Robótica Móvil
Implementación del Algoritmo *Split & Merge*

Iñaki Rañó

Notas

El objetivo de esta práctica es implementar el algoritmo de *Split & Merge* para la segmentación de barridos láser, i.e. obtener segmentos de líneas rectas a partir de nubes de puntos 2D. Estos segmentos se pueden utilizar en entornos de oficinas como *landmarks* para localización (p.e. combinados con un mapa CAD del entorno) o SLAM (p.e. construir un mapa geométrico con segmentos rectos), además de para extraer distancias y direcciones con paredes (p.e. para implementar comportamientos de seguimiento de paredes).

1 Introducción

El algoritmo de *Split & Merge* permite, dada una nube de puntos en coordenadas Cartesianas, obtener los extremos de segmentos rectos, i.e. segmentos lineales, existentes en el entorno. La entrada del algoritmo es un conjunto de puntos en coordenadas Cartesianas (x_i, y_i) con $i = 1, 2, \cdots, n$ ordenados (al estilo de un barrido de lidar o laser) y la salida es una lista de segmentos s_j definidos como pares de los puntos extremos. El algoritmo tiene dos parámetros a configurar, un parámetro de distancia (d_{th}) y un parámetro angular θ_{th}. Como su nombre indica el algoritmo tiene dos partes principales: un paso *Split* en el que segmentos candidatos se parten atendiendo al parámetro de distancia, y un paso *Merge* en el que segmentos candidatos se unen atendiendo al parámetro angular. Además de esto se suele implementar un tercer paso (*Purge*) que elimina segmentos cortos y/o segmentos que incluyan pocos puntos.

1.1 *Split & Merge*: *Split*

Dados los puntos de entrada $\mathbf{p}_i = (x_i, y_i)\ i = 1, 2, \cdots, n$ en coordenadas Cartesianas, el algoritmo empieza creando un segmento candidato con el primer y último punto, i.e. $s_1 = (\mathbf{p}_1, \mathbf{p}_n)$. Cualquier punto en dicho segmento $\mathbf{p} = (x, y)$ deberá cumplir la ecuación de la recta definida por estos puntos, i.e. $\frac{y_n - y_1}{x_n - x_1} = \frac{y - y_1}{x - x_1}$, que reorganizando términos puede escribirse como:

$$(y_n - y_1)(x - x_1) - (x_n - x_1)(y - y_1) = 0$$
$$(y_n - y_1)(x - x_1) + (x_1 - x_n)(y - y_1) = 0$$

y dividiendo ambas partes por $l = \sqrt{(x_1 - x_n)^2 + (y_n - y_1)^2}$ se puede escribir como $\hat{\mathbf{u}}_\perp \cdot (\mathbf{p} - \mathbf{p_1}) = 0$ donde $\hat{\mathbf{u}}_\perp = [\frac{y_n - y_1}{l}, -\frac{x_n - x_1}{l}]$ es un vector ortonormal a la recta definida por $\mathbf{p}_1$ y $\mathbf{p}_n$ (perpendicular y de norma unitaria) y '$\cdot$' representa el producto escalar. Esta ecuación tiene una sencilla interpretación geométrica, i.e los puntos de la recta son aquellos para los que

la proyección de $\mathbf{p} - \mathbf{p_1}$ en la dirección perpendicular a la recta es cero, i.e. la distancia (con signo) del punto a la recta es cero. Esta forma de la ecuación de una línea proporciona un mecanismo directo para calcular la distancia de un punto a una recta que pase por $\mathbf{p}_1$ y con vector ortogonal $\hat{\mathbf{u}}_\perp$, i.e. la distancia de $\mathbf{p}$ a la recta es $d = |\hat{\mathbf{u}}_\perp \cdot (\mathbf{p} - \mathbf{p_1})|$.

Volviendo a la parte *Split* del algoritmo, una vez definido el segmento candidato s_1 se busca el punto entre los índices 2 y $n-1$ (i.e. todos los demás puntos porque 1 y n pertencen a la recta, i.e tienen distancia cero) con la mayor distancia (d_M) a dicha recta (calculada p.e. usando la ecuación anterior). Si la distancia máxima es menor que el parámetro de distancia d_{th} se considera que el segmento candidato s_1 es en efecto un segmento recto. Si, por el contrario, la distancia máxima es mayor que d_{th}, el segmento $s_1 = (\mathbf{p}_1, \mathbf{p}_n)$ se parte (*split*) en dos segmentos. Supongamos que el punto más alejado del segmento original es $\mathbf{p}_i$, entonces el segmento candidato $s_1 = (\mathbf{p}_1, \mathbf{p}_n)$ se convierte en los siguientes dos segmentos candidatos $s_1 = (\mathbf{p}_1, \mathbf{p}_i)$ y $s_2 = (\mathbf{p}_i, \mathbf{p}_n)$[1]. A continuación repetimos el proceso con el nuevo segmento s_1, buscando el punto más alejado entre los puntos 2 e $i-1$. Suponiendo que el punto más alejado tiene índice j, i.e. $\mathbf{p}_j$ con $d_M = |\hat{\mathbf{u}}_\perp \cdot (\mathbf{p}_j - \mathbf{p}_1)|$, si $d_M < d_{th}$ (distancia menor que distancia *threshold*) el segmento candidato s_1 se considera un segmento recto (i.e. ningún punto intermedio está más alejado que d_{th}), en caso contrario el segmento s_1 se parte (*split*) en otros dos segmentos dando lugar a una nueva lista de segmentos $s_1 = (\mathbf{p}_1, \mathbf{p}_j)$, $s_2 = (\mathbf{p}_j, \mathbf{p}_i)$ y $s_3 = (\mathbf{p}_i, \mathbf{p}_n)$. Es importante observar que el anterior segmento s_2 es en la nueva lista el segmento s_3, i.e. al partir un segmento se crean al principio de la lista dos nuevos segmentos a procesar, mientras que el resto de segmentos se dejan al final. Esto asegura que la lista de segmentos se procese de forma exhaustiva. En el caso de que el segmento s_1 cumpla la condición de segmento recto (i.e. desviación menor que d_{th}), se elimina de la lista de candidatos, guardandolo en una segunda lista ordenada de segmentos reales, y procesando a continuación el siguiente segmento mientras haya segmentos candidatos.

Una forma directa de implementar este algoritmo *split* es a través de una llamada a una función recursiva que cuando parte un segmento se invoca a ella misma dos veces con los rangos de puntos apropiados. Otra opción, la iterativa, requiere una estructura de datos LIFO (*Last-In Firts-Out*) que puede implementarse como un vector con un pequeño código para eliminar/insertar elementos al principio del vector. El algoritmo extraería el primer elemento de la pila LIFO, lo procesaría e introduciría dos (en caso de *split*) o ninguno (en caso de segmento lineal). Este proceso se repetiría mientras la pila LIFO no esté vacia, mientras que cada segmento lineal válido tendría que ser almacenado en, p.e. un array de segmentos de salida.

1.2 *Split & Merge*: *Merge*

La salida de la parte *Split* del algoritmo es una lista ordenada de segmentos $s_1, s_2, \cdots, s_m$ con puntos a distancias menores que d_{th}, pero es posible que debido al ruido algunas rectas estén "sobre-segmentadas", p.e. segmentos consecutivos que pertencen a una misma pared. El siguiente paso del algoritmo (*merge*) consiste, por tanto, en unir dichos segmentos. Para ello se considera que dos segmentos candidatos son en realidad uno si el ángulo entre ellos es pequeño, i.e. menor que un ángulo límite a_{th}. Por ello es necesario calcular el ángulo entre dos segmentos consecutivos en coordenadas Cartesianas. Dado el segmento $s_i = (\mathbf{p}_j, \mathbf{p}_k)$ el ángulo que forma este segmento con el eje x del sistema de referencia del

[1] También prodrian ser $s_1 = (\mathbf{p}_1, \mathbf{p}_i)$ y $s_2 = (\mathbf{p}_{i+1}, \mathbf{p}_n)$ o $s_1 = (\mathbf{p}_1, \mathbf{p}_{i-1})$ y $s_2 = (\mathbf{p}_i, \mathbf{p}_n)$ para evitar que un mismo punto pertenezca a dos segmentos

lidar es:

$$\alpha_i = \arctan\left[\frac{y_k - y_j}{x_k - x_j}\right]$$

donde $\mathbf{p}_j = (x_j, y_j)$ y $\mathbf{p}_k = (x_k, y_k)$. Usando la misma fórmula para obtener el ángulo α_{i+1} del segmento $s_{i+1} = (\mathbf{p}_k, \mathbf{p}_l)$ podemos calcular la diferencia entre la alineación de segmentos como $|\alpha_i - \alpha_{i+1}|$. Si esta diferencia es menor que el parámetro a_{th} el algoritmo *merge* unirá los dos segmentos en uno definido como $s = (\mathbf{p}_k, \mathbf{p}_l)$. Es importante observar que la forma de calcular los ángulos α_i debe ser consistente para los dos segmentos, i.e. el primer punto es el inicio y el segundo punto es el fin del segmento, de lo contrario los ángulos se calcularán para direcciones opuestas. Por otra parte, y dado que los ángulos α_i y α_{i+1} están en el rango $(-\pi, \pi]$, su diferencia no estará necesariamente en ese rango, por lo que es necesario normalizar la diferencia angular al rango $(-\pi, \pi]$ sumando o restando 2π dependiendo del valor de la diferencia.

En el caso en que no sea necesario juntar los segmentos considerados s_i y s_{i+1}, el segmento candidato s_i puede considerarse como definitivo, y el proceso se repite con los segmentos s_{i+1} y s_{i+2}. Si los dos segmentos se han unido en uno solo (nuevo s_i) es necesario considerar este nuevo segmento y el siguiente (i.e. s_{i+2}), reconsiderando el segmento mientras se una con su adyeacente para formar uno solo. Por tanto, la salida de esta etapa del algoritmo es una lista de segmentos con un tamaño menor (si es necesario juntar segmentos) o igual (si las diferencias en direcciones de todos los segmentos consecutivos son mayores que a_{th}) a la lista de segmentos candidatos de entrada.

1.3 *Split & Merge*: *Purge*

Aunque no contemplado en el algoritmo original, se puede implementar un paso *purge* para eliminar algunos segmentos que incluyan entre sus extremos pocos puntos o segmentos de poca distancia. Este paso puede implementarse después del paso *split*, de forma que el paso *merge* puede unir, por ejemplo segmentos discontínuos provenientes de la detección de patas de sillas o mesas, generando un perfil de las paredes del entorno. Este paso se puede implementar para que elimine segmentos que cumplan una o las dos condiciones, i.e. si el segmento $s_i = (\mathbf{p}_j, \mathbf{p}_k)$ cumple $|k - j| < \Delta$ (menos que Δ puntos) o $|\mathbf{p}_j - \mathbf{p}_k| < l_{th}$ (segmentos más cortos que l_{th}) se elimina de la lista.

2 Implementación del algoritmo *Split & Merge*

El fichero `splitAndMerge.py` contiene el código necesario para la implementación del algoritmo *Split & Merge*. Como en la práctica anterior usaremos ficheros con lecturas de Lidar. El fichero declara dos clases principales, la clase `Segment` y la clase `SplitAndMerge`. La primera clase almacenará el segmento a través de sus extremos (`p0` y `p1`) y los indices correspondientes (`idx0` y `idx1`), e implementa métodos para calcular la distancia del segmento a un punto (`distance()` para *split*), el ángulo con el siguiente segmento (`angle()` para *merge*) y su longitud (`length()` para *purge*), aunque solo este último método está implementado. Por tanto, el primer paso es implementar los métodos de la clase `Segment` que calculan la distancia a un punto y el ángulo con el siguiente segmento. Dado que el algoritmo se basa en estos dos métodos es muy importante asegurarse que estén bien implementados **Q1. ¿Qué pruebas has hecho para asegurarte que no hay ningún *bug* en estos métodos? Muestra las pruebas y explica los resultados.**

Una vez completada la implementación de la clase `Segment` podemos pasar a implementar la parte *split* del algoritmo en el método `split()` de la clase `SplitAndMerge`. El argumento de entrada es una lista de puntos 2D como arrays de `numpy` y el parámetro de salida debe ser una lista (u otro contenedor de datos) con los segmentos candidatos después del proceso *split*. Como se ha mencionado anteriormente, esta parte puede implementarse de forma recursiva llamando al método `split()` con las sublistas de puntos correspondientes a los nuevos segmentos candidatos, aunque puede ser necesario modificar la forma en que se llama al método. En el caso recursivo el método deberá construir listas a partir de las listas (o segmentos individuales) retornados. A pesar de la simplicidad de las implementaciones recursivas de algunas funciones, el coste computacional de la recursividad es bastante alto, ya que cada vez que se llama a la función el contexto de ejecución y los parámetros de llamada se copian en la pila del proceso[2]. Una alternativa es implementar el algoritmo de *split* de forma iterativa aunque, como se ha mencionado, es necesario disponer de un contenedor LIFO en el que se introduce el primer segmento candidato y despues procesar el contenedor hasta que éste esté vacio. Por suerte `python` implementa un paquete `queue` que proporciona (entre otras) una implementación de pilas LIFO (`queue.LifoQueue()`). Si se define un objeto de este tipo, los métodos `put()` y `get()` pueden usarse para almacenar y extraer datos respectivamente. El método `empty()` permite comprobar si el contenedor está o no vacio, y el campo `queue` permite obtener los datos almacenados, que pueden convertirse en una lista a través del constructor de la clase `list` como `my_list = list(my_queue.queue)` donde `my_queue` es una cola y `my_list` será una lista con los elementos de la cola. Esta conversión puede venir bien para depurar el código de *split* (o *merge*) visualmente usando el método `plotSegments()` de la clase `SplitAndMerge`, ya que este método toma como entrada una lista (`list`) de segmentos, no una cola de segmentos (`queue.LifoQueue()`). Implementa el método `split()` de la forma que te parezca (i.e iterativa o recursiva) usando el método `plotSegments()` para visualizar los pasos del algoritmo. **Q2. ¿Cómo procesa tu algoritmo los segmentos? Muestra una secuencia de pasos de procesamiento de segmentos candidatos y comenta los resultados en relación con tu implementación. Q3. ¿Cuántos segmentos obtienes y de qué longitud para el fichero y la lectura que usas? Cambia la lectura (variable `idx` en el programa principal) y recalcula el número de segmentos y sus longitudes. ¿Son iguales? ¿Qué significa eso?**

La lista de segmentos candidatos retornada por el método `split()` se utilizará como argumento de entrada para el método `merge()`. En este caso es interesante también tener un contenedor estilo LIFO ya que cuando se mezclan dos segmentos consecutivos hay que volver a insertar el segmento resultante para compararlo con el siguiente. Es más, en caso de que no se combinen dos segmentos, el primero se puede almacenar en el contenedor final pero el segundo hay que volver a insertarlo para la siguiente comparación. Implementa el método `merge()` de la clase. **Q4. ¿Qué valor has puesto para tu *threshold* angular? ¿Cuantos segmentos se han unido en tu caso? i.e. ¿Cual es la diferencia entre el número de segmentos de entrada y salida?** Aumenta el valor de a_{th} **Q5. ¿Cuántos segmentos se unen con el nuevo valor del parámetro? Explica los resultados, incluyendo una gráfica de los segmentos obtenidos en cada caso.**

Por último implementaremos el método `purge()` para eliminar segmentos con pocos puntos y/o segmentos cortos de la lista proporcionada por el método `merge()`. **Q6. ¿Qué parámetros y condiciones usas para eliminar segmentos? ¿Cuántos segmentos se eliminan según tus parámetros para el fichero de datos e índice seleccionado? Q7. Cambia la lectura (variable `idx` en el programa principal) y recalcula el número de segmentos y sus longitudes. ¿Son iguales? Compara y comenta los resultados con**

[2]Este es el origen del nombre de la web `stackoverflow.com`

el caso Q5.

Bonus: Supón que los puntos que definen el segmento son correctos, i.e. no tienen error, pero al calcular la distancia de la recta a los puntos intermedios consideras la covarianza (error) en la lectura. **Q8. Explica cómo calcularias la distancia media entre la recta y el punto y su indertidumbre (varianza) usando la ecuación de distancia** $d = |\hat{\mathbf{u}}_\perp \cdot (\mathbf{p} - \mathbf{p}_1|$ **conocida la covarianza del punto en coordenadas Cartesianas, i.e.** $\mathbf{p} \sim \mathcal{N}(\mathbf{p}, \Sigma_c)$.