Práctica 3. Localización: Filtro de partículas

Javier Ahumada, Rodrigo Covas, Javier Escobar

Contenido

1. Lee el maldito manual

Pregunta #1: ¿Qué métodos de maps.py pueden ser útiles para completar el código solicitado en la Sección 3? ¿Para qué los utilizaría?

En primer lugar, tenemos la función "contains", que dado un punto con sus coordenadas x e y nos dice si el punto está contenido en el mapa. Esta función nos será útil por ejemplo al generar puntos aleatorios para solo quedarnos con los que se generen dentro del mapa.

También tenemos el método "check_collision", que dados dos puntos nos dice si el segmento que los une interseca de alguna manera con el mapa y nos devuelve el punto de intersección más cercano y la distancia a la que se encuentra del primer punto. Esta función nos será útil en primer lugar cuando queramos mover las partículas generadas para evitar que atraviesen las paredes y se salgan del mapa. En segundo lugar, nos servirá para detectar la distancia a la que se encuentran de cada partícula las paredes del mapa.

Pregunta #2: ¿Qué optimizaciones se han introducido en la clase Map para reducir los tiempos de ejecución de las Secciones 4 y 5?

En la sección 4 (TODO 3.5) utilizamos la función "check_collision" para que, al mover las partículas, si alguna va a atravesar las paredes se quede en su lugar en el punto de intersección como si se hubiera chocado con la pared. En la sección 5 (TODO 3.6) utilizamos la misma función para obtener las medidas predichas que debería devolver el LiDAR para cada partícula.

Para optimizar esta función, el mapa está dividido en regiones. Esto permite que la función "check_collision" solo compruebe las colisiones en la región donde se encuentra la partícula en lugar de comprobar con todas las paredes del mapa.

2. ¿Perdido o encontrado?

En esta sección simplemente tendremos que crear los publicadores y suscriptores para que el particle_filter_node publique en el tema /pose si el filtro de partículas ha convergido o no. Si el robot está localizado publicará también la estimación de la pose.

El siguiente código crea el publicador en el nodo particle_filter_node:

```
self._pose_publisher = self.create_publisher(PoseStamped, "/pose", 10)
```

El siguiente código publica el mensaje de tipo PoseStamped en el tema /pose. Si el robot no está localizado, el mensaje solo contendrá el header y el booleano que dice si el robot está localizado o no. En caso de estar localizado, el mensaje contendrá también la posición estimada en x y en y así como la orientación estimada del robot. Para esto último utilizaremos la función "euler2quat" para convertir la orientación en cuaterniones, que es lo que esperamos en un mensaje de tipo PoseStamped.

```
pose_msg = PoseStamped()
pose_msg.header.stamp = self.get_clock().now().to_msg()
if self._localized:
      pose_msg.pose.position.x = x_h
      pose_msg.pose.position.y = y_h

      q = euler2quat(0.0, 0.0, theta_h)
      pose_msg.pose.orientation.x = q[0]
      pose_msg.pose.orientation.y = q[1]
      pose_msg.pose.orientation.z = q[2]
      pose_msg.pose.orientation.w = q[3]
pose_msg.localized = self._localized

self._pose_publisher.publish(pose_msg)
```

El nodo de simulación "coppeliasim_node" se suscribirá a este tema sólo si el parámetro "enable_localization" es verdadero. Además, el tema se sincronizará con "cmd_vel" para llamar al "next_step_callback" con toda la información de velocidad y pose y poder calcular la simulación con mayor precisión:

```
self._subscribers: list[message_filters.Subscriber] = []
self.subscribers.append(
      message_filters.Subscriber(self, TwistStamped, "/cmd_vel",
      qos_profile=10)
)
if enable_localization:
      self._subscribers.append(
            message_filters.Subscriber(self, PoseStamped, "/pose",
      qos_profile=10)
                )
ts = message_filters.ApproximateTimeSynchronizer(
      self._subscribers, queue_size=10, slop=9
)
ts.registerCallback(self._next_step_callback)
```

3. No tengo ni idea de dónde estoy

En este apartado vamos a completar el método “init_particles” para generar partículas aleatorias válidas en el mapa y se almacene su pose. Como ya se ha explicado, para comprobar que una partícula generada es válida utilizaremos la función “contains” de la clase map para asegurarnos de que la partícula está dentro del espacio vacío alcanzable dentro del entorno.

Si se quiere resolver un problema de localización global, estas partículas se generarán según una uniforme a lo largo de todo el mapa, y se le asignará a cada partícula una orientación aleatoria entre las cuatro siguientes: 0, π/2, π, 3π/2; con el objetivo de reducir la carga computacional.

```
if global_localization:
	x_min, y_min, x_max, y_max = self._map.bounds()
	for i in range(particle_count):
		x: float
		y: float
		valid = False
		while not valid:
			x = random.uniform(x_min, x_max)
			y = random.uniform(y_min, y_max)
			valid = self._map.contains((x, y))
		theta = random.choice([0, math.pi / 2, math.pi, 3 * math.pi / 2])
		particles[i] = (x, y, theta)
```

Sin embargo, si se conoce la ubicación inicial aproximada del robot realizaremos el seguimiento de la pose. Para ello generaremos partículas en torno a la posición inicial aproximada del robot utilizando una gaussiana. En este caso, la orientación de las partículas también se generará con una gaussiana en lugar de elegir entre tan solo 4 opciones.

```
else:
	x_mean, y_mean, theta_mean = initial_pose
	x_sigma, y_sigma, theta_sigma = initial_pose_sigma
	for i in range(particle_count):
		x: float
		y: float
		valid = False
		while not valid:
			x = random.gauss(x_mean, x_sigma)
			y = random.gauss(y_mean, y_sigma)
			valid = self._map.contains((x, y))
		theta = random.gauss(theta_mean, theta_sigma)
	particles[i] = (x, y, theta)
```

4. Las partículas salen de excursión...y se chocan

En este apartado completaremos la función "move" del filtro de partículas. En ella moveremos todas las partículas siguiendo las velocidades lineal y angular del robot de acuerdo con sus respectivas orientaciones. A estas velocidades les añadiremos un ruido gaussiano con media cero para tener en cuenta la incertidumbre del movimiento y mejorar la convergencia del algoritmo.

Como hemos explicado antes, también utilizaremos la función "check_collision" para evitar que las partículas se salgan del mapa, y que en su lugar se queden en el punto de intersección con el mapa como si se hubieran chocado con la pared.

```
for i, particle in enumerate(self._particles):
	x, y, theta = particle
	x_old, y_old, theta_old = x, y, theta
	noise_v = random.gauss(v, self._sigma_v)
	noise_w = random.gauss(w, self._sigma_w)

	x += noise_v * math.cos(theta) * self._dt
	y += noise_v * math.sin(theta) * self._dt

	theta += noise_w * self._dt
	theta %= 2 * math.pi

	self._particles[i] = (x, y, theta)

	intersection, _ = self._map.check_collision([(x_old, y_old), (x, y)])
	if intersection:
		x_collision, y_collision = intersection
		self._particles[i] = (x_collision, y_collision, theta)
```

5. Echando un vistazo virtual alrededor

En este apartado completaremos la función "sense" que devuelve las medidas predichas para cada rayo del LiDAR dada la pose de una partícula. Para este propósito, utilizaremos la función "check_collision" trazando rayos desde la partícula en dirección a cada rayo del LiDAR (en realidad solo utilizaremos 16 rayos del LiDAR para reducir la carga computacional del algoritmo). Esta función nos devolverá el punto de intersección más cercano en la dirección de cada rayo, así como la distancia a este punto. Esta distancia será la medida predicha que debería medir ese rayo del LiDAR si el robot se encontrase donde nuestra partícula.

```
rays = self._lidar_rays(particle, tuple(range(0, 240, 15)))
for ray in rays:
      _, distance = self._map.check_collision(ray, True)
      if distance <= 1:  # self._sensor_range_max
            z_hat.append(distance)
      else:
            z_hat.append(float("nan"))
      return z_hat
```

6. Estoy de acuerdo en estar en desacuerdo

En este apartado completaremos la función "gaussian" para determinar la discrepancia entre las medidas experimentales obtenidas de los sensores y las predichas para una partícula. Esta función recibe una media (mu), que es la medida experimental; una desviación típica (sigma); y una variable (x), que es la medida predicha para una partícula.

```
return math.exp(-((mu - x) ** 2) / (2 * sigma**2)) / ((2 * math.pi * sigma**2) ** 0.5)
```

7. ¿Crees lo que ven tus ojos?

En este apartado completaremos el método "measurment_probability". Este método estima la verosimilitud de un conjunto de medidas dada la pose del robot. Para ello obtendremos las medidas predichas para la partícula llamando a la función "sense". Para gestionar las medidas con valor "nan", las sustituiremos por un valor un poco inferior a la medida mínima, para diferenciarlos de las verdaderas medidas con ese valor (ya que probablemente se encuentren aún más cerca). Luego llamaremos a la función "gaussian" anteriormente programada para determinar la discrepancia entre las medidas reales y virtuales. La probabilidad de la partícula será igual al producto de estas discrepancias.

```
measurements = [measure if not math.isnan(measure) else
self._sensor_range_min - 0.01 for measure in measurements]

sim_measurements = self._sense(particle)
sim_measurements = [measure if not math.isnan(measure) else
self._sensor_range_min - 0.01 for measure in sim_measurements]

for z, z_hat in zip(measurements[::15], sim_measurements):
      probability *= self._gaussian(z, self._sigma_z, z_hat)
return probability
```

8. Solo sobreviven los más aptos

En este apartado completaremos el método “resample”, que se encarga del remuestreo de partículas según las importancias obtenidas del método “measurment_probability” para generar un nuevo conjunto de partículas que aproximen mejor la pose del robot después de moverse y medir.

En primer lugar, calculamos la importancia de cada partícula con “measurment_probability”. Para facilitar trabajar con estas probabilidades tan bajas, que causan problemas de cálculo al ejecutar la simulación, realizamos distintas operaciones para ajustarlas.

Hemos decidido utilizar el muestreo sistemático. Para ello generamos un número aleatorio de una uniforme entre 0 y W/N (siendo N el número de partículas que estamos muestreando y W la suma de las importancias, en nuestro caso 1 al haber normalizado). El resto de posiciones se obtienen sumando a este número aleatorio el mismo número 1/N. Para seleccionar las partículas cogemos aquellas cuya importancia acumulada supera cada uno de estos valores generados, para lo que utilizamos la función “searchsorted” de numpy.

```
# Calculate measurement probabilities for each particle
raw_probabilities = np.array(
      [self._measurement_probability(measurements, particle) for
      particle in self._particles]
)

log_probabilities = np.log(raw_probabilities + 1e-300)
max_log_prob = np.max(log_probabilities)
log_probabilities -= (max_log_prob)
probabilities = np.exp(log_probabilities)

# Normalize the probabilities to sum to 1
probabilities /= np.clip(np.sum(probabilities), 1e-300, None)

# Compute the cumulative sum of the normalized probabilities
cumulative_sum = np.cumsum(probabilities)
cumulative_sum[-1] = 1.0

# Generate N uniform random values between 0 and 1/N
N = self._particle_count
start = np.random.uniform(0, 1 / N)
positions = start + np.arange(N) / N

indexes = np.searchsorted(cumulative_sum, positions)
indexes = np.clip(indexes, 0, len(self._particles) - 1)
self._particles = self._particles[indexes].copy()
```

9. La unión hace la fuerza

En este apartado completaremos el método “compute_pose”. Este método utiliza el algoritmo de agrupación DBSCAN de la librería Scikit-learn para devolver las agrupaciones de partículas que se encuentran a una distancia muy pequeña unas de otras. El número de partículas se ajustará según el número de agrupaciones que haya, manteniendo 20 veces el número de agrupaciones que haya o 50 partículas cuando solo queden 1 o 2 agrupaciones.

Si solo queda una agrupación diremos que el robot está localizado, y estimaremos la pose como una media de las posiciones de las partículas. Para calcular la orientación estimada convertiremos los ángulos en vectores del círculo unitario utilizando el seno y el coseno para poder hacer sus medias aritméticas y luego utilizaremos la función de “atan2” para realizar la arcotangente y obtener la orientación promedio correcta.

```
localized = False
pose = (float("inf"), float("inf"), float("inf"))

# Apply DBSCAN clustering
clustering = DBSCAN(eps=0.5, min_samples=10).fit(self._particles[:, :2])
clusters = clustering.labels_

# Get unique clusters excluding noise (-1)
unique_clusters = np.unique(clusters[clusters != -1])

# If there is at least one valid cluster
if len(unique_clusters) > 0:
	# Adjust the number of particles based on the number of clusters
	self._particle_count = max(50, len(unique_clusters) * 20)

	# If only one cluster remains, estimate the pose
	if len(unique_clusters) == 1:
		localized = True
		cluster_particles = self._particles[clusters ==
	unique_clusters[0]]

		# Compute mean (x, y)
		x_mean, y_mean = np.mean(cluster_particles[:, :2], axis=0)

		# Compute mean orientation (theta) using atan2 to handle
	periodicity
		theta_mean = np.arctan2(
			np.mean(np.array([
				math.sin(cluster_particle[2])
				for cluster_particle in cluster_particles
			])),
			np.mean(np.array([
				math.cos(cluster_particle[2])
				for cluster_particle in cluster_particles
			])),
		)

	pose = (x_mean, y_mean, theta_mean)

return localized, pose
```

10. Análisis de sensibilidad

Pregunta #3: ¿Cuál es el efecto de incrementar o disminuir el número de partículas a nivel de tiempos de computación y exactitud de la localización? ¿Cuál es el valor inicial más apropiado de este hiperparámetro para este entorno? ¿Qué estrategia de reducción de partículas funciona mejor?

Al incrementar el número de partículas aumentamos la exactitud de la localización, puesto que tenemos más hipótesis que comprobar sobre su posición. Sin embargo, aumentan a su vez los tiempos de computación al tener que calcular las importancias de muchas más partículas y tener que comprobar más partículas en el remuestreo.

En nuestro programa hemos reducido el número de partículas según el número de clusters que nos devuelve el algoritmo DBSCAN. Manteniendo un número de partículas igual a 20 veces el número de clústers, o 50 partículas cuando solo quedan 1 o 2 clústers. Esto hace que la próxima vez que realicemos el remuestreo nos quedemos solo con ese número de partículas.

Pregunta #4: ¿Qué sucede si modifica los parámetros de ruido de movimiento y medida?

Si disminuimos el ruido haremos que el algoritmo converja más rápido. Sin embargo, un ruido más alto puede ayudarnos a dispersar más las partículas, ayudando a que se comprueben más posibilidades en el mapa y evitemos un sesgo por culpa de no haber generado una partícula aleatoria justo donde estaba el robot.