Universidad de Chile
Facultad de Ciencias Físicas y Matemáticas
Departamento de Ingeniería Eléctrica

Tarea 4

Planificación

Integrantes: Sebastián Herrera T.
Profesores: Felipe Inostroza Ferrari
Javier Ruiz del Solar
Auxiliar: Francisco Leiva C.
Ayudantes: Eduardo Jorquera C.
Gonzalo Olguín M.

Fecha de entrega: 4 de julio de 2025
Santiago de Chile

Índice de Contenidos

Índice de Figuras

Índice de Códigos

1. Planificación global

1.1. Implementación RRT Planner

Para implementar RRT se debió completar varias funciones según las instrucciones proporcionadas en el código base, las funciones a completar fueron: `Node`, `RRTPlanner`, `reset_tree`, `sample_random_position`, `get_nearest_neighbour`, `get_new_position`, `check_for_collision`, `generate_rrt`, `recover_plan`, `plot_rrt`. Una vez completadas estas funciones, la clase `RRTPlanner` quedó:

Código 1: Clase Node y RRTPlanner implementada.

```
class Node:

    def __init__(self, x, y):
        self.position = (x,y)
        self.parent = []


class RRTPlanner:

    def __init__(self,
                 input_map,
                 init_position=None,
                 target_position=None,
                 nb_iterations=2000,
                 traverse_distance=2.0,
                 random_seed=0):

        # get map and its dimensions
        self._map  = input_map

        self._map_height, self._map_width = input_map.shape

        self._nb_iterations = nb_iterations
        self._traverse_distance = traverse_distance

        self._init_position = init_position

        if self._init_position is None:
            self._init_position = [self._map_width // 2, self._map_height // 2]

        self._target_position = target_position

        # Initialize tree
        self._tree = []
        self._tree.append(Node(self._init_position[0], self._init_position[1]))

        self._plan = None

        np.random.seed(random_seed)
```

```
40
41
42      def set_random_seed(self, random_seed):
43          np.random.seed(random_seed)
44          self.reset_tree()
45
46
47      def reset_tree(self):
48          self._tree = [Node(self._init_position[0], self._init_position[1])]
49          pass
50
51
52      def set_init_position(self, init_position):
53          self._init_position = init_position
54          self.reset_tree()
55
56
57      def set_target_position(self, target_position):
58          self._target_position = target_position
59
60
61      def sample_random_position(self):
62
63          sampled_x = np.random.uniform(0, self._map_width)
64          sampled_y = np.random.uniform(0, self._map_height)
65
66          return (sampled_x, sampled_y)
67
68
69      def get_nearest_neighbour(self, position):
70
71          min_distance = float('inf')
72          min_index = -1
73
74          for idx, node in enumerate(self._tree):
75              node_x, node_y = node.position
76              distance = np.linalg.norm(np.array(position) - np.array([node_x, node_y]))
77              if distance < min_distance:
78                  min_distance = distance
79                  min_index = idx
80
81          return min_index, self._tree[min_index]
82
83
84      def get_new_position(self, random_position, nearest_position):
85
86          dx = random_position[0] - nearest_position[0]
87          dy = random_position[1] - nearest_position[1]
88
89          angle = np.arctan2(dy, dx)
90
```

```
91          new_x = nearest_position[0] + self._traverse_distance * np.cos(angle)
92          new_y = nearest_position[1] + self._traverse_distance * np.sin(angle)
93
94          new_x = np.clip(new_x, 0, self._map_width - 1)
95          new_y = np.clip(new_y, 0, self._map_height - 1)
96
97          return new_x, new_y
98
99
100     def recover_plan(self):
101
102         plan = []
103
104         current_node = self._tree[-1]
105         while current_node.parent != []:
106             plan.append(list(current_node.position))
107             current_node = current_node.parent[1]
108
109         plan.append(list(self._init_position))  # agrega el nodo inicial
110         plan.reverse()
111         return np.array(plan)
112
113     def check_for_collision(self, new_position):
114
115         for i in [-2, 0, 2]:
116             for j in [-2, 0, 2]:
117                 x = int(np.clip(new_position[0] + i, 0, self._map_width - 1))
118                 y = int(np.clip(new_position[1] + j, 0, self._map_height - 1))
119                 if self._map[y, x] == 1:
120                     return True
121         return False
122
123
124     def generate_rrt(self):
125
126         for _ in range(self._nb_iterations):
127             random_position = self.sample_random_position()
128             nearest_idx, nearest_node = self.get_nearest_neighbour(random_position)
129             nearest_position = nearest_node.position
130             new_position = self.get_new_position(random_position, nearest_position)
131
132             if self.check_for_collision(new_position):
133                 continue
134
135             new_node = Node(new_position[0], new_position[1])
136             new_node.parent = [nearest_idx, nearest_node]
137             self._tree.append(new_node)
138
139             if self._target_position is not None:
140                 distance_to_target = np.linalg.norm(np.array(new_position) - np.array(self.
    ↪ _target_position))
```

```
            if distance_to_target <= self._traverse_distance:
                target_node = Node(self._target_position[0], self._target_position[1])
                target_node.parent = [len(self._tree) - 1, new_node]
                self._tree.append(target_node)
                self._plan = self.recover_plan()
                break

    return self._plan


def plot_rrt(self):

    positions = []
    edges = []

    for idx, node in enumerate(self._tree):
        positions.append(list(node.position))
        if node.parent != []:
            edges.append([node.parent[0], idx])

    positions = np.array(positions)
    edges = np.array(edges)

    x_pos = positions[:, 0]
    y_pos = positions[:, 1]

    # Plotting
    fig, ax = plt.subplots()

    ax.imshow(self._map, cmap='binary')
    ax.set_xlim((0, self._map_width))
    ax.set_ylim((0, self._map_height))

    ax.plot(x_pos[edges.T], y_pos[edges.T], color='C0')
    ax.scatter(positions[:,0], positions[:,1], s=20)

    if self._target_position is not None:
        ax.scatter(self._target_position[0], self._target_position[1], s=50)

    if self._plan is not None:
        ax.scatter(self._plan[:,0], self._plan[:,1], s=20)
        ax.plot(self._plan[:,0], self._plan[:,1], color='red')

    ax.scatter(self._init_position[0], self._init_position[1], s=50)

    ax.set_aspect('equal')
    plt.show()
```

1.2. Evaluación RRT

Para evaluar el desempeño del algoritmo RRT, se consideraron mapas binarios predeterminados. Cada mapa posee una distribución única de obstaculos, para cada prueba se variaron distintos parámetros. A continuación se desglozarán los parámetros de cada experimiento:

- **1a)** Mapa vacío 64x64, `traverse_distance` $= 2.0$, `nb_iterations` $\in [10, 100, 500]$, punto de inicio $(5, 5)$, punto de destino $(58, 58)$.
- **1b)** Mapa vacío 64x64, `traverse_distance` $\in [1.0, 2.0, 4.0]$, `nb_iterations` $= 2.0$, punto de inicio $(5, 5)$, punto de destino $(58, 58)$.
- **1c)** Se repiten los experimentos **1a)** y **1b)** con la diferencia que se realizará en un mapa vacío 64x128 y punto de destino $(120, 58)$.
- **1d)** Mapa con obstáculos Nº1 64x64, `traverse_distance` $= 2.0$, `nb_iterations` $= [500, 1000, 2000, 4000]$, punto de inicio $(5, 5)$, punto de destino $(58, 32)$.
- **1e)** Mapa con obstáculos Nº2 64x64, `traverse_distance` $= 2.0$, `nb_iterations` $= [500, 1000, 2000, 4000]$, punto de inicio $(2, 60)$, punto de destino $(60, 12)$.
- **1f)** Mapa con obstáculos Nº3 64x64, `traverse_distance` $= 2.0$, `nb_iterations` $= [5000, 10000, 20000]$, punto de inicio $(2, 60)$, punto de destino $(2, 2)$.

1.3. Resultados

1.3.1. 1a)

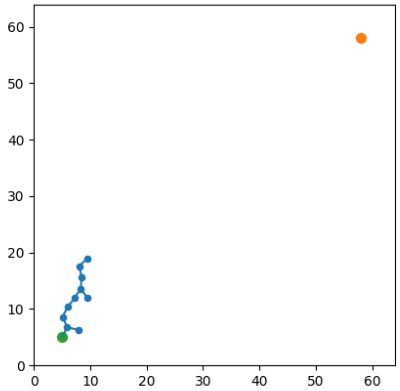


Figura 1: RRT 1a). nb_iterations = 10. Tiempo de ejecución: 0.004[s]

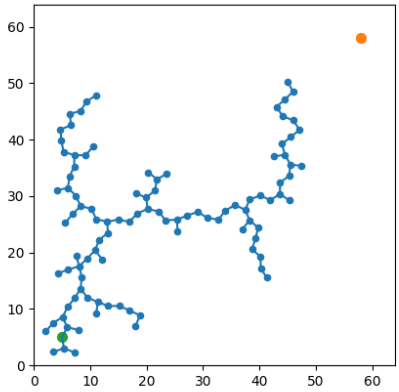


Figura 2: RRT 1a). nb_iterations = 100. Tiempo de ejecución: 0.07[s]

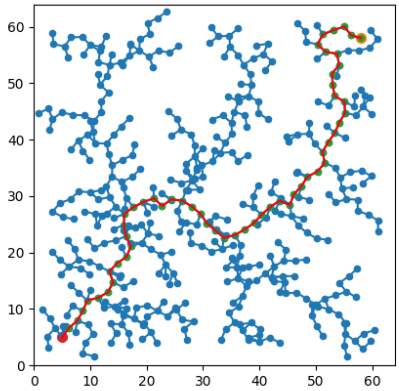


Figura 3: RRT 1a). nb_iterations = 500. Tiempo de ejecución: 0.42[s]

1.3.2. 1b)

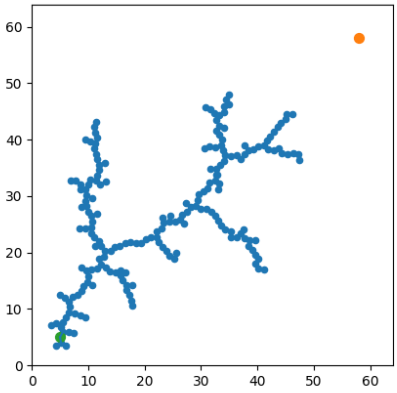


Figura 4: RRT 1b). traverse_distance = 1.0. Tiempo de ejecución: 0.18[s]

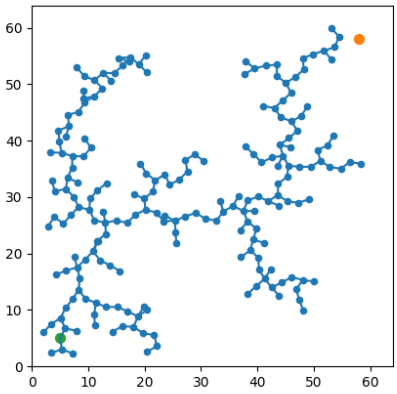


Figura 5: RRT 1b). traverse_distance = 2.0. Tiempo de ejecución: 0.2[s]

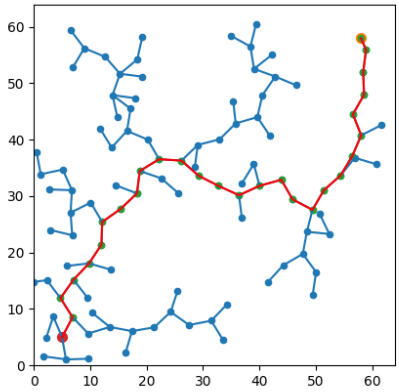
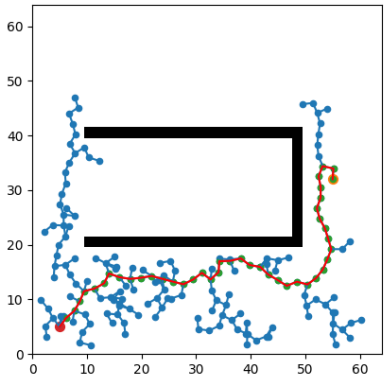


Figura 6: RRT 1b). traverse_distance = 4.0. Tiempo de ejecución: 0.13[s]

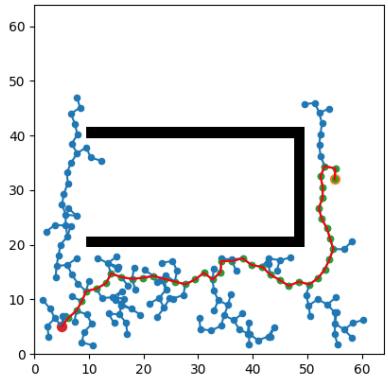
1.3.3. 1c)

Los resultados para el mapa 64x128 se encuentran en el anexo del presente informe.

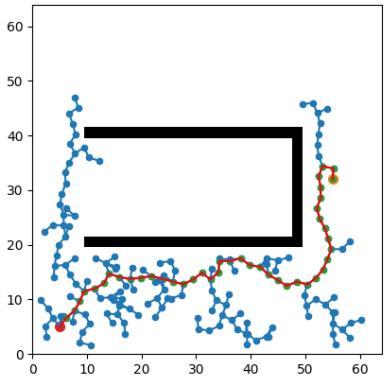
1.3.4. 1d)



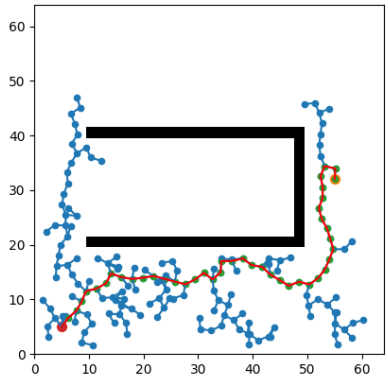
(a) nb_iterations = 500. Tiempo de ejecución: 0.14[s]



(b) nb_iterations = 1000. Tiempo de ejecución: 0.15[s]



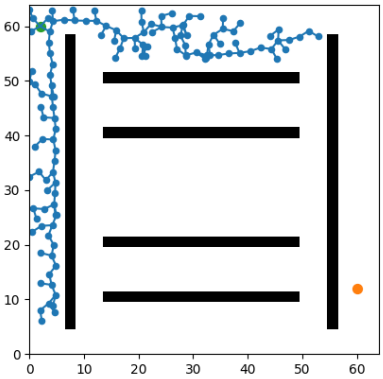
(c) nb_iterations = 2000. Tiempo de ejecución: 0.15[s]



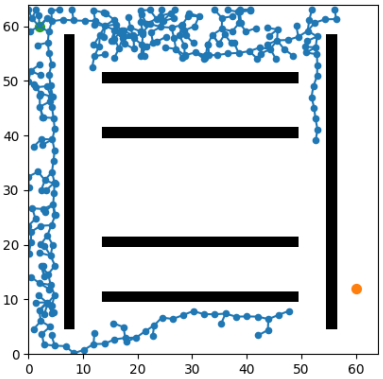
(d) nb_iterations = 4000. Tiempo de ejecución: 0.14[s]

Figura 7: Resultados para el experimento 1d)

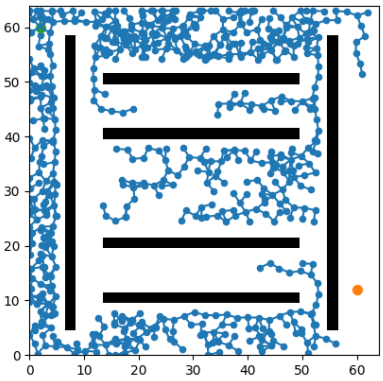
1.3.5. 1e)



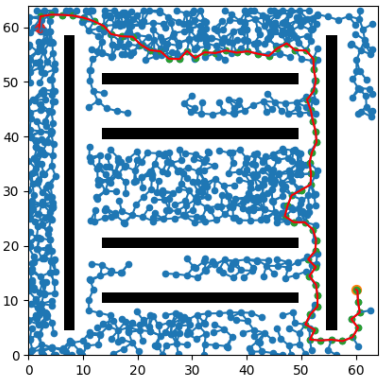
(a) nb_iterations = 500. Tiempo de ejecución: 0.29[s]



(b) nb_iterations = 1000. Tiempo de ejecución: 0.93[s]



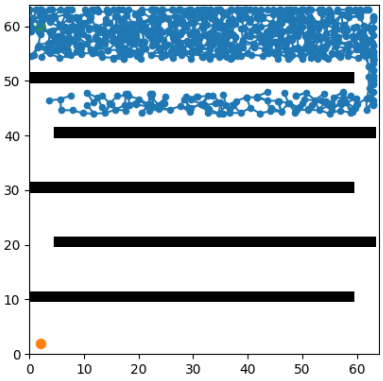
(c) nb_iterations = 2000. Tiempo de ejecución: 0.91[s]



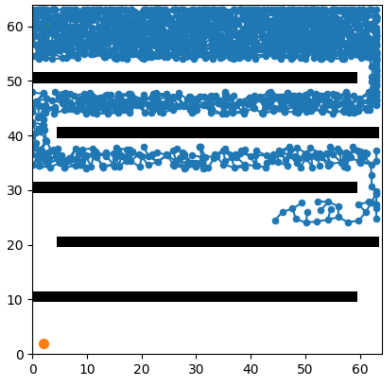
(d) nb_iterations = 4000. Tiempo de ejecución: 0.99[s]

Figura 8: Resultados para el experimento 1e)

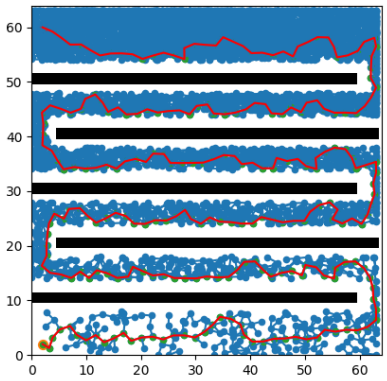
1.3.6. 1f)



(a) nb_iterations = 500. Tiempo de ejecución: 14.12[s]



(b) nb_iterations = 1000. Tiempo de ejecución: 62.42[s]



(c) nb_iterations = 2000. Tiempo de ejecución: 121.57[s]

Figura 9: Resultados para el experimento 1f)

El tiempo de ejecución del algoritmo crece de forma proporcional con el número de iteraciones permitidas en el RRT. En mapas vacíos, los tiempos fueron muy bajos incluso con altas iteraciones. Sin embargo, en mapas con obstáculos complejos, como el de la prueba **1f**, los tiempos se incrementaron significativamente. Por ejemplo, con 2000 iteraciones en el mapa **1f** el tiempo fue de 121.57 segundos, mientras que en mapas sin obstáculos o con obstáculos simples el tiempo fue inferior a 1 segundo.

Esto muestra que la complejidad del entorno influye drásticamente en el costo computacional, ya que en ambientes con muchos obstáculos el algoritmo necesita generar muchos nodos adicionales para evitar colisiones.

En mapas vacíos, el RRT genera trayectorias bastante directas y con pocas ramas innecesarias, especialmente cuando el número de iteraciones es alto. No obstante, en mapas con obstáculos, los planes tienden a ser más sinuosos o incluso incompletos si no se alcanzan suficientes iteraciones.

Se observa que al aumentar `traverse_distance`, las trayectorias son más directas pero menos capaces de rodear obstáculos con precisión. Por el contrario, valores bajos de `traverse_distance` permiten trayectorias más detalladas pero requieren más iteraciones para llegar al objetivo.

El algoritmo RRT, por diseño, realiza una búsqueda aleatoria guiada, expandiendo nodos desde el árbol inicial hasta alcanzar el objetivo. Su eficiencia depende directamente de:

- La distancia de expansión (traverse_distance).
- El número de iteraciones.
- La densidad de obstáculos en el mapa.

Además, la función `get_nearest_neighbour` realiza una búsqueda lineal por todos los nodos del árbol, lo cual impacta directamente en el tiempo de ejecución, especialmente en los casos con muchos nodos.

El principal cuello de botella es la función `get_nearest_neighbour`, ya que su implementación actual requiere recorrer todos los nodos del árbol para cada nueva muestra aleatoria, lo cual genera un tiempo de cómputo que crece linealmente con el tamaño del árbol. Esto se vuelve crítico en mapas con obstáculos complejos o cuando el número de iteraciones es alto, como en los casos **1e** y **1f**.

Una mejora directa sería reemplazar la búsqueda lineal por una estructura de datos eficiente, como un árbol **KD** (KD-Tree) o un árbol de búsqueda espacial. Estas estructuras permiten búsquedas de vecinos más rápidas (logarítmicas o sub-lineales) incluso con grandes cantidades de nodos, reduciendo significativamente el tiempo de ejecución.

Otra mejora complementaria sería ajustar dinámicamente el parámetro `traverse_distance` durante la ejecución para adaptarse mejor a entornos abiertos o congestionados.

2. Planificación local

En esta parte, se implementó el algoritmo *Pure Pursuit*.

2.1. Implementación Pure Pursuit

Nuevamente, se terminaron de implementar las funciones necesarias para el funcionamiento de las clases `DifferentialRobot` y `PurePursuitPlanner`:

Código 2: Métodos implementados para la clase DifferentialRobot.

```
    def set_pose(self, position, orientation=0):

        self._position = position
        self._orientation = orientation
        self._local_planner.set_pose(position, orientation)

    def set_map(self, input_map):
        self._map = input_map
        self._map_height, self._map_width = input_map.shape

    def step(self, action, dt=0.1):

        v_x, v_theta = action
        x, y = self._position
        theta = self._orientation

        new_x = x + v_x * np.cos(theta) * dt
        new_y = y + v_x * np.sin(theta) * dt
        new_theta = theta + v_theta * dt

        position = [new_x, new_y]
        orientation = new_theta

        if self.check_for_collision(position):
            return
        else:
            self.set_pose(position, orientation)
```

Código 3: Métodos y constructor implementados para la clase PurePursuitPlanner.

```
    def __init__(self,
                 kx=1.0,
                 look_ahead_dist=5.0):

        self._kx = kx
        self._look_ahead_dist = look_ahead_dist

        self._plan_idx = 0

        self._current_position = None
```

```
        self._current_orientation = None

        self._success = False
        self._dist_thresh = 1.0
    def get_local_pose(self, waypoint):

        dx = waypoint[0] - self._current_position[0]
        dy = waypoint[1] - self._current_position[1]

        theta = -self._current_orientation
        x_local = dx * np.cos(theta) - dy * np.sin(theta)
        y_local = dx * np.sin(theta) + dy * np.cos(theta)

        return [x_local, y_local]
    def get_waypoint(self):
        for i in range(self._plan_idx, len(self._plan)):
            dist = np.linalg.norm(np.array(self._current_position) - np.array(self._plan[i]))
            if dist <= self._look_ahead_dist:
                if i + 1 < len(self._plan):
                    self._plan_idx = i + 1
                break

        if self._plan_idx == len(self._plan) - 1:
            if np.linalg.norm(np.array(self._current_position) - np.array(self._plan[self._plan_idx])) <=
 ↪  self._dist_thresh:
                self._success = True

        return self._plan[self._plan_idx]
    def get_ctrl_cmd(self):
        waypoint = self.get_waypoint()
        local_position = self.get_local_pose(waypoint)

        x = local_position[0]
        y = local_position[1]

        linear_vel = np.clip(self._kx * x, self._min_linear_vel, self._max_linear_vel)
        angular_vel = np.clip(2 * y / (self._look_ahead_dist ** 2), self._min_angular_vel, self.
 ↪ _max_angular_vel)

        return [linear_vel, angular_vel]
```

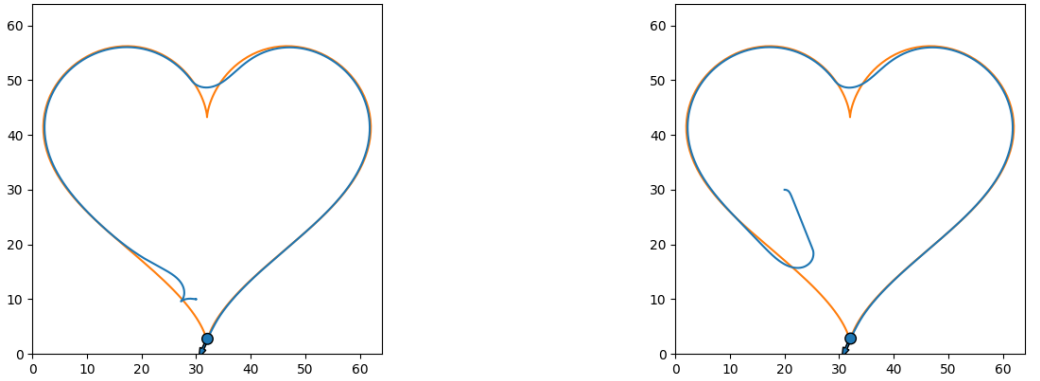
2.2. Evaluación Pure Pursuit

Para evaluar el desempeño de Pure Pursuit, se utilizó un camino geométrico predefinido. El robot diferencial debió seguir este camino con distintos parámetros presentados a continuación:

- **2a)** $kx = 1$, `look_ahead_distance`$= 5.0$. Pose inicial $\in [(30, 10), (20, 30)]$.
- **2b)** $kx = 1$, `look_ahead_distance`$= 1.0$. Pose inicial $\in [(30, 10), (20, 30)]$.
- **2c)** $kx = 1$, `look_ahead_distance`$= 10.0$. Pose inicial $\in [(30, 10), (20, 30)]$.
- **2d)** Se repiten los experimentos **2a)**, **2b)**, **2c)** pero con la velocidad angular limitada al rango $[-0.1, 0.1]$.

2.3. Resultados

2a)

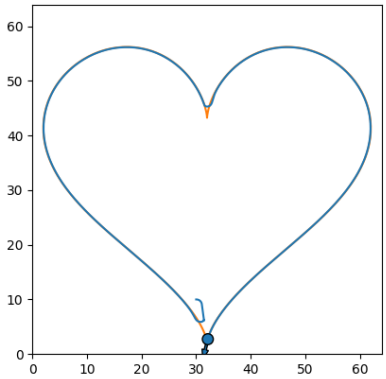


(a) Resultados Pose Inicial $(30, 10)$.

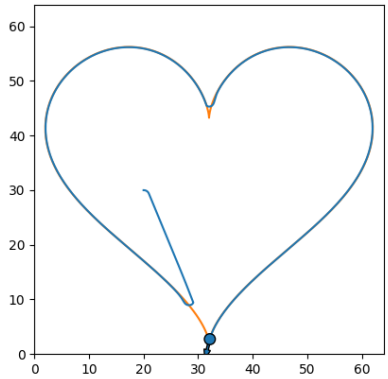
(b) Resultados Pose Inicial $(20, 30)$.

Figura 10: Resultados para el experimento 2a) `look_ahead_distance`$= 5.0$.

2b)



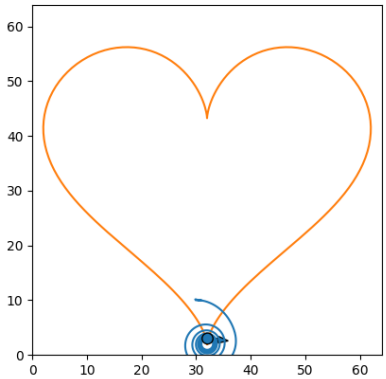
(a) Resultados Pose Inicial $(30, 10)$.



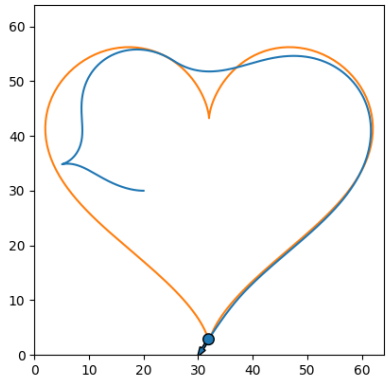
(b) Resultados Pose Inicial $(20, 30)$.

Figura 11: Resultados para el experimento 2b) `look_ahead_distance`= 1.0

2c)



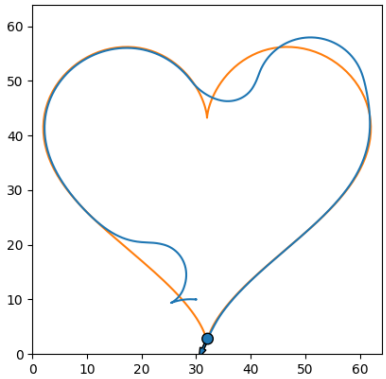
(a) Resultados Pose Inicial $(30, 10)$.



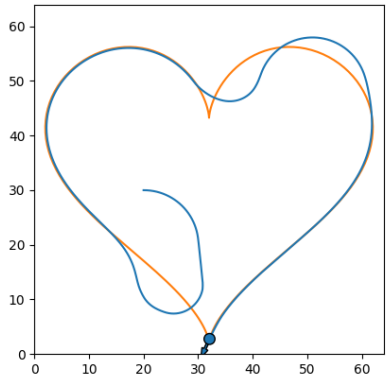
(b) Resultados Pose Inicial $(20, 30)$.

Figura 12: Resultados para el experimento 2c) `look_ahead_distance`= 10.0

2d)

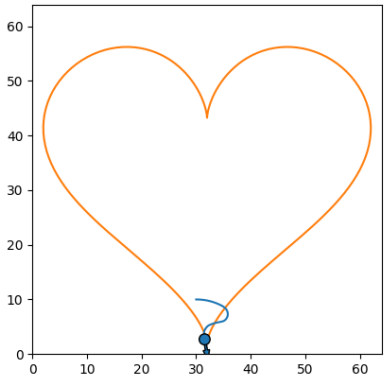


(a) Resultados Pose Inicial $(30, 10)$.

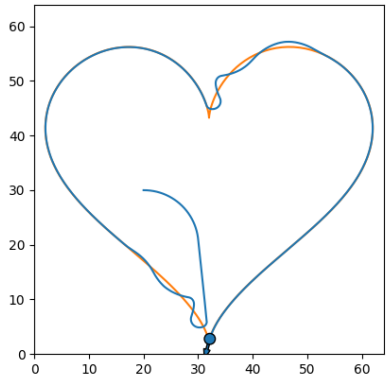


(b) Resultados Pose Inicial $(20, 30)$.

Figura 13: Resultados para el experimento 2d) `look_ahead_distance`= 5.0.

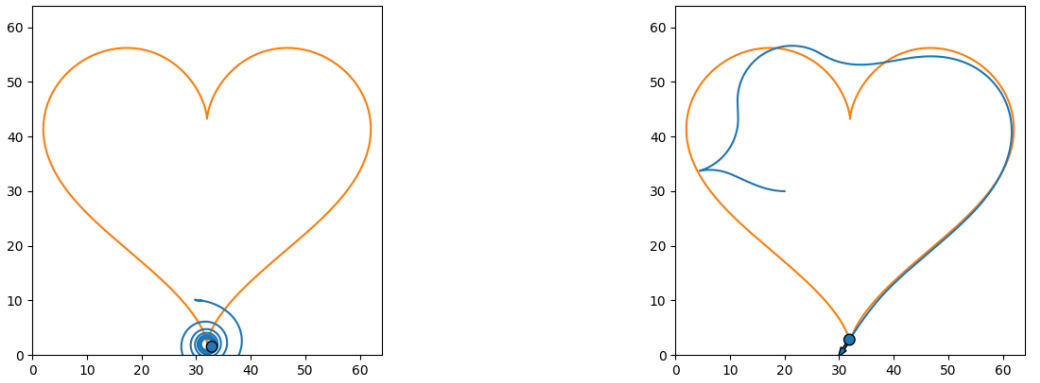


(a) Resultados Pose Inicial $(30, 10)$.



(b) Resultados Pose Inicial $(20, 30)$.

Figura 14: Resultados para el experimento 2d) `look_ahead_distance`= 1.0



(a) Resultados Pose Inicial $(30, 10)$.

(b) Resultados Pose Inicial $(20, 30)$.

Figura 15: Resultados para el experimento 2d) `look_ahead_distance`= 10.0

Efecto del parámetro `look_ahead_distance` sin límite de velocidad angular

Experimento 2a (`look_ahead_distance` = 5.0)

- En ambas posiciones iniciales el robot siguió la trayectoria de forma razonablemente precisa, con una leve impresición al realizar la curva del centro de la forma geométrica. La única diferencia es que con la pose inicial **(30,10)**, el robot retrocedió en ángulo para acercarse a la trayectoria deseada. En cambio, en el otro caso, el robot avanzó directamente para encontrarse con un punto cercano de la trayectoria.

Experimento 2b (`look_ahead_distance` = 1.0)

- El robot siguió la trayectoria con alta precisión en ambos casos, adaptándose muy bien a las curvas. Sin embargo, presentó movimientos más bruscos y oscilaciones debido a las correcciones constantes hacia waypoints cercanos. Además, para el caso de la pose inicial **(30,10)**, el robot ya no retrocede en ángulo sino que avanza "derecho" hacia el punto más cercano de la trayectoria. En el otro caso se repite este mismo comportamiento.

Experimento 2c (`look_ahead_distance` = 10.0)

- **Posición inicial (30,10):** El robot no logró seguir la trayectoria. Comenzó a describir un movimiento en espiral alrededor de su posición inicial, con cada vuelta acercándose más al punto de destino, pero sin seguir el camino planificado. El espacio entre las vueltas fue disminuyendo progresivamente, generando una trayectoria cíclica ineficiente. Este comportamiento se debe a la combinación de un valor excesivo de `look_ahead_distance`, una orientación inicial desfavorable y la geometría curva del trayecto.

- **Posición inicial (20,30):** El robot completó la trayectoria, pero con grandes desviaciones. Siguió el camino de forma muy inexacta y alejándose significativamente del plan original, al menos durante la mayor parte de la trayectoria, luego en el último tercio de trayectoria el robot sigue de forma aceptable la trayectoria propuesta hasta llegar al final destinado.

Efecto del límite de velocidad angular ($[-0.1, 0.1]$)

Experimento 2a (`look_ahead_distance` = 5.0)

- **Posición inicial (30,10):** El robot logró completar la trayectoria, aunque con una trayectoria más desviada y movimientos menos suaves en comparación con el caso sin límite angular. La restricción limitó la capacidad de corrección, generando errores moderados de tracking, pero sin impedir que el robot alcanzara el objetivo.
- **Posición inicial (20,30):** Se observó un comportamiento similar. El robot completó el recorrido, pero con una trayectoria más amplia y con mayor error acumulado en las curvas, aunque manteniendo la viabilidad del seguimiento.

Experimento 2b (`look_ahead_distance` = 1.0)

- **Posición inicial (30,10):** El robot fue incapaz de seguir la trayectoria. Se dirigió inmediatamente al punto de destino, ignorando totalmente la trayectoria designada.
- **Posición inicial (20,30):** El robot en este caso sí logra volver a la trayectoria designada, pero se presentan oscilaciones considerables para estabilizar su seguimiento de la trayectoria después de enfrentarse a curvas pronunciada. No obstante, el robot tuvo un desempeño bastante decente, logrando llegar al punto de destino sin mayores problemas.

Experimento 2c (`look_ahead_distance` = 10.0)

- **Posición inicial (30,10):** El robot repitió el movimiento en espiral observado sin límite de velocidad. La restricción angular no fue el factor determinante, aunque se debe mencionar que la espiral descrita es un poco más espaciada que en el caso sin límite de velocidad; el problema radica en el valor excesivo de `look_ahead_distance` y la orientación inicial desfavorable.
- **Posición inicial (20,30):** Tuvo un comportamiento bastante similar al caso sin limite de velocidad, sólo cambiaron levemente los primeros movimientos del robot, el seguimiento descrito es casi igual al caso sin limite.

Los resultados muestran que valores bajos de `look_ahead_distance` (1.0) permiten al robot seguir la trayectoria con alta precisión, adaptándose muy bien a las curvas. Sin embargo, esto conlleva movimientos más bruscos y oscilaciones, especialmente en trayectorias con curvas cerradas o cambios de dirección abruptos. El robot tiende a corregir constantemente su rumbo, lo que puede afectar la suavidad del desplazamiento.

Por su parte, un valor medio (5.0) ofrece un buen compromiso entre precisión y estabilidad. El robot logra completar la trayectoria en ambos puntos de partida, con un seguimiento relativamente suave y errores de tracking moderados. Este valor se comporta como un punto de equilibrio razonable para trayectorias de dificultad media.

En contraste, un valor alto (10.0) genera problemas severos. En la posición inicial $(30, 10)$, el robot quedó atrapado en un movimiento en espiral, sin capacidad de seguir la trayectoria planificada. Aunque el robot se fue acercando al punto de destino en cada vuelta, lo hacía de forma cíclica y sin respetar el plan original. En la posición inicial $(20, 30)$, si bien el robot logró completar la trayectoria, lo hizo con grandes desviaciones en los primeros tramos, tomando atajos y alejándose significativamente del camino. Solo en la última parte del recorrido consiguió ajustarse adecuadamente al plan.

La introducción de un límite angular ($[-0.1, 0.1]$) tuvo efectos variados dependiendo de la configuración de `look_ahead_distance`:

- Con valores medios (5.0), el robot mantuvo la capacidad de completar la trayectoria, aunque con trayectorias más amplias, movimientos menos suaves y mayor error acumulado, especialmente en las curvas.
- Con valores bajos (1.0), el impacto fue más severo. En algunas situaciones, el robot fue incapaz de seguir la trayectoria, desviándose de forma drástica o incluso ignorando completamente el plan para dirigirse directamente al punto final. Sin embargo, en otras configuraciones más favorables, el robot logró retomar el camino tras presentar oscilaciones significativas.
- Con valores altos (10.0), el límite angular no alteró sustancialmente el comportamiento. El robot continuó mostrando el mismo patrón de movimiento en espiral o desviaciones amplias, lo que indica que el problema principal en estos casos proviene del parámetro de control y no de las restricciones físicas del sistema.

Los experimentos evidencian que la selección del parámetro `look_ahead_distance` debe realizarse cuidadosamente, considerando:

- La geometría de la trayectoria a seguir.
- La orientación inicial del robot con respecto al plan.
- Las restricciones físicas, como la velocidad angular máxima.

Valores demasiado bajos generan movimientos bruscos y requieren alta capacidad de maniobra, mientras que valores excesivamente altos pueden provocar bloqueos o trayectorias altamente ineficientes.

Por lo tanto, para obtener un desempeño robusto en el algoritmo Pure Pursuit, es necesario:

- Ajustar el parámetro `look_ahead_distance` de acuerdo a la complejidad del entorno y las capacidades del robot.
- Realizar pruebas sistemáticas en simulaciones para identificar posibles fallos antes de implementar el sistema en entornos reales.
- Evaluar la combinación entre parámetros de control y límites físicos para asegurar la viabilidad y seguridad del seguimiento de trayectorias.

Pure Pursuit, aunque es un método efectivo, puede presentar fallos críticos si sus parámetros no se ajustan adecuadamente al contexto operativo del robot.

3. Planificación global y local

Finalmente, es momento de integrar los algoritmos desarrollados en las secciones anteriores para tener un planificador robusto.

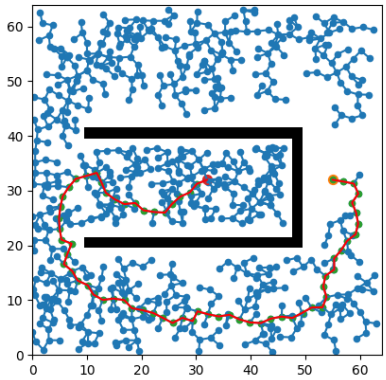
4. Experimentación

Para realizar esto, se fijaron los parámetros `nb_iterations` = 2000 y `traverse_distance`= 2.0 para el RRT, y `kx`= 1 para PurePursuit. Se experimentará en 3 mapas con distintos obstáculos dónde solo se variará el parámetro `look_ahead_distance` perteneciente al PurePursuit:

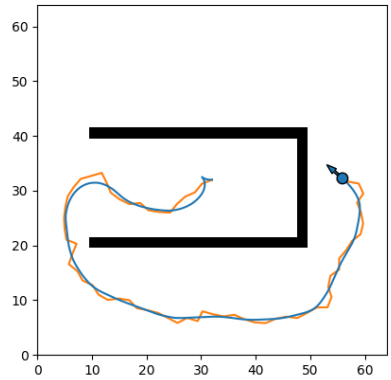
- **3a)** Mapa Nº1 `look_ahead_distance`$\in [1.0, 5.0, 10.0]$, Punto de inicio $(32, 32)$, Punto de destino $(55, 32)$.
- **3b)** Mapa Nº2 `look_ahead_distance`$\in [1.0, 5.0, 10.0]$, Punto de inicio $(2, 60)$, Punto de destino $(60, 12)$.
- **3c)** Mapa Nº3 `look_ahead_distance`$\in [1.0, 5.0, 10.0]$, Punto de inicio $(2, 60)$, Punto de destino $(2, 2)$.

4.1. Resultados

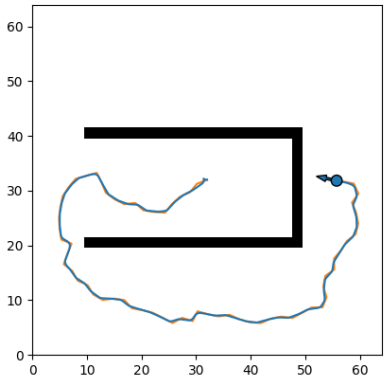
3a)



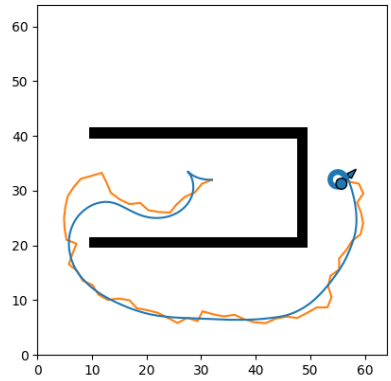
(a) Resultados RRT.



(b) Resultados PurePursuit look_ahead_distance 5.0.



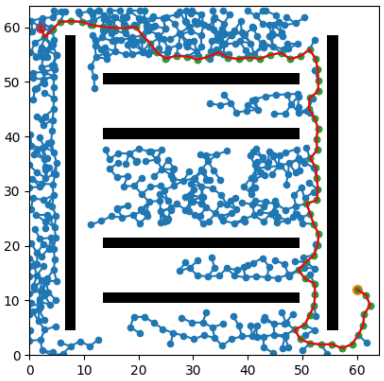
(c) Resultados PurePursuit look_ahead_distance 1.0.



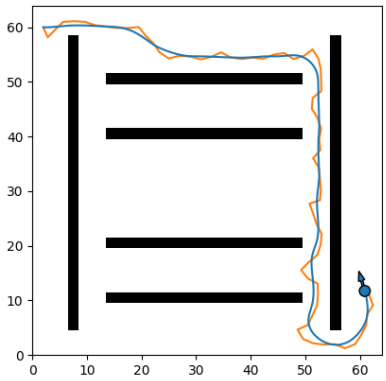
(d) Resultados PurePursuit look_ahead_distance 10.0.

Figura 16: Resultados para el experimento 3a) Mapa Nº1.

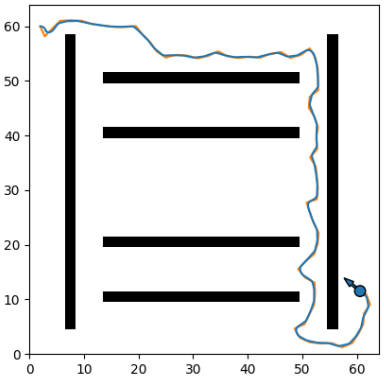
3b)



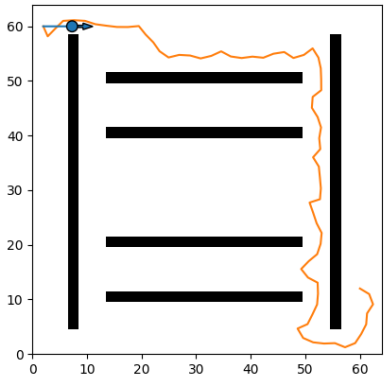
(a) Resultados RRT.



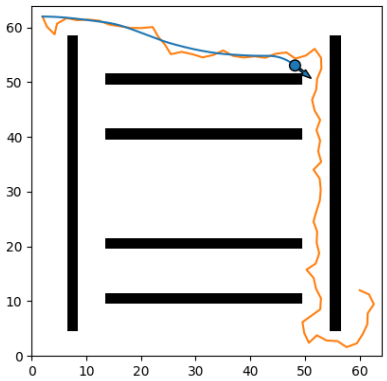
(b) Resultados PurePursuit look_ahead_distance 5.0.



(c) Resultados PurePursuit look_ahead_distance 1.0.



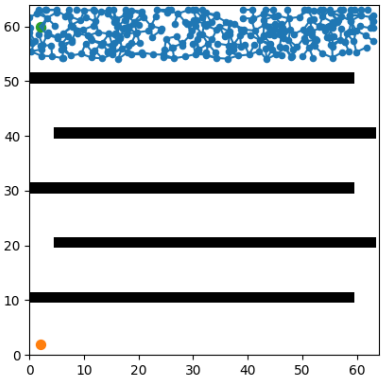
(d) Resultados PurePursuit look_ahead_distance 10.0.



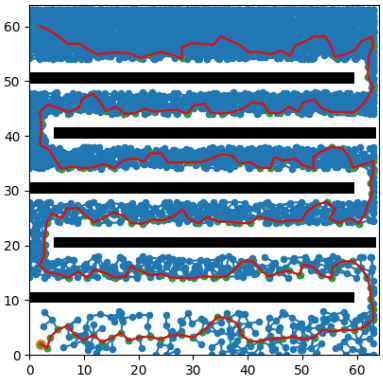
(e) Resultados PurePursuit look_ahead_distance 10.0.
Punto inicial cambiado a $(2, 62)$.

Figura 17: Resultados para el experimento 3b) Mapa N°2.

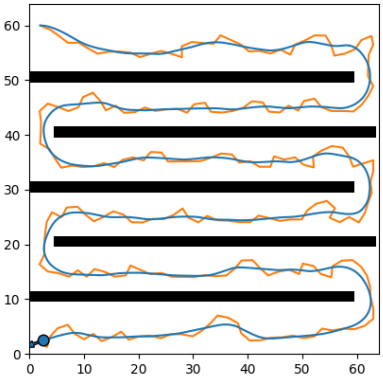
3c)



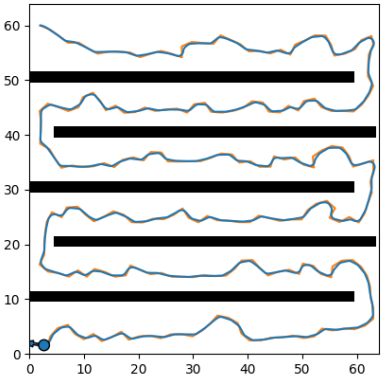
(a) Resultados RRT. No alcanzó su objetivo.



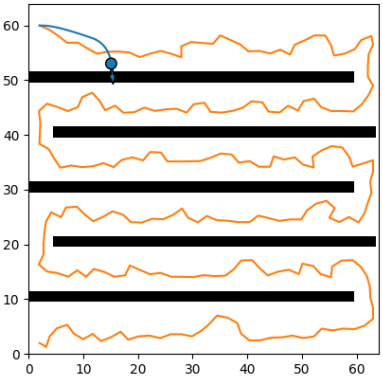
(b) Resultados RRT. Alcanzó su objetivo con `nb_iterations`= 20000.



(c) Resultados PurePursuit look_ahead_distance 5.0.



(d) Resultados PurePursuit look_ahead_distance 1.0.



(e) Resultados PurePursuit look_ahead_distance 10.0.

Figura 18: Resultados para el experimento 3c) Mapa N°3.

Los resultados obtenidos reflejan de forma clara las interacciones entre los algoritmos RRT y Pure Pursuit.

Desde el punto de vista de la planificación global, RRT fue capaz de generar trayectorias factibles en los mapas con obstáculos, aunque en algunos casos la calidad de los planes fue baja, con trayectorias muy irregulares o con numerosos giros bruscos. Esto coincide con los análisis previos, donde se observó que el RRT tiende a generar caminos con poca suavidad, especialmente en mapas complejos.

En cuanto al seguimiento local, se confirmaron los comportamientos observados previamente:

- Con valores bajos de `look_ahead_distance`, Pure Pursuit intentó seguir la trayectoria con alta precisión, pero generó movimientos bruscos e inestabilidad al enfrentarse a los giros cerrados generados por el RRT.
- Con valores medios, el robot logró un buen compromiso entre precisión y suavidad, el resultado es realmente suave, ignora los giros abruptos que produjo el RRT logrando que su camino sea bastante limpio.
- Con valores altos, el robot presentó trayectorias muy suavizadas, ignorando partes significativas del plan generado por RRT. En partes con cambios abruptos (como en **3a.d)**), se presentaron problemas para converger al punto final. En algunos casos el robot chocó al ser incapaz de seguir las curvas generadas por el RRT necesarias para llegar al punto de destino.

Esto demuestra que el parámetro `look_ahead_distance` debe ajustarse no solo según las capacidades del robot, sino también considerando la calidad y características del plan global generado.

La principal debilidad observada en el sistema integrado es la falta de cohesión entre la planificación global y la planificación local. El RRT genera trayectorias con giros bruscos y poco suaves, que luego son difíciles de seguir para Pure Pursuit, especialmente cuando se usan configuraciones que priorizan la suavidad del movimiento local.

Esto genera un desacoplamiento entre ambos niveles de planificación:

- Si el Pure Pursuit utiliza un `look_ahead_distance` pequeño, el robot puede seguir con precisión la trayectoria, pero tiende a oscilar debido a la brusquedad del plan, lo cual implicaría que el robot tenga las capacidades físicas para mantener este ritmo brusco.
- Si se utiliza un `look_ahead_distance` grande para suavizar el movimiento, el robot ignora partes importantes de la trayectoria, generando un seguimiento impreciso o incompleto.

Una solución hipotética a esta debilidad sería incorporar un proceso de **suavizado de trayectoria** entre la etapa de planificación global y la planificación local.

Este suavizado consistiría en aplicar un filtro o algoritmo de optimización que transforme la trayectoria generada por el RRT en un camino más suave y continuo, manteniendo la factibilidad respecto a los obstáculos.

Una posible técnica sería el **algoritmo de suavizado por splines**, donde se ajustan curvas suaves (por ejemplo, B-splines) a los puntos generados por el RRT. Otra alternativa sería utilizar técnicas como **shortcut smoothing**, que elimina segmentos innecesarios manteniendo la viabilidad del camino.

Este proceso de suavizado permitiría que el Pure Pursuit pueda seguir la trayectoria resultante de forma más efectiva, incluso con valores moderados o altos de `look_ahead_distance`, reduciendo así los conflictos entre precisión y suavidad.

Además, esta solución no requeriría modificar la lógica interna de RRT ni Pure Pursuit, ya que actúa como una capa intermedia entre ambos algoritmos.

En resumen, si bien la combinación de RRT y Pure Pursuit es efectiva en ciertos escenarios, su integración requiere un ajuste cuidadoso y, en muchos casos, la introducción de un proceso adicional de suavizado de trayectorias para garantizar un comportamiento robusto y eficiente en entornos reales.

1. Anexo

1.1. Resultados 1c)

1a) con mapa 64x128

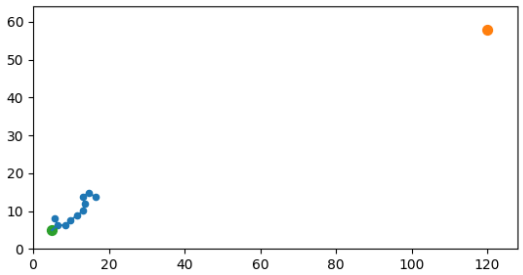


Figura 19: RRT 1c). nb_iterations = 10. Tiempo de ejecución: 0.004[s]

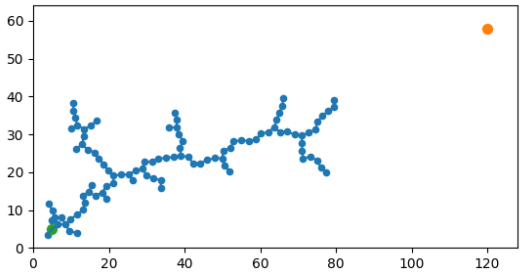


Figura 20: RRT 1c). nb_iterations = 100. Tiempo de ejecución: 0.08[s]

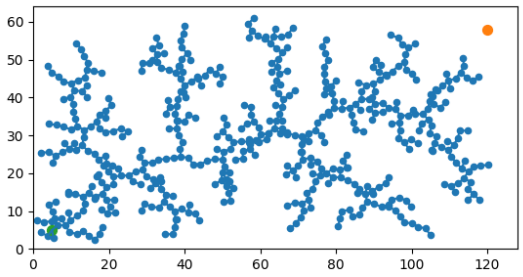


Figura 21: RRT 1a). nb_iterations = 500. Tiempo de ejecución: 0.99[s]

1b) con mapa 64x128

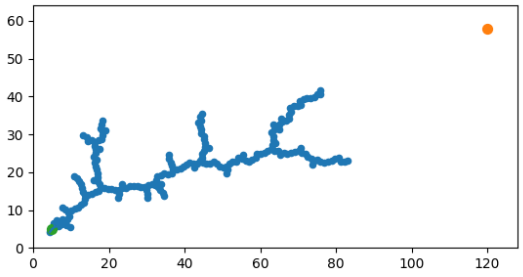


Figura 22: RRT 1c). traverse_distance = 1.0. Tiempo de ejecución: 0.2[s]

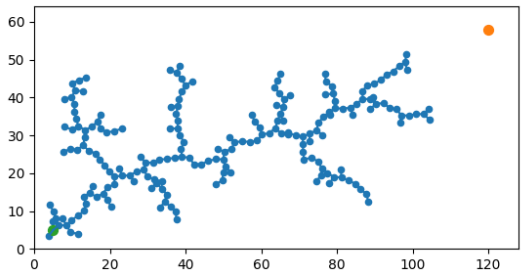


Figura 23: RRT 1c). traverse_distance = 2.0. Tiempo de ejecución: 0.2[s]

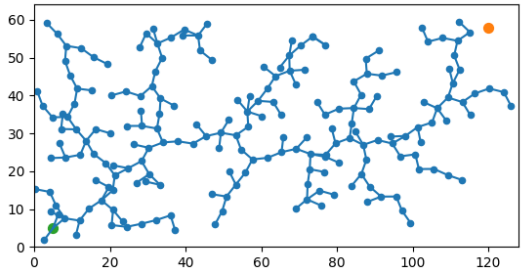


Figura 24: RRT 1c). traverse_distance = 4.0. Tiempo de ejecución: 0.198[s]