Robótica Móvil
Evitación de Obstáculos con Métodos de Potencial

Iñaki Rañó

Notas

El objetivo de esta práctica es implementar métodos de potencial para la evitación de obstáculos de dos robots móviles simulados (uno que sigue un modelo integrador y otro con modelo uniciclo). El simulador está implementado en `C++` ya que simular un sensor de proximidad estilo Lidar o Laser involucra calcular intersecciones de segmentos con los modelos de los obstáculos (círculos, elipses o polígonos) en el entorno lo que en `python` es prohibitivo. El simulador funciona en GNU/Linux y requiere la instalación de las librerías `Eigen3` y `gtkmm-3`

1 Introducción

Aunque hay distintos métodos de evitación de obstáculos, algunos de ellos ya implementados en software para robots como `ROS`, en esta práctica vamos a implementar el método de potencial. Los métodos de potencial se basan en la analogía de los campos electrostáticos, en los que partículas con carga de distinto signo se atraen y partículas con carga de igual signo se repelen. El nombre "métodos de potencial" se debe a que las fuerzas de atracción y repulsión son proporcionales al gradiente de una función escalar, la función de potencial (i.e. energía potencial). Dejando de lado la justificación teórica, las partículas con carga sufren fuerzas atractivas o repulsivas dependiendo de la configuración espacial de otras cargas en su entorno. En el caso de los métodos de potencial en robótica se considera que el robot y los obstáculos tienen carga del mismo signo (i.e. se repelen mutuamente), mientras que el objetivo tiene carga de signo contrario al robot (i.e. se atraen). Es importante observar que tanto el objetivo como los obstáculos son cargas estáticas, mientras que el robot es el que se mueve en el entorno. Por otra parte, las fuerzas, como vectores, tienen la dirección que une el robot con los obstáculos y con la posición objetivo, mientras que la norma (longitud) de la fuerza dependerá de alguna forma de la distancia[1]. Por otra parte la analogía es aproximada, ya que en un sistema físico real la energía se conserva y el robot (más bien la partícula con carga) oscilaría alrededor de la carga con signo opuesto.

Suponiendo que el robot se encuentra en la posición $\mathbf{x}_r$ y su objetivo está en la posición $\mathbf{x}_t$, entonces para que la fuerza sea atractiva debe apuntar desde la posición del robot a la posición del objetivo. Si definimos el vector normalizado $\hat{\mathbf{u}}_t = \frac{\mathbf{x}_t - \mathbf{x}_r}{|\mathbf{x}_t - \mathbf{x}_r|}$, entonces la fuerza atractiva debe ser proporcional (con proporcionalidad positiva) a dicho vector, es decir $\mathbf{F}_a = f_a(|\mathbf{x}_t - \mathbf{x}_r|)\hat{\mathbf{u}}_t$, donde $f_a(\cdot)$ es una función de la distancia entre el robot y el objetivo ($|\mathbf{x}_t - \mathbf{x}_r|$) que toma valores positivos ($f_a(x) > 0 \ \forall x$). La forma de $f_a(\cdot)$ suele tomarse

[1] Cualquier fuerza que dependa sólo de la distancia se deriva de un potencial.

tal que cuando el robot está muy alejado del objetivo devuelva valores constantes, mientras que el valor devuelto decrece hacia cero cuando el robot está cerca del objetivo.

Supongamos ahora que el robot en la posición $\mathbf{x}_r$ detecta un obstáculo en la posición $\mathbf{x}_o$ que genera una fuerza repulsiva, i.e. el vector de fuerza applicado al robot va en la dirección contraria en que se detecta el obstáculo[2]. Si definimos el vector normalizado $\hat{\mathbf{u}}_o = \frac{\mathbf{x}_r - \mathbf{x}_0}{|\mathbf{x}_r - \mathbf{x}_o|}$, entonces la fuerza repulsiva debe ser proporcional (con proporcionalidad positiva) a dicho vector, es decir $\mathbf{F}_o = f_r(|\mathbf{x}_r - \mathbf{x}_o|)\hat{\mathbf{u}}_o$, donde $f_r(\cdot)$ es una función de la distancia entre el robot y el obstáculo ($|\mathbf{x}_r - \mathbf{x}_o|$) que toma valores positivos ($f_r(x) > 0 \; \forall x$). La forma de $F_r(\cdot)$ suele tomarse tal que cuando el robot está suficientemente alejado del obstáculo el valor es cero[3], mientras que cuando el obstáculo está cerca el valor devuelto por la función es alto, incluso, en ocasiones, tendiendo a infinito para distancia cero.

Una vez calculadas la fuerza atractiva y las repulsivas creadas por todos los obstáculos detectados, la fuerza resultante es una combinación lineal de la suma de todas las fuerzas (posiblemente multiplicadas por pesos, e.g. $\mathbf{F} = k_a\mathbf{F}_a + K_r\mathbf{F}_o$). El robot debe moverse en la dirección de la fuerza resultante, lo cual es factible en el caso del robot integrador ya que puede moverse en cualquier dirección, pero, como veremos más adelante, el movimiento se complica en el caso del robot uniciclo. Si los parámetros (funciones $f_a(\cdot)$, $f_r(\cdot)$ y posiblemente pesos) se ajustan de forma adecuada la fuerza resultante puede enviarse directamente como señal de velocidad al robot, aunque en esta práctica no será necesario, ya que el simulador limita la velocidad máxima del robot (para evitar problemas de divergencia en la simulación por errores en el código).

2 Entorno de simulación

El simulador para esta práctica ha sido adaptado[4] para simular robots de tipo integrador y uniciclo en entornos 2D con distintos tipos de obstáculos (en realidad círculos y polígonos). Para compilar el simulador es necesario instalar (p.e. usando `sudo apt-get install` en Ubuntu) las librerías `Eigen3` y `libgtkmm-3.0-dev`[5]. Una vez instaladas dichas librerías el fichero `makefile` en la carpera de código fuente genera una librería (`libRS.a`) y compila el fichero que contiene el programa principal `OARobot.cc`. Si estas interesado o interesada puedes explorar los fichero fuente `C++` aunque el código siendo optimista se pordría calificar como mediocre, i.e. no es un buen patrón a seguir. Los ficheros que forman la librería son: `environment.??` define el entorno, su tamaño y una serie de primitivas geométricas (segmento, círculo y polígono) necesarias para simular el robot; `robot.??` define el robot con sus características como velocidades máximas y mínimas, número de lecturas del Lidar simulado... y proporciona funciones que permiten simular tanto un modelo de movimiento integrador como uniciclo; `simulation.??` define una clase que realiza la simulación a partir del entorno y el robot; y `simulation-plot.??` que visualiza la simulación.

En principio lo único que se necesita entender del simulador para completar la práctica son los ficheros `OARobot.cc` y `robot.hh`. En el fichero principal `OARobot.cc` se define una clase llamada `OARobot` derivada de la clase `rs::Robot` para la cual solo hay que completar el método `action()` para que implemente el método de evitación de obstáculos basado en potencial. Si abres el fichero `OARobot.cc` verás que después de todos los *incudes* está la línea de código `#define INTEGRATOR` que se usa para una compilación condicional del código que viene a continuación. La clase `rs::Robot` tiene dos constructores (puedes abrir

[2]Las posiciones son siempre referidas al sistema de referencia del mundo.

[3]Para que obstáculos lejanos no afecten al movimiento

[4]Del simulador multi-robot de computación distribuida para sistemas multirrobot

[5]Es posible que también se necesite `libgstreamermm-1.0-dev`

el fichero `robot.hh` para verlo) muy similares, con la diferencia que uno de ellos toma como parámetro un objeto de clase `Position`, mientras el otro toma un objeto de clase `Pose`. Mientras que el primer constructor se usa para crear un robot con movimiento de tipo integrador, i.e. `Position`$= (x, y)$, el otro crea un robot uniciclo, i.e. `Pose`$= (x, y, \theta)$. que no son más que vectores `Eigen` de 2 y 3 dimensiones. En la clase `rs:Robot` se define una unión llamada `m_p` que contiene dos campos, una posición (`m_p.pos`) y una pose (`m_p.pose`), por lo que al implementar la clase derivada `OARobot` es necesario ser consciente de si se implementa para un robot integrador con posición `m_p.pos` o un robot uniciclo con pose `m_p.pose`.

Las velocidades a calcular, que deben guardarse en el campo `m_vel` de la clase `mr::Robot` serán interpretadas por el simulador de forma distinta para el modelo integrador y el uniciclo. Mientras que para el integrador `m_vel` contiene las velocidades en x y en y (v_x, v_y), en el caso del uniciclo `m_vel` se interpreta como velocidad lineal y angular, es decir `m_vel`$= (v, \omega)$. Independientemente del modelo que se use la clase `rs::Robot` define los métodos `pose()` y `position()` que retornan un objeto de la clase que corresponda, solo que en el caso del modelo integrador el ángulo de la pose se considera cero[6].

Por último, para implementar la evitación de obstáculos se simula un sensor de proximidad estilo Laser que cubre los 360° alrededor del robot, y sus lecturas empiezan en $-\pi$ y terminan en π en el sistema de referencia del robot con incrementos angulares constantes. El número de lecturas que se obtienen está configurado a través de la estructura `rs::RobotSettings` una instancia de la cual se inicializa en el fichero `OARobot.cc`. El rango de este sensor es también configurable a través de la estructura `rs::RobotSettings` y por defecto en el fichero tiene un valor de 2m. Las lecturas de distancia se guardan en un objeto de tipo `std::vector<float>` llamado `m_R`. Aunque en principio la primera y la última lectura coinciden `m_R[0]` corresponde con $-\pi$ y `m_R[m_R.size()-1]` corresponde con π, esto no deberiá afectar mucho a la evitación de obstáculos.

Por último el programa principal en el fichero `OARobot.cc` soporta dos parámetos de entrada, una cadena de texto (nombre de fichero) y un número entero (mapa a usar para la simulación). Al llamar al programa desde la línea de comandos se puede ejecutar simplemente como `./OARobot`, pero si se añade un argumento, p.e. `./OARobot simulacion.dat` el programa guardará en el fichero `simulacion.dat` los resultados de la simulación de forma que cada línea del fichero corresponde con un instante de tiempo y sigue el formato:

$$\text{Tiempo} \quad \text{Pos. x} \quad \text{Pos. y} \quad \theta \quad r_{-\pi} \quad \cdots \quad r_{\pi}$$

que puede usarse para analizar o dibujar la trayectoria usando algún software (Matlab, Excel,...).

El segundo parámetro indica el mapa del entorno a usar en la simulación. Hay dos mapas definidos correspondientes con el número 0 y el 1, que contienen respectivamente obstáculos circulares y un obstáculo poligonal en forma de U (*dead-end* o *cul-del-sac*). Si se introduce un número distinto a esos el entorno no contendrá obstáculos (lo cual puede venir bien, por ejemplo, para hacer *debugging* del método). Es decir, llamar al programa usando `./OARobot`, `./OARobot foo.dat` o `./OARobot foo.dat 0` generará un entorno con obstáculos circulares. Si la llamada es `./OARobot foo.dat 1` el entorno contendrá un obstáculo poligonal en forma de U, mientras que `./OARobot foo.dat 3` (por ejemplo) generará un entorno sin obstáculos.

[6] El sistema de referencia del robot integrador es paralelo al del mundo/simulador.

3 Implementación para el modelo integrador

En esta parte de la práctica implementaremos un método de potencial para un robot que sigue un modelo integrador. Lo primero es asegurarte que la línea de código `#define INTEGRATOR` no está comentada para que se compile la parte correspondiente que define la clase `OARobot` y la que define el apuntador `rp` al objeto de tipo `OARobot`. En la definición del apuntador puedes cambiar la posición inicial del robot en el entorno (el tamaño del entorno por defecto es de 10×10 metros, i.e. va desde $(-5, -5)$ a $(5, 5)$), y tendrás que hacerlo para las simulaciones que se piden. Teniendo en cuenta que la posición del robot está almacenada en el campo `m_p.pos` de la clase `rs::Robot`, las lecturas del escáner Laser simulado en `m_R` y que la velocidad `m_vel` será (v_X, v_y), implementa un método de potencial para que el robot se acerque al objetivo $(0, 0)$. **Q1. Explica qué funciones de potencial has usado para los términos atractivos y repulsivos o derivalas de la implementación de tu fuerzas.** Simula tu método para distintas posiciones en el entorno definido como opción 0 de llamada al programa `OARobot`. **Q2. ¿Sufre tu implementación del problema de mínimos locales? ¿Desde qué posición/posiciones se queda el robot atrapado en un mínimo local?** Ilustra tu respuesta con pantallazos del simulador y/o representando la trayectoria guardada de la simulación. Si tu implementación no sufre de mínimos locales en ningún lugar del entorno modifícala para que lo haga y poder responder a Q2. **Q3. Modifica tu implementación para solventar el problema de mínimos locales. ¿Que cambios has hecho?** (puedes modificar la configuración del robot si quieres, i.e. los campos de la estructura `rSettings`).

Una vez tu implementación esté completada y probada en el entorno 0, pasa al entorno 1 (obstáculo poligonal en forma de U)[7]. A través de simulaciones **Q4. Encuentra la zona del entorno desde la cuál un robot uniciclo que comienze ahí caerá en el mínimo local. Muestra dos simulaciones con posiciones iniciales cercanas que den resultados distintos, i.e. mínimo local vs el robot alcanza el objetivo.**

4 Implementación para el modelo uniciclo

Ahora vamos a implementar un método de potencial para un robot uniciclo. Puesto que los robot de tipo uniciclo no pueden moverse instantáneamente en cualquier dirección pero el método de potencial indica la dirección en que el robot debe moverse, es necesario algún mecanismo de adaptación al modelo uniciclo. Lo más sencillo es usar, por ejemplo, un controlador proporcional sobre la dirección para hacer que el robot intente approximarse a la orientación de la fuerza resultante, es decir, si la orientación del robot es θ y el ángulo que forma la fuerza resultante con el eje x es α, podría asignarse la velocidad angular ω como $\omega = K(\alpha - \theta)$. La velocidad lineal puede asignarse directamente como una función de la distancia al objetivo o usar una implementación más elaborada (p.e. al estilo *VFH*) que reduce la velocidad lineal si el robot está muy desalineado con la fuerza resultante. Estas adaptaciones, aunque funcionan en la práctica, implican que los resultados de estabilidad teórica de los métodos de potencial ya no son válidos, i.e. es posible que el robot colisione con los obstáculos.

Tras comentar la línea de código `#define INTEGRATOR`, implementa el método de potencial para el robot uniciclo tomando como posición objetivo $(0, 0)$. Simula tu método en el entorno 0 para distintas poses del robot cambiando la parte del código en que se define el apuntador al objeto `OARobot`. **Q5. ¿Sufre tu implementación del problema de mínimos locales? ¿Desde qué pose/poses se queda el robot atrapado en un mínimo local?**

[7] En este caso el número de rayos del escáner Laser simulado se reduce a 90 para acelerar la simulación

¿Hay algún par de poses con posiciones idénticas que den resultados diferentes, i.e. mínimo local vs alcanzar el objetivo?

Pasa ahora a simular el entorno 1. **Q6. Muestra una simulación en la que el robot caiga en un mínimo local. ¿Cuál es la pose inicial del robot? Q7. ¿Puedes encontrar la zona del entorno en que el robot cae en el mínimo local? ¿Cómo lo harías?**.

Como punto final en esta práctica volveremos a la primera práctica de control de movimiento. Usa el entorno 2 (entorno sin obstáculos) para simular la evitación de obstáculos del robot uniciclo. **Q8. ¿Hay alguna diferencia entre el comportamiento de este robot y el implementado en la primera práctica? ¿Cuál y por qué o por qué no?**