Robótica Móvil
Localización de un Robot con Filtro de Kalman Extendido

Iñaki Rañó

Notas

El objetivo de esta práctica es implementar el filtro de Kalman extendido (*EKF*) para la localización de un robot móvil de tipo integrador. Se simularán dos tipos de 'sensores' distintos, uno que puede medir coordenadas Cartesianas y otro que mide distancia y ángulo (*range and bearing*), por lo que habrá que implementar dos filtros de Kalman extendidos, uno para cada tipo de sensor.

1 Introducción

El filtro de Kalman se puede utilizar en robótica móvil para estimar la posición (o la pose) de un robot dado un mapa del entorno. Para ello el robot debe ser capaz de observar con sus sensores las *landmarks* del entorno y ser capaz de asociar las *landmarks* observadas con las que tiene almacenadas en el mapa. En principio el mapa proporciona las posiciones de las *landmarks*, típicamente en coordenadas Cartesianas, mientras que las obervaciones de las *landmarks* realizadas por el robot suelen ser del tipo distancia y ángulo (*Range & Bearing*). En este caso vamos a trabajar con un robot de tipo integrador con las velocidades Cartesianas como entradas, por lo que el modelo de transición de estado es lineal, y en su versión discreta puede expresarse como:

$$x_{k+1} = x_k + Tv_x$$
$$y_{k+1} = y_k + Tv_y$$

donde (x_k, y_k) es el estado, (v_x, v_y) son las entradas de control y T es el periodo de discretización. Para el robot de tipo uniciclo las ecuaciones de transición de estado son no lineales pero se puede localizar el robot también con el filtro de Kalman extendido.

Como vamos a considerar dos modelos de medida (coordenadas Cartesianas y *Range & Bearing*) habrá dos ecuaciones distintas de observación. Llamaremos $\mathbf{x}_{li} = [x_{li}, y_{li}]$ a las coordenadas de la *landmark* i en el mapa, y $\hat{\mathbf{x}}_k = [\hat{x}_k, \hat{y}_k]$ a la posición estimada del robot en el instante k, y supondremos que desde esa posición el robot puede observar la *landmark* i. En el caso en que las medidas sean las coordenadas Cartesianas de un *landmark* (relativas al robot) el robot observará la *landmark* en una posición relativa a su sistema de referencia $\hat{\mathbf{y}}_k = \mathbf{x}_{li} - \hat{\mathbf{x}}_k$. En un instante de tiempo k el robot puede observar simultaneamente más de una *landmark* por lo que el número de salidas del modelo o la dimensión del vector de observaciones $\hat{\mathbf{y}}_k$ puede variar de un instante a otro. En cualquier caso, si en el instante k el

robot observa $p(k)$ *landmarks*, que indexaremos con el índice j, y si conocemos la función de correspondencia $a : j \rightarrow i$, una función que para cada *landmark* detectada nos diga a qué *landmark* del mapa corresponde, la salida será:

$$\hat{\mathbf{y}}_k = \begin{bmatrix} \mathbf{x}_{la(1)} - \hat{\mathbf{x}}_k \\ \mathbf{x}_{la(2)} - \hat{\mathbf{x}}_k \\ \vdots \\ \mathbf{x}_{la(p(k))} - \hat{\mathbf{x}}_k \end{bmatrix}$$

donde $\mathbf{x}_{la(j)}$ es la posición en el mapa de la *landmark* correspondiente a la *landmark* j observada por el robot, y cada par de componentes del vector corresponde con la posición relativa de la *landmark* detectada. Esta es la ecuación de salida ($\mathbf{y}_k = C\mathbf{x}_k$) que predice lo que el robot debería observar de estar en el estado $\mathbf{x}_k$. Como se puede ver esta ecuación de salida es no lineal ya que incluye la posición de las *landmarks* del mapa que son constantes[1].

En el caso del modelo de observación basado en *range & bearing* para una *landmark* (j) detectada por el robot la salida predicha será:

$$\hat{\mathbf{y}}_k = \begin{bmatrix} \sqrt{(x_{la(j)} - \hat{x}_k)^2 + (y_{la(j)} - \hat{y}_k)^2} \\ \arctan\left[\frac{y_{la(j)} - \hat{y}_k}{x_{la(j)} - \hat{x}_k}\right] \end{bmatrix}$$

y en el caso general habrá tantas componentes de salida como *landmarks* detectadas ($p(k)$), es decir

$$\hat{\mathbf{y}}_k = \begin{bmatrix} \sqrt{(x_{la(1)} - \hat{x}_k)^2 + (y_{la(1)} - \hat{y}_k)^2} \\ \arctan\left[\frac{y_{la(1)} - \hat{y}_k}{x_{la(1)} - \hat{x}_k}\right] \\ \vdots \\ \sqrt{(x_{la(p(k))} - \hat{x}_k)^2 + (y_{la(p(k))} - \hat{y}_k)^2} \\ \arctan\left[\frac{y_{la(p(k))} - \hat{y}_k}{x_{la(p(k))} - \hat{x}_k}\right] \end{bmatrix}$$

que claramente es una ecuación de medida no lineal. En cualquier caso es importatne recordar que las ecuaciones anteriores son las que predicen las medidas a observar ($\hat{\mathbf{y}}_k$) a partir del estado estimado ($\hat{\mathbf{x}}_k$), mientras que llamaremos $\mathbf{y}_k$ a la observación real del robot. Esto nos permite calcular el error de innovación $\tilde{\mathbf{y}}_k$ simplemente como $\tilde{\mathbf{y}}_k = \mathbf{y}_k - \hat{\mathbf{y}}_k$, donde recordemos que $\mathbf{y}_k$ es generado por los sensores mientras $\hat{\mathbf{y}}_k$ se calcula a partir del mapa (junto con la función de correspondencia) y la estimación del estado.

Con estos dos modelos de observación en mente podemos definir las ecuaciones del sistema discreto de la forma estándar:

$$\mathbf{x}_{k+1} = \mathbf{x}_k + T\mathbf{v}_k + \mathbf{w}$$
$$\mathbf{y}_k = \mathbf{G}_k(\mathbf{x}_k) + \mathbf{v}$$

donde $\mathbf{w} \sim \mathcal{N}(\mathbf{0}, Q)$, $\mathbf{v} \sim \mathcal{N}(\mathbf{0}, R)$ y $\mathbf{G}_k(\mathbf{x}_k)$ es una de las ecuaciones de salida anteriores que depende de k (el instante de tiempo) ya que las *landmarks* detectadas cambian de un instante a otro. Partiendo de una estimación inicial de la posición $\hat{\mathbf{x}}_{0|0}$ con matriz de covarianza $\hat{P}_{0|0}$ se puede utilizar el filtro de Kalman extendido para actualizar la estimación del estado de la siguiente forma:

[1] Es decir, la ecuación de salida/medida del sistema no puede en realidad escribirse como $\mathbf{y_k} = C\mathbf{x}_k$.

1. Predicción:

$$\hat{\mathbf{x}}_{k+1|k} = A\hat{\mathbf{x}}_{k|k} + B\mathbf{v}_k$$
$$\hat{P}_{k+1|k} = A\hat{P}_{k|k}A^T + Q$$

2. Actualización:

$$\hat{\mathbf{x}}_{k+1|k+1} = \hat{\mathbf{x}}_{k+1|k} + K\tilde{\mathbf{y}}_{k+1}$$
$$\hat{P}_{k+1|k+1} = \hat{P}_{k+1|k} - KS_{k+1}K^T$$

donde $\tilde{\mathbf{y}}_{k+1}$ es el error de innovación, la ganancia de Kalman es $K = \hat{P}_{k+1|k}[J\mathbf{G}_k]^T S_{k+1}^{-1}$, la covarianza de la innovación es $S_{k+1} = [J\mathbf{G}_k]\hat{P}_{k+1|k}[J\mathbf{G}_k]^T + R$ y $J\mathbf{G}_k$ es la matriz Jacobiana de la ecuación de observación (derivadas parciales con respecto a $\hat{x}_k$ y $\hat{y}_k$). Para intentar entender un poco mejor las ecuaciones del filtro de Kalman en este caso vamos a analizar las dimensiones de los vectores y las matrices involucradas. Por un lado $\hat{\mathbf{x}}_{k|k}$ es un vector bidimensional mientras que su covarianza $\hat{P}_{k|k}$ es una matriz 2×2. Como A es una matriz constante 2×2 y la covarianza del ruido de estado Q también es una matriz 2×2 las ecuaciones de predicción son consistentes en términos de dimensiones de matrices y vectores. En cuanto a las dimensiones en la ecuación de actualización la cosa se complica ligeramente. Por un lado el vector de innovación $\tilde{\mathbf{y}}_{k+1}$ tiene dimensión variable con el tiempo (k) igual a $2p(k)$, ya que depende del número de *landmarks* detectado en el instante k. Esto implica que la ganancia de Kalman K debe ser una matriz de tamaño $2 \times (2p(k))$ para que el producto $K\tilde{\mathbf{y}}_{k+1}$ sea un vector de dimensión 2. Dado este tamaño de K, para que la ecuación de actualización de la covarianza sea consistente la matriz S_{k+1} debe tener tamaño $(2p(k)) \times (2p(k))$, lo cual tiene sentido ya que es la covarianza de la innovación (un vector de dimensión $2p(k)$). Si consideramos ahora la ecuación para la ganancia de Kalman $K = \hat{P}_{k+1|k}[J\mathbf{G}_k]^T S_{k+1}^{-1}$, para que los productos sean consistentes la matriz Jacobiana $J\mathbf{G}_k$ debe tener tamaño $2p(k) \times 2$, lo cual concuerda con el tamaño de $\hat{\mathbf{y}}_k$ ($2p(k)$) y el hecho de que depende de $\hat{\mathbf{x}}_k$, que es un vector de dimensión 2. Por último, sabemos que la covarianza de la innovación $\hat{S}_{k+1}$ es una matriz de tamaño $(2p(k)) \times (2p(k))$ que resulta de la suma de $[J\mathbf{G}_k]\hat{P}_{k+1|k}[J\mathbf{G}_k]^T$ con tamaño $(2p(k)) \times (2p(k))$ y la matriz de covarianza del ruido de medida R. El problema es que R, en principio, tiene tamaño 2×2 para cada una de las medidas, con lo cual la suma de matrices que resulta en S_{k+1} es inconsistente en términos de tamaños de matrices. Lo que ocurre es que hay una matriz R de tamaño 2×2 para cada una de las $p(k)$ *landmarks* observadas por el robot, lo que implica la necesidad de construir una matriz de ruido de medida repitiendo $p(k)$ veces la matriz R para una sola medida. Esto se consigue a través del producto de Kronecker[2] $I_{(2p(k))\times(2p(k))} \otimes R$, es decir hay que sustituir $R \rightarrow I_{(2p(k))\times(2p(k))} \otimes R$, donde $I_{(2p(k))\times(2p(k))}$ es la matriz identidad de tamaño $(2p(k)) \times (2p(k))$.

2 Simulación del entorno y el robot

Para la implementación de la localización basada en *EKF* se proporcionan tres ficheros `python`; `Simulator.py`, `EKFLocalisationCart.py` y `EKFLocalisationRB.py`.

El fichero `Simulator.py` define las clases auxiliares necesarias para implementar la localización; `Landmark`, `Map` y `Robot`, además de una clase que permite la visualización del proceso de localización del robot (`EnvPlot`). La clase `Landmark` contiene las coordenadas de la *landmark* y es usada para almacenar la posición de ésta en el mapa y como valor de

[2] El producto de Kronecker está implementado en `numpy` como `numpy.kron()`.

retorno en el método de medida del robot. Además contiene un identificador único (campo `id`) para cada *landmark*. La clase `Map` no es más que una lista de objetos tipo `Landmark` en coordenadas Cartesianas, y por tanto puede trararse como una lista estándar. Una característica de la clase `Landmark` es que se ha sobrecargado el método `__eq__` para permitir comparar *landmarks* basandose en su identificador, es decir se puede comparar una *landmark* del mapa, e.g. `map[ii]`, con una observada por el robot, e.g. `obs[jj]`, de la siguiente forma `map[ii] == obs[jj]` para comprobar si corresponde con la misma *landmark*, i.e. para implementar la función de correspondencia. La clase `Robot` tiene dos métodos importantes, el método `action()` que mueve el robot en una trayectoria predeterminada (un cuadrado en el entorno) y retorna la velocidad usada, y el método `measure()` que toma como entrada el mapa y retorna una lista de las *landmarks* observadas por el robot (i.e aquellas que están dentro de su rango sensorial). Es importante destacar que la clase `Robot` tiene un campo llamado `type` que define el tipo de observación generada por el método `measure()`. Si el valor de `type` es la cadena `'xy'`, el método `measure()` retornará una lista de *landmarks* con un `numpy.array` que contiene las coordenadas Cartesianas de la *landmark* relativa al robot. En caso que el campo `type` tenga el valor `'rb'` las *landmarks* devueltas por el método `measure()` contendrán la distancia y el ángulo (relativos al robot) de la *landmark* detectada. Por lo tanto la forma de interpretar el contenido de las *landmarks* que retorna el método `measure()` dependerá de si estamos tratando con el caso Cartesiano o el *Range & Bearing*.

Finalmente, al representar gráficamente la simulación se muestra el entorno con las *landmarks* como estrellas en color negro, el robot simulado con el signo '$+$' en azul centrado en un círculo discontinuo (también de color azul) que representa el rango de percepción del robot. La posición de las *landmarks* detectadas (y corrompidas con ruido) por el robot aparencen como estrellas azules, generalmente cerca de las *landmarks* del mapa. Finalmente, la clase que representa gráficamente la simulación dibuja una elipse de color rojo con el signo '$+$' en rojo en su centro para representar la estimación del filtro de Kalman (promedio e incertidumbre a 3σ, i.e. 99.7% de probabilidad de que el valor real esté dentro de la elipse). La simulación muestra primero la situación inicial y pausa el entorno gráfico hasta que se pulse una tecla.

3 Localización con medidas Cartesianas

Para implementar la localización con medidas Cartesianas usarémos el `script` proporcionado en el fichero `EKFLocalisationCart.py`, que contiene la definición incompleta de la clase `KF` para implementar el filtro de Kalman. Este *script* de `python` comienza definiendo una variable `noLandmarks` que representa el número de *landmarks* que contendrá el mapa a usar. Las *landmarks* se ubican de manera aleatoria en el entorno, de forma que cada ejecución generará una configuración diferente. El *script* define las matrices de covarianza de ruido de estado (Q) y de medida (R) y crea un objeto de clase `Robot` usando esas matrices. En principio, aunque es posible acceder al estado (posición) del robot, no es necesario ya que se estimará a través de las acciones (método `action()`) y las observaciones (método `measure()`). Al final del *script* se define un objeto de clase `KF` al que se pasa una estimación inicial del estado en forma de media (`xHat`) y matriz de covarianza (`PHat`) junto con las matrices de covarianza del error de estado y medida. Después se simula el robot y el filtro de Kalman usando el algoritmo de Euler, llamando a los métodos `action()` y `measure()` del objeto de clase `Robot` e intercalado a las llamadas al filtro de Kalman `predict()` y `update()` (métodos a implementar).

Q1. Implementa los métodos `predict()` y `update()` de la clase `KF`. ¿Cuántos pasos tarda el filtro de Kalman en estimar correctamente la posición del robot, i.e. que la

posición real esté dentro de la incertidumbre dibujada en la matriz de covarianza?

Para ver como funciona el filtro de Kalman más en detalle realizaremos simulaciones en las que las estimaciones de los ruidos de estado y medida no coinciden con las del robot real. Para ello primero reduciremos la covarianza del error de medida que se le pasa al filtro de Kalman, sustituyendo la línea de código en que se crea el objeto de tipo `KF`, `kf = KF(xHat, PHat, Q, R, m)` por `kf = KF(xHat, PHat, Q, 0.01*R, m)`, i.e. el error de medida del robot es 10 veces mayor que lo que supone el filtro de Kalman. **Q2. Repite la simulación de la localización con estos parámetros. ¿Cuántos pasos tarda el filtro de Kalman en estimar correctamente la posición del robot? ¿Qué ocurre con la estimación? ¿Está la posición del robot real dentro del rango definido por la Gaussiana?** Justifica tu respuesta usando las ecuaciones del filtro y el significado del error de medida, puedes ilustrarla con imágenes del simulador.

A continuación vamos a incrementar el error de medida que se le pasa al filtro de Kalman sustituyendo la línea anterior por `kf = KF(xHat, PHat, Q, 100*R, m)`. **Q3. Simula la localización con estos parámetros. ¿Cuántos pasos tarda el filtro de Kalman en estimar correctamente la posición del robot? ¿Qué ocurre con la estimación? ¿Está la posición del robot real dentro del rango definido por la Gaussiana? ¿Por qué?**. Justifica tu respuesta usando las ecuaciones del filtro y el significado del error de medida, puedes ilustrarla con imágenes del simulador.

Q4. Repite los procesos anteriores dejando la covarianza del ruido de medida como estaba y cambia la covarianza del ruido de estado. ¿Qué parámetro crees que tiene más influencia en el resultado del filtro para localización? ¿Por qué?

Q5. Repite la primera simulación, i.e. usa las matrices de covarianza originales, reduciendo el número de *landmarks* a 5. ¿Qué ocurre cuando el robot no detecta ninguna *landmark*? ¿Por qué?

4 Localización basado en *Range & Bearing*

Para implementar la localización con medidas de distancia y ángulo usarémos el *script* proporcionado en el fichero `EKFLocalisationRB.py`, que contiene la definición incompleta de la clase `KF` que implementa el filtro de Kalman. La estructura de este *script* es igual al caso anterior, pero en este caso las lecturas obtenidas por el método `measure()` de la clase `Robot` correnponden a la distancia y ángulo de la *landmark* referidas a la posición del robot[3]. **Q6. Implementa los métodos de la clase `KF` para localizar el robot en el mapa usando distancia y ángulo a las *landmarks*. ¿Notas alguna diferencia con respecto al método anterior? Ejecuta varias simulaciones con diferentes mapas aleatorios observando qué pasa con la forma de la incertidumbre**

Q7. Reduce el número de *landmarks* en el mapa a 5. ¿Qué ocurre cuando el robot no detecta ninguna *landmark*? ¿y si detecta solo una y pasa relativamente cerca de ella? Justifica tus observaciones igual que en el apartado anterior.

[3]Compara la llamada al constructor de la clase `Robot` en este *script* y en el anterior.