Sensor-Emitter Simulation Description

Adam Campbell

July 7, 2009

Contents

1 Introduction

This document describes the *Sensor-Emitter* code written in the Summer of 2009 by Adam Campbell[1] at the University of Central Florida. The document is organized as follows: Section 2 describes the details of the simulation's parameter file. Section 3 describes what happens during each time step of the simulation. Section 4 describes the output files that are written for each run, and finally, Section 5 lists some known issues with the simulator.

1.1 Brief overview of program

The program simulates a sensor network in a two-dimensional environment consisting of sensors and emitters. Both sensors and emitters are assigned one of N different frequencies, with sensors in frequency i being able to detect emitters in frequency i. Each sensor is described by its position in the environment, the frequency it is sensing, and its communication distance. Each emitter is described by its position, the direction of its signal, the distance the signal travels, the angle the signal spreads, and the frequency of the signal. The details of sensors and emitters are described in Sections 3.1 and 3.2, respectively.

1.2 Compiling/running the program

The simulation is written in Java[2] 1.5.0 and uses the MASON[3] Multiagent Simulation Toolkit Version 13. Make sure to install the latest version of the Java SDK before running and compiling the program. All simulation code can be found in the `mason/sim/app/sensor_emitter/` directory, and generic utility files can be found in `mason/sim/util_amc/`.

To compile the code, enter the `mason` directory and type:

javac sim/app/sensor_emitter/*.java

To run the simulation visualization program, enter the `mason` directory (if you're not already there) and type:

java sim.app.sensor_emitter.GraphicalUserInterface

To run the simulation without any visualization (useful if you want to gather statistics by running a batch of runs), enter the `mason` directory and type:

java sim.app.sensor_emitter.Simulation -for [number of steps]

Where *number of steps* is an integer value indicating the number of steps to run the simulation.

2 Parameter file

The simulation is controlled by the `run.conf` parameter file found in the `mason` directory. The parameter file allows the user to specify general simulation parameters, output parameters, frequency parameters, sensor parameters, and emitter parameters. These parameters are described in more detail below. Details can also be found in the `run.conf.README` file in the `mason` directory.

2.1 General simulation parameters

2.1.1 `FIELD_WIDTH` and `FIELD_HEIGHT`

`FIELD_WIDTH` and `FIELD_HEIGHT` specify the width and height of the simulation environment. Both parameters take `int` values.

[1] Send any inquiries to acampbel@cs.ucf.edu
[2] http://java.sun.com/
[3] http://www.cs.gmu.edu/ eclab/projects/mason/

2.1.2 DISPLAY_WIDTH and DISPLAY_HEIGHT

DISPLAY_WIDTH and DISPLAY_HEIGHT specify the width and height of the visualization window that is shown on the screen. Note, this does not have to be the same as FIELD_WIDTH and FIELD_HEIGHT. Both parameters take `int` values.

2.1.3 SENSOR_DISCRETIZATION and EMITTER_DISCRETIZATION

SENSOR_DISCRETIZATION and EMITTER_DISCRETIZATION are used by the `Continuous2D` class in the MASON package. These parameters affect the performance of the simulation, as they determine the number of agents that are returned whenever a `getObjectsWithinRange()` function is called on a `Continuous2D`. I typically set SENSOR_DISCRETIZATION to the maximum communication distance of any sensor, and I set EMITTER_DISCRETIZATION to the maximum distance that any emitter emits its signal. Finding optimal values for these is probably tricky, so I don't really spend too much time tweaking them. Both parameters take `int` values.

2.1.4 BACKGROUND_COLOR

BACKGROUND_COLOR specifies the color of the simulation's background. It takes three `int` values in the range $[0, 255]$ specifying the color's red, green, and blue values.

2.2 Output parameters

2.2.1 WRITE_DATA

WRITE_DATA indicates whether or not to write output for this simulation run. Details on what is written out is described in Section 4. This parameter takes a `boolean` value, i.e., `true` or `false`.

2.2.2 DATA_WRITE_INTERVAL

DATA_WRITE_INTERVAL specifies how often output is written to the output files. When set at 1, data is written every single time step. I wouldn't suggest setting this value to anything other than 1. This parameter takes positive `int` values.

2.2.3 OUTPUT_DIR

OUTPUT_DIR is a `String` indicating the directory that the output is written (if WRITE_DATA is set to `true`).

2.3 Frequency params

2.3.1 NUM_FREQUENCIES

NUM_FREQUENCIES specifies the number of frequencies that are used by sensors and emitters in the simulation. It takes a positive `int` as its value.

2.3.2 SENSOR_COLOR_i and EMITTER_COLOR_i

For each integer $1 \leq i \leq NUM\_FREQUENCIES$, there should be a SENSOR_COLOR_i and EMITTER_COLOR_i parameter specifying the colors of sensors and emitters, respectively. When a sensor is in frequency i, it is drawn as a SENSOR_COLOR_i colored filled circle. When an emitter is in frequency i, its signal is drawn in color EMITTER_COLOR_i. Each SENSOR_COLOR_i and EMITTER_COLOR_i parameter take in three or four `int` values in the range $[0, 255]$. The first three values specify the red, green, blue values for the color, and the optional fourth value specifies the alpha value for the color. I usually don't use the alpha value, as it slows down the simulation, but when I do use it, I use it for the emitters.

2.4 Sensor parameters

2.4.1 NUM_SENSOR_GROUPS

Sensors are created in groups. This allows the user to give different behaviors to groups or individual sensors. NUM_SENSOR_GROUPS specifies the number of sensor groups in the simulation. For each integer $1 \leq i \leq NUM\_SENSOR\_GROUPS$, there should be the following parameters:

- SENSOR_POSITIONER_i
- SENSOR_PHYSICAL_RADIUS_i
- SENSOR_COMM_DISTANCE_i
- SENSOR_UPDATE_PROBABILITY_i
- SENSOR_INITIAL_FREQUENCY_ASSIGNER_i
- SENSOR_MOVER_i
- SENSOR_FREQUENCY_UPDATER_i

These parameters specify how sensor group i is created and are described in detail below. NUM_SENSOR_GROUPS takes a non-negative `int` value.

2.4.2 SENSOR_POSITIONER_i

SENSOR_POSITIONER_i determines how many sensors to make and where to put those sensors. It can take on one of two values: `SinglePositioner` and `GridPositioner`.

`SinglePositioner` is used to create one sensor at the specified location in the environment. Its form is as follows:

SinglePositioner [x] [y]

where `x` and `y` are the coordinates of the sensor. These two parameters are of type `double`.

`GridPositioner` is used to create a rectangular grid of sensors. Rows and columns are evenly spaced in the rectangle specified by the parameters. Its form is as follows:

GridPositioner [rows] [cols] [xAnchor] [yAnchor] [width] [height]

where `rows` and `cols` are the number of rows and columns of sensors that are to be created, `xAnchor` and `yAnchor` are values indicating the position of the up-left-most sensor, and `width` and `height` are the width and height of the rectangle that the sensors are created in. `rows` and `cols` are of type `int`, and the other four parameters are of type `double`

2.4.3 SENSOR_PHYSICAL_RADIUS_i

SENSOR_PHYSICAL_RADIUS_i specifies how big to draw the sensors in the simulation. It has no effect on the dynamics of a simulation as sensors are always treated as point objects when computing the distance between sensors-sensors and sensors-emitters. It takes a `double` value.

2.4.4 SENSOR_COMM_DISTANCE_i

SENSOR_COMM_DISTANCE_i specifies how far each sensor broadcasts information to its neighbors. All enabled sensors within the range of a broadcasting sensor receive the message. The performance of the simulation decreases as this parameter and the number of sensors increases. Sensors use the information from their neighbors to decide which frequency to sense. This parameter takes a `double` value.

2.4.5 SENSOR_UPDATE_PROBABILITY_i

SENSOR_UPDATE_PROBABILITY_i gives the probability that a sensor update its frequency during each time step. When this value equals 1.0, the sensors call their frequency updater every single time step. As this value decreases, so too, does the average number of sensors that perform an action each time step. This value takes a `double` in the range $[0, 1]$. Note, if the value is 0.0, sensors won't change their frequency.

2.4.6 SENSOR_INITIAL_FREQUENCY_ASSIGNER_i

SENSOR_INITIAL_FREQUENCY_ASSIGNER_i determines how the sensors are initially assigned their frequencies and can take one of the following two values: `RandomAssigner` and `HomogeneousAssigner`.

`RandomAssigner` assigns random frequencies to each of the sensors in the group. It does not take additional parameters.

`HomogeneousAssigner` is used to assign the same frequency to each sensor in the group. Its form is as follows:

HomogeneousAssigner [frequency]

where `frequency` is of type `int` in the range $[0, NUM\_FREQUENCIES - 1]$ and specifies which frequency to assign all of the sensors. Note that the frequencies used throughout the simulation code are zero-based.

2.4.7 SENSOR_MOVER_i

SENSOR_MOVER_i determines how the sensors move in their environment. It can take one of three values: `StationaryMover`, `FixedWaypointMover`, and `RandomWaypointMover`.

`StationaryMover` does not take any additional parameters and is used when you don't want the sensors to move.

`FixedWaypointMover` is used to move a sensor repeatedly between a fixed number of waypoints. Its form is as follows:

FixedWaypointMover [x1] [y1] [x2] [y2] ... [xn] [yn]

where `xi` and `yi` are the i^{th} waypoint. Once the sensor has reached waypoint i, it moves on to waypoint $i + 1$. When it reaches waypoint n, it cycles back to waypoint 1. All `xi` and `yi` are of type `double`.

`RandomWaypointMover` moves the sensors around randomly within the specified rectangle. A waypoint is randomly generated in the given area, and once the sensor reaches that point, a new waypoint is randomly generated. This process is continuously repeated. Its form is as follows:

RandomWaypointMover [xAnchor] [yAnchor] [width] [height]

where `xAnchor` and `yAnchor` are values indicating the position of the up-left-most sensor, and `width` and `height` are the width and height of the rectangle that the sensors move in. Each of these parameters is of type `double`.

2.4.8 SENSOR_FREQUENCY_UPDATER_i

SENSOR_FREQUENCY_UPDATER_i indicates the algorithm that the sensors use to determine which frequency to sense. It can take one of the following four values: `StationaryFrequencyUpdater`, `Method1FrequencyUpdater`, `Method2FrequencyUpdater`, and `Method3FrequencyUpdater`.

`StationaryFrequencyUpdater` does nothing to the frequencies, i.e., the sensors never sense anything other than their initially assigned frequency.

`Method1FrequencyUpdater` uses the messages that a sensor currently has to count the number of neighboring sensors in each one of the frequencies. The sensor also takes into account its own frequency when making these counts. With these counts, the sensor determines the frequencies that are the least represented, and then chooses randomly amongst one of these frequencies.

`Method2FrequencyUpdater` differs from `Method1FrequencyUpdater` in two ways: First, the sensor does not take into account its own frequency when obtaining the counts. Second, if its own frequency is one of the least represented, then the sensor stays in its current frequency. This helps to reduce the number of times sensors switch their frequencies.

`Method3FrequencyUpdater` has the sensor randomly choose from one of the frequencies. Note, the sensor could choose its own frequency when randomly choosing one.

More details on the three previous frequency updating algorithms can be found in Campbell and Wu [1].

2.5 Emitter parameters

2.5.1 NUM_EMITTER_GROUPS

Like sensors, emitters are created in groups. This allows the user to give different behaviors to groups or individual emitters. `NUM_EMITTER_GROUPS` specifies the number of sensor groups in the simulation. For each integer $1 \leq i \leq NUM\_EMITTER\_GROUPS$, there should be the following parameters:

- `EMITTER_POSITIONER_i`
- `EMITTER_PHYSICAL_RADIUS_i`
- `EMITTER_SIGNAL_INFO_i`
- `EMITTER_INITIAL_FREQUENCY_ASSIGNER_i`
- `EMITTER_SIGNAL_CONTROLLER_ASSIGNER_i`
- `EMITTER_MOVER_i`

These parameters specify how emitter group i is created and are described in detail below.
`NUM_EMITTER_GROUPS` takes a non-negative `int` value.

2.5.2 EMITTER_POSITIONER_i

`EMITTER_POSITIONER_i` determines how many emitters to make and where to put those emitters. It can take on one of two values: `SinglePositioner` and `GridPositioner`. The details of these are described above in Section 2.4.2.

2.5.3 EMITTER_PHYSICAL_RADIUS_i

`EMITTER_PHYSICAL_RADIUS_i` specifies how big to draw the emitters in the simulation. It has no effect on the dynamics of a simulation as emitters are always treated as point objects when computing the distance between sensors and emitters. It takes a `double` value.

2.5.4 EMITTER_SIGNAL_INFO_i

`EMITTER_SIGNAL_INFO_i` specifies the signal properties for the emitters in the group. It takes four parameters, `emitDirectionX`, `emitDirectionY`, `emitDistance`, and `emitSpread`.
`emitDirectionX` and `emitDirectionY` determine the initial direction of the emitters, `emitDistance` is the distance that the signal travels, and `emitSpread` is the total angle that the signal spreads. All four of these parameters are of type `double`.

2.5.5 EMITTER_INITIAL_FREQUENCY_ASSIGNER_i

`EMITTER_INITIAL_FREQUENCY_ASSIGNER_i` determines how the emitters are initially assigned their frequencies and can take one of the following two values: `RandomAssigner` and `HomogeneousAssigner`. The details of these are described above in Section 2.4.6.

2.5.6 EMITTER_SIGNAL_CONTROLLER_i

EMITTER_SIGNAL_CONTROLLER_i determines how the emitters direct their signals. It takes one of two values: StationarySignalController and FixedPointSignalController.

StationarySignalController does nothing to the initial directions of the emitters' signals. It takes no additional parameters.

FixedPointSignalController allows the user to specify a point in the environment that the emitter(s) always directs its signal towards. Its form is as follows:

FixedPointSignalController [targetX] [targetY]

where targetX and targetY are the coordinates of the point that the emitter points its signal towards. These two values are of type double.

2.5.7 EMITTER_MOVER_i

EMITTER_MOVER_i determines how the emitters move in their environment. It can take one of three values: StationaryMover, FixedWaypointMover, and RandomWaypointMover. The details of these are described above in Section 2.4.7.

3 Simulation details

During each time step of the simulation, the sensors and emitters are randomly ordered and processed one at a time. That is, both sensors and emitters are updated asynchronously, and from one time step to the next, the order that the agents are updated differs. The following two subsections describe what happens to each sensor and emitter during each step of the simulation.

3.1 Sensor behavior

A sensor that is not disabled performs the following actions each time step: First, with probability SENSOR_UPDATE_PROBABILITY_i (see Section 2.4.5) the sensor updates its frequency using its frequency updater (see Section 2.4.8). Next, it clears all of the incoming messages that it had from neighboring sensors. After this it moves to a new location using its agent mover (see Section 2.4.7), and then it obtains the list of emitters that it can sense. Finally, it broadcasts information about its current frequency to neighboring sensors.

If a sensor is disabled, it simply clears all of its incoming messages.

3.2 Emitter behavior

An emitter that is not disabled performs the following actions each time step: First, it moves to a new location using its agent mover (see Section 2.5.7). Finally, it uses its signal controller to control the parameters of its signal (see Section 2.5.6).

A disabled emitter does nothing.

4 Output files

If WRITE_DATA is true, then the files described in the remainder of this section are written in OUTPUT_DIR. See Section 2.2 for more information on these parameters.

4.1 runX.meta

This file gives the random number generator's seed for this run along with a time stamp indicating when the run took place. The seed can be used to duplicate runs.

4.2 `runX.params`

This file is a copy of the parameter file used for the run.

4.3 `runX.sensedCount`

For each time step of the simulation, a line is printed out containing the number of sensors that are sensing each emitter. Thus, if there are four emitters, there will be five columns in this file (one for the time step, plus a column for each emitter).

4.4 `runX.sensingSensors`

A line is printed in this file for each sensor that senses an emitter during any time step of the simulation. Each line contains the time step that the sensor sensed the emitter, the sensor's id, the emitter's id, and the frequency of the sensor when it sensed the emitter. Note, a line is printed for each time step that a sensor senses an emitter, even if a sensor senses the same emitter for multiple, consecutive time steps.

5 Known issues

5.1 Visualization freezes

The program sometimes freezes when first hitting the start button on the visualizer. The program does not respond, even when trying to close it out by clicking the window's X button. To close the program when this happens, hit `Ctrl-C` at the command prompt where you started the program. This bug is being looked in to.

References

[1] Adam Campbell and Annie S. Wu. On the significance of synchroneity in emergent systems. In *Proceedings of the Eighth International Conference on Autonomous Agents and Multiagent Systems*, pages 449–456, 2009.