Swarm Robotics: Agrupamiento

Iñaki Rañó

Notas

Esta práctica se debe realizar en un sistema GNU/Linux (o alguna versión de Unix). El lenguaje de programación será `C++` y utilizaremos una librería en fase de desarrollo, por lo que se recomienda informar sobre cualquier problema que pueda aparecer con el código para depurar posibles *bugs*. Es posible que sea necesario completar la práctica en dos sesiones.

1 Introducción

En esta práctica vamos a implementar un algoritmo sencillo de agrupamiento para un enjambre de robots que siguen un modelo de movimiento uniciclo. Los robots pueden percivir las posiciones Cartesianas y otra información de sus vecinos más próximos. El comportamiento de cada robot será idéntico y consistirá en lo siguiente:

- El robot se moverá en línea recta y cambiará de dirección aleatoriamente con cierta probabilidad.
- Mientras no detecte ningún otro robot en su vecindario seguirá moviendose y cambiando la velocidad aleatoriamente cada cierto tiempo.
- Cuando el robot detecte uno o más robots en su entorno perceptual se detendrá con cierta probabilidad, que se incrementará cuantos más robots detecte.

El simulador está implementado en C++ y permite visualizar el movimiento del enjambre. Los ficheros proporcionados (ver ficheros) implementan el simulador y son los siguientes:

- `base.hh`: Define dos tipos de datos básicos, `Position2d` y `Velocity2d` que no son más que redefiniciones de vectores bidimensionales de la librería `Eigen`[1]. También incluye la declaración de dos funciones que operan en tipos `Position2d`, `float distance()` que calcula la distancia entre dos puntos y `float angle()` que calcula el ángulo entre el eje x del entorno y el segmento que va desde el primer punto al segundo. Las funciones `wrapVar()` y `wrap2pi()` se usan internamente en la librería.
- `robot.hh`: Este fichero define la clase `Robot`, que es una clase base de la que se deben derivar implementaciones de robots para el enjambre. Los robots tienen una configuración definida en la estructuta `RobotSettings` (ver fichero) y hay una configuración por defector llamada `defaultRobotSettings` definida en `robot.cc`. Dado que el número de robots que vamos a manejar será variable utilizaremos apuntadores para reservar memoria para los robots dinámicamente. En lugar de usar apuntadores básicos de C++ (que en los nuevos estándares están desaconsejados), utilizaremos *Smart Pointers*, que se liberan automáticamente cuando ninguna variable los está usando. Concretamente el fichero `robot.hh` define un tipo `RobotPtr` como un `shared_ptr` de la librería estándar. La clase

[1] Ver https://eigen.tuxfamily.org/dox/AsciiQuickReference.txt para una referencia rápida de Eigen.

`Robot` define los métodos del robot algunos de los cuales son virtuales para que sea posible implementarlos en clases derivadas. Los métodos que hay que implementar en clases derivadas son `action()` que calcula la velocidad del robot y `clone()`, que reserva memoria para un robot y copia los datos del objeto que llama a la función generando un nuevo apuntador al robot creado[2]. Por último, el método `step()` integra el movimiento del robot usando el algoritmo de integración de Euler (ver código del método en `robot.cc`).

- `swarm.hh`: Este fichero define la clase enjambre, que no es más que un vector de apuntadores compartidos (`shared_ptr`) a objetos de tipo robot. Se define también en este fichero un apuntador compartido a objetos de la clase `Swarm`. Como en el caso de la clase `Robot` hay una función `clone()` que permite hacer una copia profunda del objeto, aunque en este caso una copia profunda y superficial daría distintos resultados ya que el vector (swarm) guarda únicamente apuntadores a objetos de tipo `Robot` (u objetos de clases derivadas). Para añadir robots al enjambre se puede usar el método `push_back()` de la clase `vector` de la librería estandar, o el método `add()` de la clase `Swarm` (ver comentarios en el fichero de cabecera). Por útimo esta clase tiene un método `step()` que hace que el enjambre pase al siguiente estado, i.e. hace que todos los robots se muevan llamando al método `step()` de cada uno de ellos.

- `simulation.hh`: Este fichero declara la clase `Simulation` se ocupa de ejecutar la simulación del enjambre. Además la clase `Simulation` almacena el estado del enjambre en cada paso de la simulación de forma que pueda guardarse, visualizarse o analizarse. La simulación tiene una duración configurable a través del método `tMax()` que también devuelve el tiempo de simulación (por defecto es de 100s). El método `dt()` permite obtener y configurar el paso de simulación, que por defecto es 0.1s. El constructor de esta clase toma como argumento un apuntador a un enjambre que será el estado inicial de los robots. Otro método importante de la clase `Simulation` es `rndSwarm()` que genera una configuración aleatoria de robots en un enjambre. Los argumentos de este método son el número de robots a generar en el enjambre, un apuntador a un objeto de tipo `Robot` (o objeto de una clase derivada de `Robot`) y un apuntador al entorno en el que se simularán el enjambre. El método `run()` ejecuta la simulación del enjambre, y una vez simulado se puede acceder al estado del enjambre en cada paso de simulación. Para acceder al enjambre se ha sobrecargado el operador[3] `[]` para que acepte un entero sin signo (que representa el paso de simulación) o un número de coma flotante (`float`) (que representa el tiempo de simulación). Mientras que el tiempo de simulación se puede obtener a través del método `tMax()`, el número de pasos de simulación vienen dados por el método `size()`. La trayectoria completa del enjambre junto con el tiempo se puede almacenar en un fichero de texto a través del método `saveTraj()`, ver fichero de cabecera.

- `simulation-plot.hh`: Este fichero declara las clases necesarias para visualizar la evolución del enjambre que se puede controlar con el teclado para reproducir/pausar (barra espaciadora), aumentar (cursor arriba/derecha) disminuir (cursor abajo/izquierda) la velocidad de reproducción, y salir (escape), ver fichero de cabecera.

1.1 Instalación y compilación del software

El simulador depende de varias librerías que deben estar instaladas para poder compilarlo. La compilación del simulador creará además una librería estática (fichero `libMRS.a`) que se puede usar para enlazar al compilar programas que la usen. Los paquetes a instalar para poder compilar

[2] El método `clone()` hace una copia profunda (`deep copy`) del objeto, que en este caso es lo mismo que una copia superficial (*shallow copy*) ver https://en.wikipedia.org/wiki/Object_copying para más detalles.

[3] Ver https://en.wikipedia.org/wiki/Operator_overloading para detalles de la sobrecarga de operadores.

la librería del simulador son Eigen 3, gtkmm 3.0 y libYALM que se pueden instalar tecleando en una terminal:

```
$ sudo apt install libeigen3-dev
$ sudo apt install libgtkmm-3.0-dev libgstreamermm-1.0-dev
$ sudo apt install libyaml-dev
```

Una vez instaladas se puede compilar el código desde el directorio fuente ejecutando el comando `make`. El resultado será un fichero `libMRS.a` (la librería estática) y el programa `aggregationSwarm`. Si intentas ejecutar el programa verás que se abre una ventana que muestra un enjambre estático (al pulsar la barra espaciadora para visualizar la evolución del enjambre los robots no se mueven), ese es el comportamiento de la clase `Robot`, y debes modificar el código para simular tu nuevo robot (ver comentarios en el fichero `aggregationSwam.cc`).

2 Implementación de un robot gregario

El fichero `aggregationSwam.cc` contiene el patrón que puedes usar para implementar un enjambre que muestre el comportamiento de agregación. La clase `AggregationRobot` deriva de la clase `Robot` pero faltan por implementar una serie de métodos: el constructor, el método `action()` y el método `clone()`. Recuerda que las acciones del robot deben ser aleatorias con cierta probabilidad, lo cual se puede conseguir usando la función `rand()`. Por ejemplo, si se quiere que cierto evento ocurra con probabilidad $p = 0.3$ se puede generar un número aleatorio entre 0 y 1 y comprobar que el número es menor que 0.3. Como la función `rand()` genera número enteros se puede obtener un número en el rango $[0, 1]$ con el siguiente código en C++:

```
const int MAXIMO(1e6);
float num(float(rand() % (MAXIMO+1)));
num /= float(MAXIMO);
```

El fichero define las características de los robots que puede modificar para intentar conseguir un mejor agrupamiento del enjambre. También se incluye en la clase `AggregationRobot` el método `rndVel()` que asigna una velocidad aleatoria al robot. Aparte de las pautas enunciadas anteriormente para la implementación del robot gregario no hay restricciones en la complejidad del comportamiento, asimismo el informe en lugar de responder a preguntas concretas escribe simplemente cual es tu impresión personal de este método de agregación. Considera factores como su simplicidad, eficacia, escalabilidad, resultado final, limitaciones, y cualquier otro aspecto que te parezca relevante puedes ilustrar tu opinión con imágenes resultates de las simulaciones (p.e. usando el comando `import` para obtener *screenshots*). Indica en el comentario las probabilidades escogidas para cada caso (paro y cambio de dirección), los parámetros del robot y cualquier adaptación que hagas sobre el método original. Será conveniente que ejecutes varias simulaciones para hacerte una idea de su funcionamiento.

Recuerda enviar el código junto con tu breve reflexión sobre el método y sus resultados.