[Tutorial 1] Getting started with the robots' simulator

Fabien BADEIG

March 9, 2023

The objective of the tutorial is to understand the functioning of the core of the simulator of robots to be then autonomous as for the realization of its project.The core of the simulator defines the basic structures of the robots and their environment. It will be a matter of extending these structures to adapt them to your needs.

Description of the simulator

The simulator is composed of situated entities (class SituatedComponent) that evolve in a discretized environment (class GridEnvironment). We consider two types of entities: passive (like obstacles (class Obstacle)) and active (like robots (class Turtlebot)). The environment is a grid of cells (class Cell). It is possible to display the grid in a graphical window or on LED tables. In this case, a color cell (class ColorCell) grid (class ColorGridEnvironment) must be used. The simulation scheduling is provided by the class SimFactory with the method schedule.

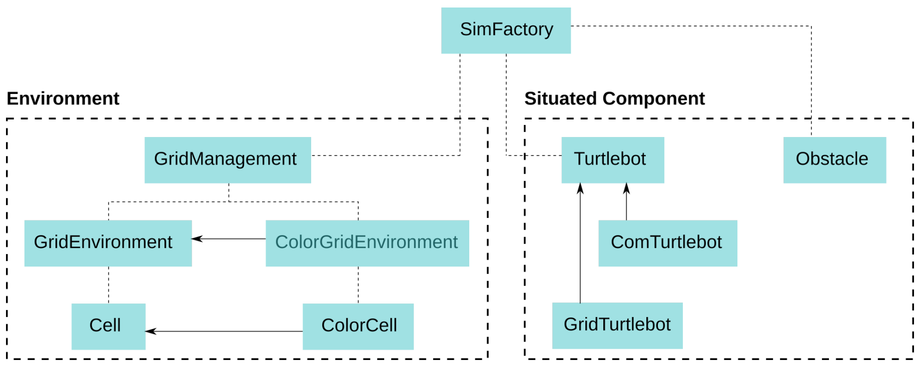

Figure 1: Functional view of the simulator

Grid Management and robot perception

The grid is a matrix of cells with n rows (x-axis) and m columns (y-axis). it is possible to add an element to the grid (i.e. change the content of a cell) or to move an element of the grid (i.e. move the content of a cell). The content of a cell is an integer corresponding to the identifier of a situated component or -1 for a non-existent cell or 0 for an empty cell.

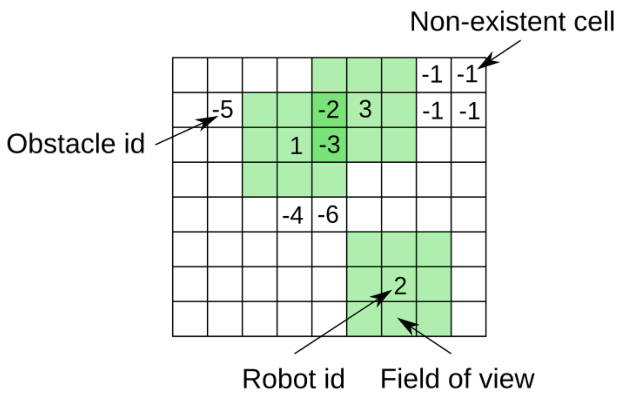

Figure 2: schematic view of environment

It is possible to have a representation of a part of the grid in a robot according to its field of view. When a robot updates its perception of the environment, it puts at the top of its grid what is in front of it, on the left what is on its left, . . .

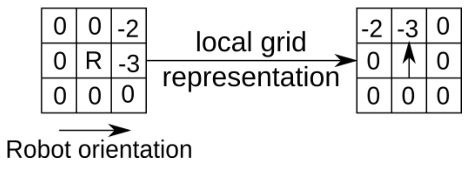

Figure 3: Grid perception for a robot

Communication between robots

The robots can communicate with each other. To do this, we need to extend the ComTurtlebot class. This class contains a list of received messages. The processing of received messages is done by the implementation of the abstract method *handleMessage(Message)*.

The process of message sending is specific because we have chosen to manage it in two steps. The robot has a list of messages to send and adds the messages to it. The sending of the messages will be done by the simulation manager (class *SimFactory*) which will call the method *receiveMessage(Message)* of the robots to add the message in the list of received messages.

Step 1: Creation of the JAVA project

The first step is to create a JAVA project and to import the maqitsimulator library (file maqitsimulator.jar). Then you have to create the file (configuration.ini) which contains the configuration parameters of your simulation. An .ini file is composed of

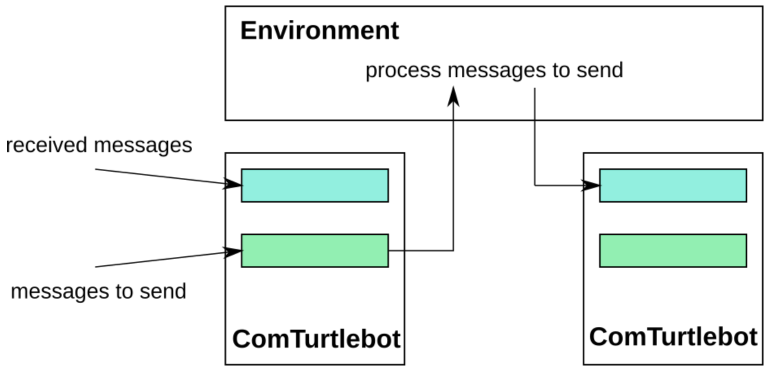

Figure 4: Communication between robots

sections and in each section the definition of a parameter is done by the syntax *parameter_name=parameter_value*. Below is a sample configuration file:

```
[configuration]
  display = 1
  simulation = 1
  mqtt = 0
  robot = 5
  obstacle = 15
  seed = 150
  field = 2
  debug = 1
  waittime = 500
  step=100

[environment]
  rows = 10
  columns = 10

[display]
  x = 10
  y = 210
  width = 800
  height = 800
  title = Display grid

[color]
  robot = 0,0,150
  goal = 50,50,50
  other = 0,0,120
  obstacle = 120,0,0
  unknown = 50,50,0
```

To load the configuration file, use the IniFile and SimProperties classes. You will test your code by displaying the simulation parameters in the console.

Step 2: Generation of a first environment for robots

To create an environment for the robots, an object of type GridManagement must be instantiated. The instantiation will be different if we want a graphical window (parameter *display*). At this step, the grid is empty and must be populated with obstacles and robots. To do this we will have to create a class that extends *SimFactory* and implement its abstract methods. The method to generate obstacles is *createObstacle()* or *createObstacle(int[] color)* (using parameter *nbobstacle*).

The basic process for creating an obstacle is:

- to retrieve a free position in the environment grid (method *getPlace()*)
- to create the obstacle with this position
- to add it to the list of obstacles in the simulation
- to update the simulation grid

You will test your code by displaying the grid of the simulation in the console or in the graphical window.

Step 3: Creation of a simple robot

To populate the simulation with robots, you must first create your robot class from the *GridTurtlebot* class. Implementing a robot class consists in defining its behavior in the method *move(int step)*. To implement a simple behavior, a robot will move forward if the cell in front of him is empty.

You will test your code by creating a robot and displaying its description in the console.

Now we will implement the abstract method of SimFactory to generate robots, called *createTurtlebot()* or *createTurtlebot(int[] color)* (using parameter *nbrobot*). The basic process for creating a robot is similar to that of the obstacles.

Step 4: Scheduling of the robots

To execute a simulation, you must define how the robots are activated. This process is implemented in the methode *schedule* of the class *SimFactory*. The basic principle is:

- to browse all the robots and for each robot:
- to retrieve the part of the grid corresponding to the perception of the robot
- to update the perception of the robot
- to make it act of a time step (method *move()* in *GridTurtlebot*)
- to finally update the environment

Step 5: Customization of its simulation environment

Create a custom simulation environment by changing the way obstacles are generated (method *createObstacle()* of *SimFactory*).