el parámetro “localization” es verdadero. Esto nos permitirá que el robot funcione utilizando nuestro nuevo código sin interferir con los archivos de lanzamiento anteriores.

if enable\_localization:

self.\_subscribers.append(

message\_filters.Subscriber(self, PoseStamped, "/pose", qos\_profile=10)

)

# 14. Al andar se hace camino

Finalmente, transferiremos al robot el código de la simulación, teniendo que hacer las transformaciones adecuadas para que funcione según la lógica de nuestro robot real, que a estas alturas difiere bastante de el funcionamiento de nuestra simulación.

Una de las diferencias más importantes entre el robot y la simulación es que, mientras en nuestra simulación estamos estimando la pose constantemente, en el robot solo nos paramos a estimarla cada cierto tiempo. Sin embargo, el seguimiento de la ruta necesita tener una estimación de la pose del robot constante para poder encontrar el siguiente nodo al que tiene que ir y calcular la velocidad adecuadamente. Puesto que no podemos estar parando al robot cada paso que demos, utilizaremos una estimación de la pose que se calcula únicamente en base a la velocidad lineal y angular del robot, sin tener en cuenta las medidas del LiDAR. Esta pose se estará publicando constantemente para que el robot pueda seguir la trayectoria. Sin embargo, este cálculo imperfecto irá acumulando error, por lo que tendremos que hacer que se pare cada cierto tiempo a corregir la estimación de la pose gracias al LiDAR.

El código de estimación de la pose sin las medidas del LiDAR es:

z\_v: float = odom\_msg.twist.twist.linear.x

z\_w: float = odom\_msg.twist.twist.angular.z

if self.\_last\_pose and self.\_localized:

x\_old = self.\_last\_pose[0]

y\_old = self.\_last\_pose[1]

theta\_old = self.\_last\_pose[2]

theta = theta\_old + z\_w \* self.\_dt

theta\_mean = (theta + theta\_old) / 2

x = x\_old + z\_v \* math.cos(theta\_mean) \* self.\_dt

y = y\_old + z\_v \* math.sin(theta\_mean) \* self.\_dt

self.\_last\_pose = (x, y, theta)

self.\_publish\_pose\_estimate(x, y, theta)