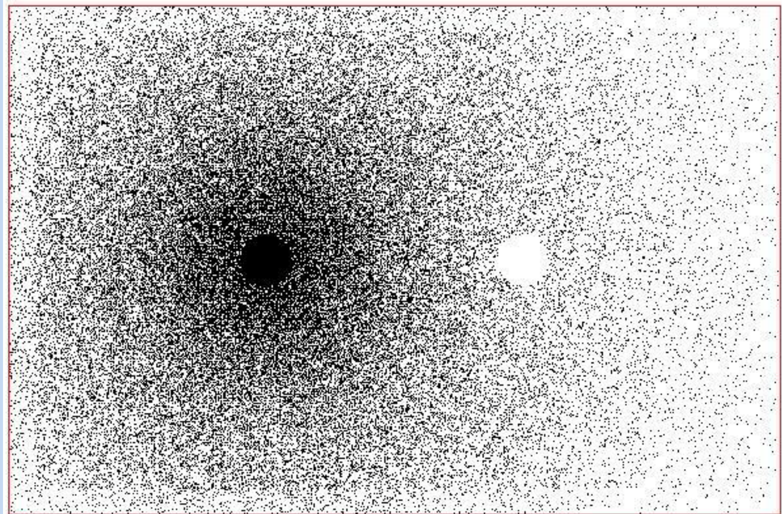
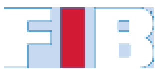
[2011]

N3Sim: Simulator for diffusion-based molecular communications in Nanonetworks





Title: *N3Sim : Simulator for diffusion-based molecular communications in Nanonetworks*

Author: *Iñaki Pascual*

Date: *June 2011*

Director: *Dr. Albert Cabellos-Aparicio*

Director's Department: *Arquitectura de Computadors (DAC)*

Co-Director: *Ignacio Llatser Martí*

Co-Director's Department: *Arquitectura de Computadors (DAC)*

Degree: *Informatics Engineering*

School: *Facultat d'Informàtica de Barcelona (FIB)*

University: *Universitat Politècnica de Catalunya (UPC) BarcelonaTech*

ABSTRACT

Nanotechnology is enabling the development of devices in a scale ranging from one to a few hundred nanometers, known as nanomachines. How these nanomachines will communicate is still an open debate. Molecular communication is a promising paradigm that has been proposed to implement nanonetworks, i.e., the interconnection of nanomachines. The peculiarities of the physical channel in diffusion-based molecular communication require the development of novel models, architectures and protocols for this new scenario, which need to be validated by simulation.

With this purpose, N3Sim, a simulator for diffusion-based molecular communication has been developed. N3Sim allows simulating scenarios where transmitters encode the information by releasing molecules into the medium, thus varying their local concentration. N3Sim models the movement of these molecules according to Brownian dynamics, and it also takes into account their inertia and the interactions among them. Receivers decode the information by sensing the particle concentration in their neighborhood.

The benefits of N3Sim are multiple: the validation of channel models for molecular communication and the evaluation of novel modulation schemes are just a few examples.

CONTENTS

1. INTRODUCTION. 6

1.1. CONTEXT. 6

1.2. DIFFUSION-BASED MOLECULAR COMMUNICATION IN NANONETWORKS. 6

1.3. PROJECT DEVELOPMENT. 10

1.4. GOALS. 10

2. STATE OF THE ART. 12

3. REQUERIMENTS ANALYSIS 14

3.1. REQUERIMENTS ELICITATION STRATEGY. 14

3.2. NON FUNCTIONAL REQUERIMENTS. 16

3.2.1. RELIABILITY. 16

3.2.2. MODIFIABILITY. 16

3.2.3. EXTENSIBILITY. 16

3.2.4. USABILITY. 16

3.3. FUNCTIONAL REQUERIMENTS. 17

3.3.1. SIMULATION SPACE. 17

3.3.2. COLLECTIVE DIFFUSION. 17

3.3.3. EMITTER. 17

3.3.4. RECEIVER. 18

3.3.5. INITIAL BACKGROUND CONCENTRATION. 18

4. SPECIFICATION. 19

4.1. USE CASE MODEL. 19

4.2. CONCEPTUAL MODEL. 20

5. DESIGN AND IMPLEMENTATION 23

5.1 INTRODUCTION **23**

5.2 THREE LAYER ARCHITECTURE **24**

5.3 PACKAGE STRUCTURE **24**

5.4 IMPLEMENTATION. **26**

5.4.1 USER INTERFACE LAYER. **26**

5.4.3 DOMAIN LAYER. **28**

5.4.4 DATA LAYER. **41**

5.5 ELECTROSTATIC FORCES **42**

5.6 N3SIM VIDEO PLAYER (N3SVIDEO). **43**

6. COLLISION DETECTION ALGORITHM 44

6.1. INTRODUCTION **44**

6.2. STATE OF THE ART **44**

6.3. IMPLEMENTATION **48**

6.4. COST ANALYSIS **51**

7. RESULTS. 53

7.1. BROWNIAN MOTION **53**

7.2. SPACE LIMITS **54**

7.3. NOISE **55**

7.4. COLISSIONS. **56**

7.5 INERTIA........ **57**

7.6. ELECTRICAL FORCES. **58**

8. FUTURE WORK. 59

8.1. PACKAGE STRUCTURE **59**

8.2. ELECTRICAL FORCES. **62**

8.3. BOUNDARIES. **63**

8.4. EXTENSION TO 3D. 63

9. CONCLUSIONS. 64

9.1. ACHIEVING GOALS. 64

9.2. PROJECT DEVELOPMENT. 66

9.3. STUDY OF THE MOLECULAR COMMUNICATION CHANNEL. 66

9.4. PERSONAL ASSESSMENT. 67

10. REFERENCES 68

ANNEX I : PUBLICATIONS DERIVED FROM THIS PROJECT 70

ANNEX II : QUICK START GUIDE 71

ANNEX III : USER GUIDE 77

ANNEX IV : LIST OF PARAMETERS 88

1. INTRODUCTION.

1.1. CONTEXT.

The project began in the DAC (Departament d'Arquitectura de Computadors of the Universitat Politècnica de Catalunya) within the group N3Cat (NaNoNetworking Center of Catalonia). N3Cat collaborates with the Georgia Institute of Technology in the study of communication among nanomachines (nanonetworks).

Molecular communication is a promising type of communication among nanomachines [1]. N3Cat recently prepared the paper [2] which proposes an end-to-end model for molecular communication among nanomachines.

It is not feasible to validate the theoretical model of communication channel through direct experimentation. So the group proposed to build a simulator to validate the theory and characterize the channel. This simulator, N3Sim, is the subject of this project.

1.2. DIFFUSION-BASED MOLECULAR COMMUNICATION IN NANONETWORKS.

Richard Feynman, Nobel Prize in Physics in 1965, settled the basics of nanotechnology in his famous lecture "There's plenty of room at the bottom" made in 1959. He explained his vision of how humans would create increasingly smaller and more efficient devices in the future. It was not until 15 years later, in 1974, that the term nanotechnology was coined to express the technology capable of working with materials at the atomic level.

Since then, the concepts of nanotechnology were developed slowly until nanotechnology investigation received a new impetus at the beginning of this century. The impetus is largely the result of technology development, which has provided techniques that allow working at the nanoscale. Some examples are the microchip industry, research on DNA and development of nanomaterials such as graphene and carbon nanotubes.

Currently nanotechnology is defined as the technology that deals with techniques that allow developing devices in the scale of one to a few hundred nanometers. At this scale nanomachines are defined as autonomous units that are able to perform a simple computation, actuation or sensing task [1].

With the increasing research in nanotechnology, nanonetworking becomes an emerging field. Nanonetworks are networks of nanomachines that may work collaboratively by communicating with each other and / or with a control system of a larger scale.

Communication among nanomachines will allow them to perform more complex tasks than a single nanomachine can do, and to extend their workspace. The workspace of a nanomachine is intrinsically very small, but most of the potential applications of nanomachines require operating on an area of higher scale. The solution for this contradiction is the work of a large number of nanomachines that can communicate in some degree to coordinate their work.

In-body drug delivery is an example of possible nanonetworks applications. The idea is that nanomachines working inside the human body could administer drugs according to the measurements, made by other nanomachines, of variables that determine the patient's condition. With this purpose nanomachines capable of detecting levels of certain substances and nanomachines able to produce and deliver drugs should work together.

Biodegradation, water and air quality monitoring, advanced fabrics and materials or defense against biological attacks are other examples of applications of nanonetworks [1].

Three different strategies in the development of nanomachines are presented in [1]: the **top-down** approach, the ***bottom-up*** approach and the ***bio-hybrid*** approach. The ***top-down*** approach is based on the increasing miniaturization of electromechanical devices. The ***bottom-up*** approach is based on the construction of nanomachines from basic blocks, which would be molecules. The ***bio-hybrid*** explores the possibilities of building nanomachines from existing biological structures. A cell can be seen as a nanomachine that performs sensing, production, control and locomotion tasks.

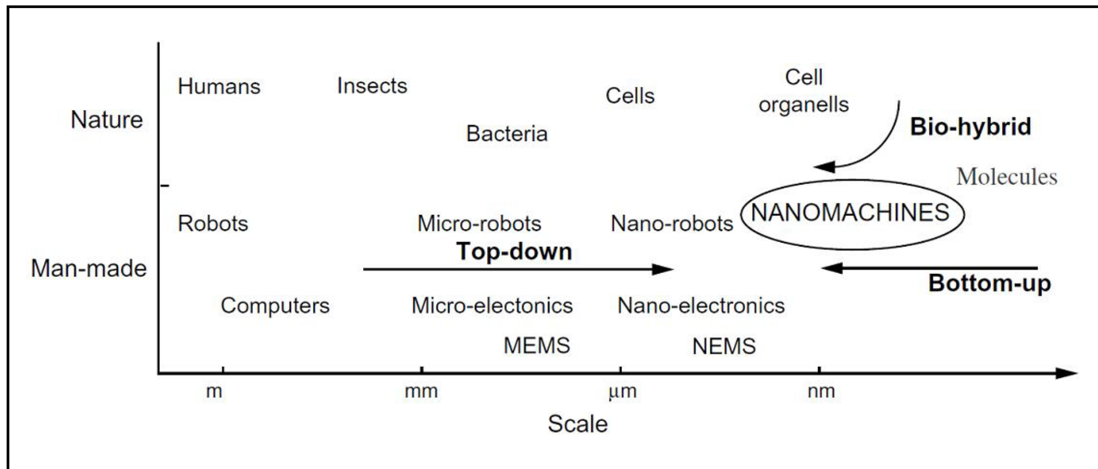


Figure 1.1. Approaches for the development of nano-machines. [1].

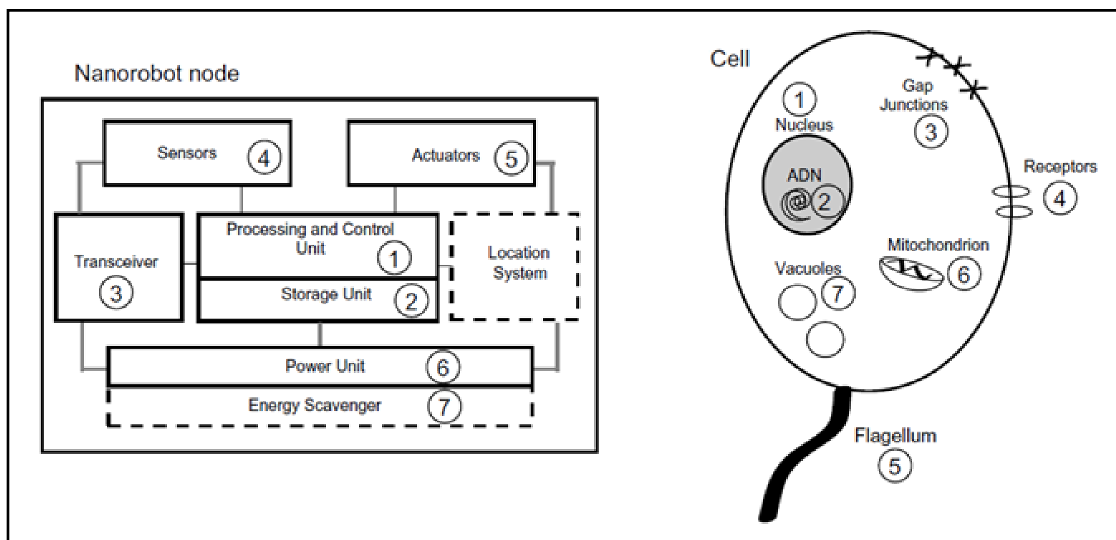


Figure 1.2. Bio-hybrid approach. Functional architecture mapping between nano-machines of a micro or nano-robot and nanomachines found in cells [1].

The main types of communication in nanonetworks are electromagnetic and molecular. Following the ***bio-hybrid*** approach, molecular communication is the most promising type of communication for nanonetworks.

One way to transport the information in molecular communication is by diffusion. Diffusion is the random movement that any particle suspended particle in a fluid experiments. The information may be encoded in the molecule and it is transported from one nanomachine to another by diffusion or in the concentration of suspended particles in the fluid. In the latter case the emitter modifies the concentration by releasing or absorbing particles, and diffusion transports this change of concentration. An example of this case is the calcium ion signaling to intercellular communication observed in the human body. The model developed in [2] is based on the latter.

Diffusion is the process by which particles suspended in a fluid experience a random endless movement. The basic diffusion process is based on Brownian motion. Brownian motion was first described by Jan Ingenhousz in 1785 when he observed the irregular movement of coal dust particles on the surface of alcohol. Nevertheless Brownian motion is traditionally regarded as discovered by the botanist Robert Brown, from whom it takes its name. In 1827, Brown described the jittery motion of minute particles ejected from pollen grains suspended in water.

This movement is due to interactions among the fluid particles, which are in continuous motion, and the suspended particles. The microscopic dynamics of this process is extremely complex. It was Einstein in 1905 who first suggested equations to describe this process from a macroscopic point of view. Einstein's equation describes the diffusion as a stochastic process in which the movement of a set of particles for a time T due to Brownian motion follows the equation:

$$X_{rms} = \sqrt{2Dt}$$

Where X_{rms} is the root-mean-square of the displacement in the direction of the X-axis and D is the diffusion coefficient, defined as

$$D = \frac{KT}{6\pi R\upsilon}$$

Where K is the Boltzman constant, T is the temperature, R is the suspended particle radius and υ is the fluid viscosity.

The same can be applied to the Y-axis and the Z-axis, as each dimension has an independent contribution to the displacement.

However, Einstein equations are valid only for suspended particles that have no interaction among them. When there are interactions among particles the diffusion, diffusion equations must be modified and the process is known as *collective diffusion*. Among the forces that may alter the diffusion process are the collisions between particles, electrical and chemical forces. The size of the suspended and fluid particles may also alter the diffusion.

These effects may increase or reduce the displacement of suspended particles, processes known as superdiffusion and subdiffusion, respectively.

1.3. PROJECT DEVELOPMENT.

This project begins with a general description of a simulator that allows studying the communication channel described in [2]. But there are still questions without answer. Questions about the diffusion process, about the molecular communication channel itself and about how this channel can be modeled with a simulator.

The N3Sim group decided to develop the simulator by incremental prototypes, and use these prototypes to help solving the previous doubts and gradually implement the functionalities that are decided as necessary.

The N3Sim group meets once a week for about half a year to discuss about the communication channel, the diffusion process and the simulator. At the same time, experiments with the simulator are carried out that help drawing conclusions about the molecular communication channel. These conclusions in turn help define the specifications of the simulator.

1.4. GOALS.

This main goal of this project is to build a simulator that allows

- Validating the model of molecular communication channel proposed in [2]. In this model the channel model is divided in three modules: emission, diffusion and reception. The goal is to model the module B, the diffusion process, leaving for future developments the other two modules.
- Study the physical layer of the molecular communication, so that modulations of the signal and communication mechanism can be proposed.
- Characterize the diffusion-based molecular communication channel measuring its main metrics [9].

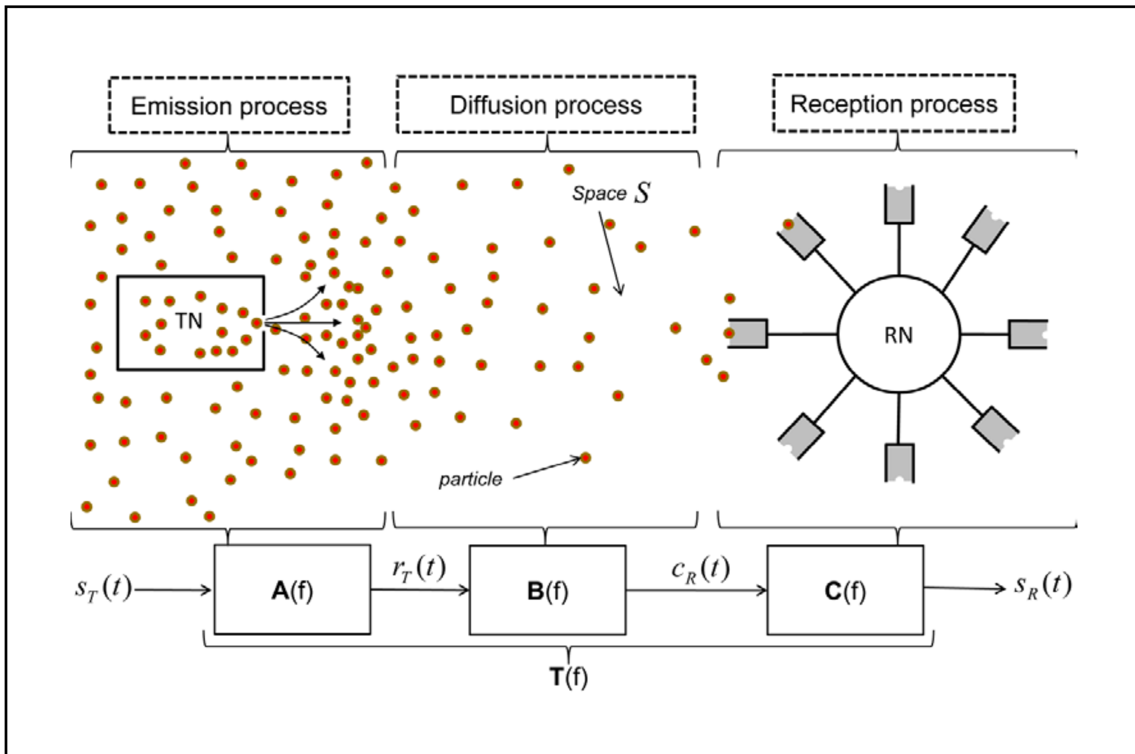


Figure 1.3. The three modules of the molecular communication channel described in [2]. N3Sim is initially built to model the module B (diffusion process) but open to incorporate the other two modules.

The simulator must be modifiable and extensible enough, so that in the future it is possible to include the modules A (transmitter) and C (receiver) proposed in [2].

2. STATE OF THE ART.

Nanonetworks and molecular communication are rather new research fields. Because of this there are few applications of simulation for this field.

It is easy to find simulators of diffusion based on Brownian motion. Most of them have an educational point of view. Some examples are the applets that can be found at:

http://galileoandeinstein.physics.virginia.edu/more_stuff/Applets

http://www.chm.davidson.edu/vce/kineticmoleculartheory/diffusion.html / brownian / brownian.html.

A very interesting example is shown in [15]. However, these applications are merely displays and do not incorporate the concepts of emitters and receivers, neither collective diffusion, because they are based only on Brownian motion diffusion.

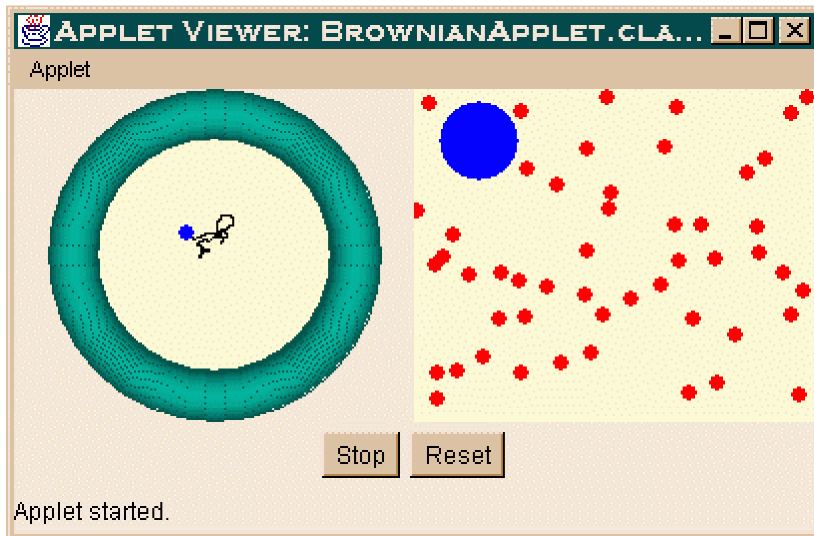


Figure. 2.1. Simulator of diffusion based on Brownian Motion from *http://galileo.phys.virginia.edu/classes/109N/more_stuff/Applets/brownian/brownian.html.*

The two most interesting applications found are the simulator developed by Michael Moore (available at *http://www.ics.uci.edu/ ~ Mikeman /*) and *NanoNS* [14]. Michael Moore simulator studies the diffusion including a transmitter and a receiver. But the simulation is performed for only one molecule, it is not valid for the simulation

of the diffusion of a set of particles. Besides, it models diffusion based only on Brownian motion.

NanoNS is the most advanced simulator we have found. A very interesting aspect of this simulator is that it has been built as a module of the NS network simulator (Network Simulator, see *http://www.nsnam.org/*) which will facilitate the future study of higher layers of the molecular communication model.

However, NanoNS has three drawbacks to the goals of this project. First, it models diffusion but not collective diffusion. Second, it uses Fick's laws of diffusion. Fick's laws of diffusion works with the flux of particles between adjacent volume. This is a useful higher abstraction of the diffusion process, but it does not allow studying some parts of the physical layer of the molecular communication. For instance, it is not possible to observe the noise inherent to molecular communication using Fick's laws.

The third drawback is that NanoNS joins the diffusion process and the reception process in one equation which is used by the simulator algorithms. From the point of view of the goals of this project this is not appropriate. On the other hand N3Sim is developing only the diffusion process and it makes easier making changes.

3. REQUERIMENTS ANALYSIS

3.1. REQUERIMENTS ELICITATION STRATEGY.

As explained in chapter 1.3, when the project started, the objectives and requirements of the project were quite clear. However there were still some unanswered questions regarding the collective diffusion, the communication channel and how to implement it in a simulator. For instance, the space, should it be limited? There was not an answer at the beginning.

The method used to answer these questions and define the requirements has been a spiral development including a phase of rapid prototyping. Prototypes are used to find the software functionalities. Figure 3.1 shows the steps of the spiral developing that has been used in the project. The spiral development of the project along with the rapid prototyping is used to define the requirements.

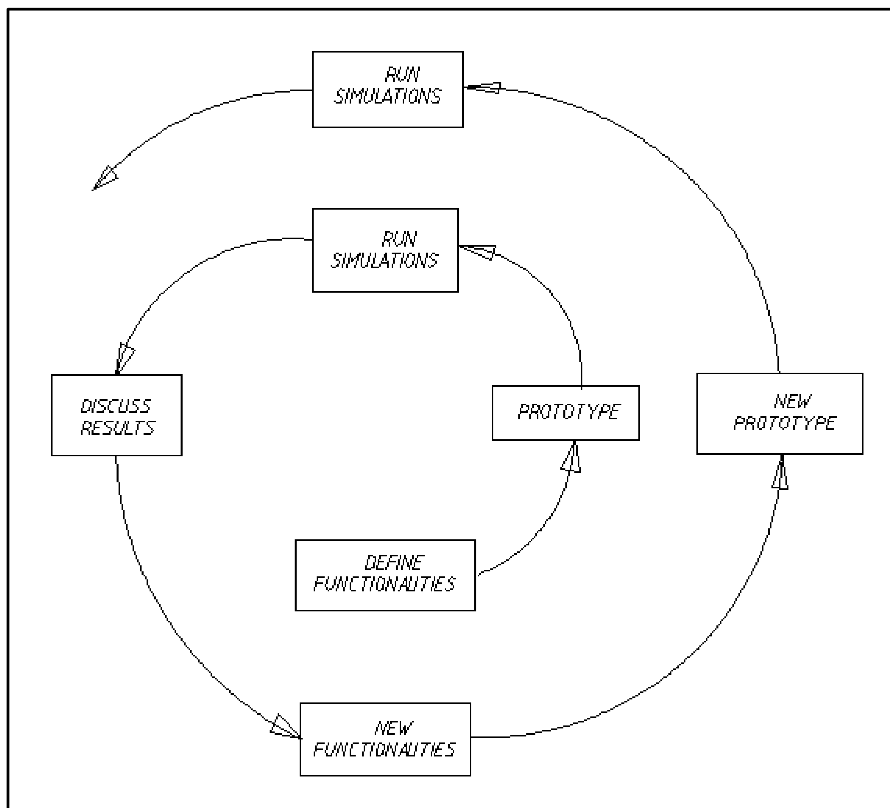


Figure.3.1. Spiral development including a phase of rapid prototyping.

For instance, regarding the question of open or closed simulation space, both possibilities were implemented and graphs were obtained as in Figure 3.2. The received signal when pulses of molecules are emitted in a limited space increases its amplitude as time increases. It follows that it is necessary to model an open space. While on the other side, to meet the requirement of initial background concentration described in section 3.3.5, a closed space is required. The resolution to this contradiction is explained in the section 5.4.3.2.

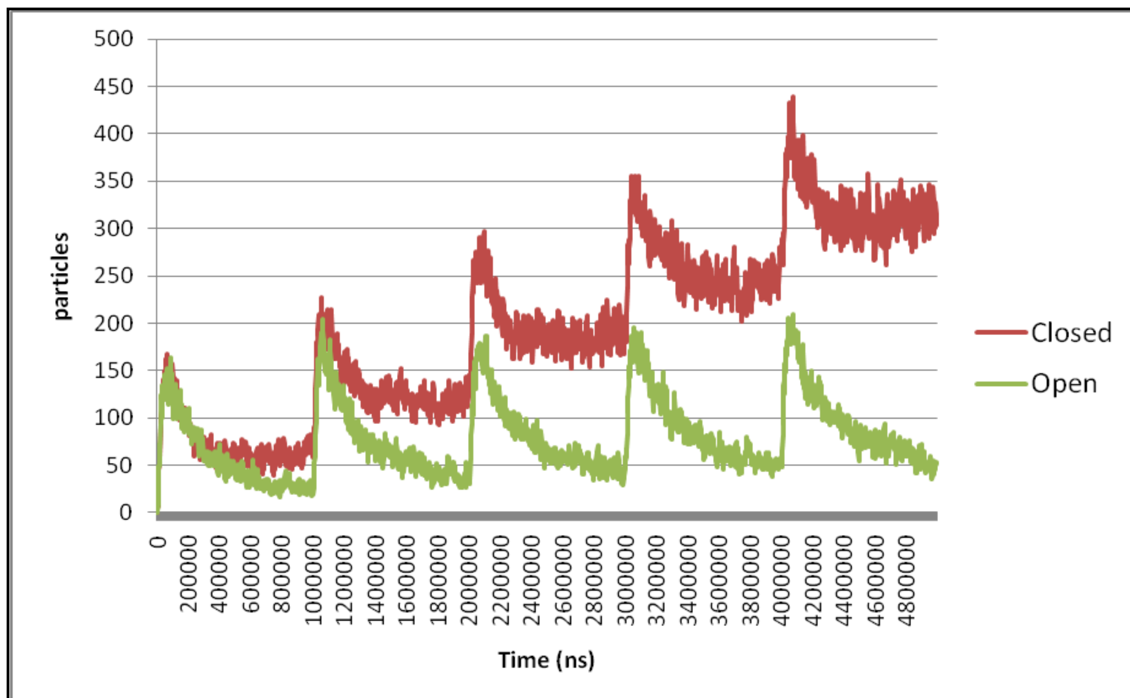


Figure.3.2. The open model is more useful for the study of molecular communication. But a problem for the requirement to simulate a background concentration.

Once a week the group discussed the results of the experiments carried out with the simulator and proposed next features for the simulator.

3.2. NON FUNCTIONAL REQUERIMENTS.

3.2.1. RELIABILITY.

In this project, reliability implies that the simulator must reproduce as faithfully as possible the laws of collective diffusion which are the basis of molecular communication. It means that no errors are acceptable in any case.

Since the simulator will be a research tool, if this requirement was not accomplished, the results of the work carried out with this simulator could be invalidated.

3.2.2. MODIFIABILITY.

Modifiability is an essential requirement due to the spiral development explained in section 3.1. Moreover, some functionalities have been implemented and later discarded or modified.

3.2.3. EXTENSIBILITY.

It is a very important requirement because the second objective outlined in section 1.4 indicates that the simulator must be capable to be extended to include models for receivers and transmitters as described in [2].

3.2.4. USABILITY.

Studying the communication channel implies running a large number of simulations. In this project, usability refers to facilitate this task to the user. Facilitate the entry of the simulation parameters and automate the execution of multiple simulations where possible.

3.3. FUNCTIONAL REQUERIMENTS.

An overview of the features of the simulator is the following. The simulator models a space (2D or 3D) that contains a collection of particles moving according to the laws of diffusion. The user should be able to place transmitters that vary the concentration of a given particle in its immediate environment and receivers that can measure the concentration at their location.

In the overview above, there are four key elements : simulation space, diffusion, transmitter and receiver. Below are the requirements for each of these elements.

3.3.1. SIMULATION SPACE.

The simulation space was initially defined as 2D to facilitate fast prototyping. The space must contain a set of particles, a set of emitters and a set of receivers. Particles must be modeled as spheres. It must be possible to set an initial concentration of particles. This requirement is discussed in section 3.3.5.

3.3.2. COLLECTIVE DIFFUSION.

The simulator has to model the collective diffusion, initially taking into account two components of collective diffusion: Brownian motion and collisions. Collisions will be modeled as elastic collisions. The design must allow adding other components of collective diffusion in the future, i.e. electrostatic forces.

3.3.3. EMITTER.

The system must allow placing any number of particle emitters at any point of the simulation space. It has to implement a basic transmitter which only emits, or absorbs, particles to the medium, in order to modify the concentration in their environment. There will be a number of predefined emission patterns and the user must be also able to define custom emission patterns.

In addition, emitters must be implemented in a way that allows the future addition of more complex models. The point is to include future models for the emission process described in [2].

3.3.4. RECEIVER.

As for the emitters, a basic model that does not interfere in the diffusion process is required. And in the future it must be possible to extend the application with more complex receivers in order to study the reception process proposed in [2].

The simulator has to allow placing any number of receivers at any point of the simulation space. Receivers should measure the concentration of particles in their environment.

3.3.5. INITIAL BACKGROUND CONCENTRATION.

One of the requirements is that it must be possible to place an initial concentration of particles in the simulation space to study how this affects the diffusion of particles.

This has important implications for the simulator. Simulation space must be limited because otherwise a background concentration would imply an infinite number of particles. But, a priori, it is not known whether it is more realistic to simulate an open or a limited space. If it were necessary to model an open space, mechanisms to simulate an open space in a limited space must be designed and implemented.

4. SPECIFICATION.

4.1. USE CASE MODEL.

The interaction between actors and system is very simple. There is only one actor, the user, and the only functionality offered by the system is running a simulation. The interaction between user and system is also very simple. The user runs the simulation and gets an ok or error output.

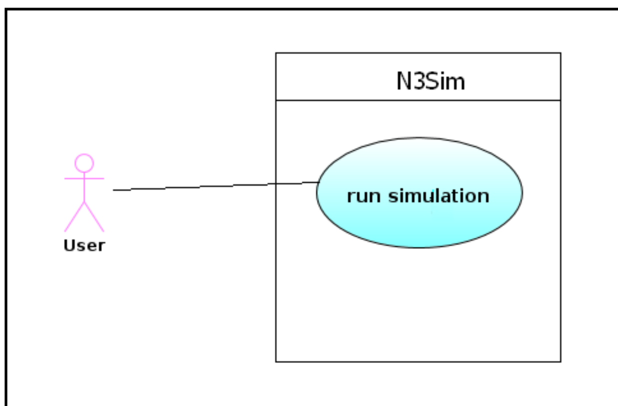


Figure 4.1. Use Case Diagram of N3Sim.

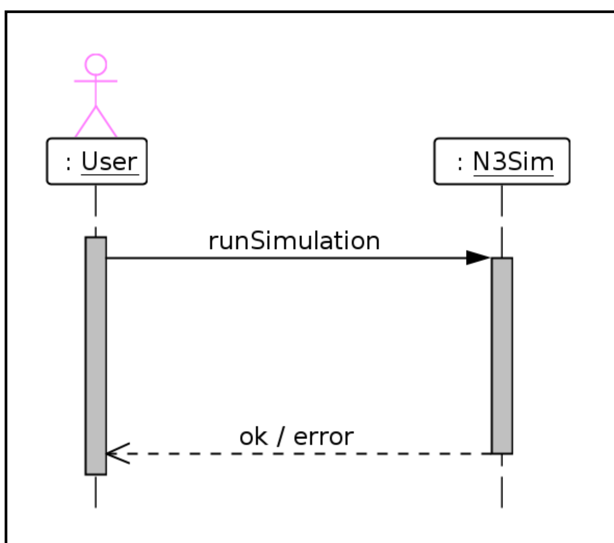


Figure 4.2. sequence diagram for the use case "Run Simulation".

4.2. CONCEPTUAL MODEL.

The conceptual model shown in figure 4.3 is intended to solve the functional requirements outlined in section 3.3.

The class ***Space*** exists to meet the requirements of the simulation space described in section 3.3.1. Class ***Space*** defines a 2D rectangular space defined by the attributes ***xSize*** and ***ySize***. It also contains a set of elements of the class ***Particle***, a set of elements of the class ***Emitter*** and a set of elements of the class ***Receiver***.

To meet the requirement of modeling collective diffusion, described in section 3.3.2, the class ***Space*** provides the methods ***makeBMStep*** and ***solveCollisions***. The method ***makeBMStep*** moves the particles according the brownian motion for a time of value ***timestep***. The method ***solveCollisions*** modifies these movements by calculating and solving the collisions that occur. Particle displacement information is stored in the variables ***nextX*** and ***nextY*** of each particle. Once all the effects of diffusion have been applied, the actual movement of particles is performed by the method ***updatePositions***. This method assigns to the variable ***x*** of each particle the value of the ***nextX*** variable and the value of variable ***nextY*** to the variable ***y***.

The progress of time is controlled by the class *Simulator* that contains the value of a ***timestep***. The class ***Simulator*** has a method ***start*** that initiates and controls the simulation. Time advances in steps of value defined by the variable ***timestep*** and at each timestep the method calls the needed operations to simulate diffusion.

The abstract class ***Boundary*** is used to define the limits of the space. The classes ***VerticalBoundary*** and ***HorizontalBoundary*** extend the class ***Boundary*** and allow implementing a rectangular space. The class ***SphereBoundaryCollision*** is used to model collisions of spheres and boundaries. This is the way a limited space is modeled.

The interface ***Emitter*** serves to meet the requirement of section 3.3.3. This interface contains a single method: ***emit***. This method takes as parameters the current time and the list of particles. An emitter is characterized by an emission function that calculates the number of emitted particles as a function of time.

The interface ***Emitter*** is implemented in several classes that extend an abstract class named ***SphereEmitter***. The class ***SphereEmitter*** models the type of emitter used in this project. It has a fixed position in space defined by the variables ***x*** and ***y***.

SphereEmitter has a radius of influence, defined by the ***emitterRadius*** variable, which defines the circular area in which the emitter may place, or absorb from, particles.

This structure supports an interface which allows the future implementation of any other type of emitter. The only condition for a future emitter is that it implements a method ***emit***. The simulator will work with the new class without further modifications as long as it implements the method ***emit***.

To meet the requirement of section 3.3.4 the interface ***Receiver*** has been created with a single method named ***count***. This method receives the set of particles of the space to measure the concentration in the receiver workspace. Just as the interface ***Emitter,*** this structure allows the future incorporation of any structure as a receiver as long as it implements the method ***count.***

Receivers implemented in this project have a fixed position in space defined by the variables ***x*** and ***y***. Two types of receiver have been implemented. One that measures the concentration in a circular area defined by the variable ***radius***, the ***SphericalReceiver***, and another one that measures the concentration in a rectangular area, the ***CubeReceiver***.

To meet the requirement of section 3.3.5, a limited space has been designed using the classes ***Space*** and ***SphereBoundaryCollision*** as explained earlier in this section. In addition, to simulate an open space in a limited space the method ***updatePositions*** is used. This method decides which particles cross the space as if an open space were implemented. Details of the implementation of this technique are described in section 5.3.4.2.

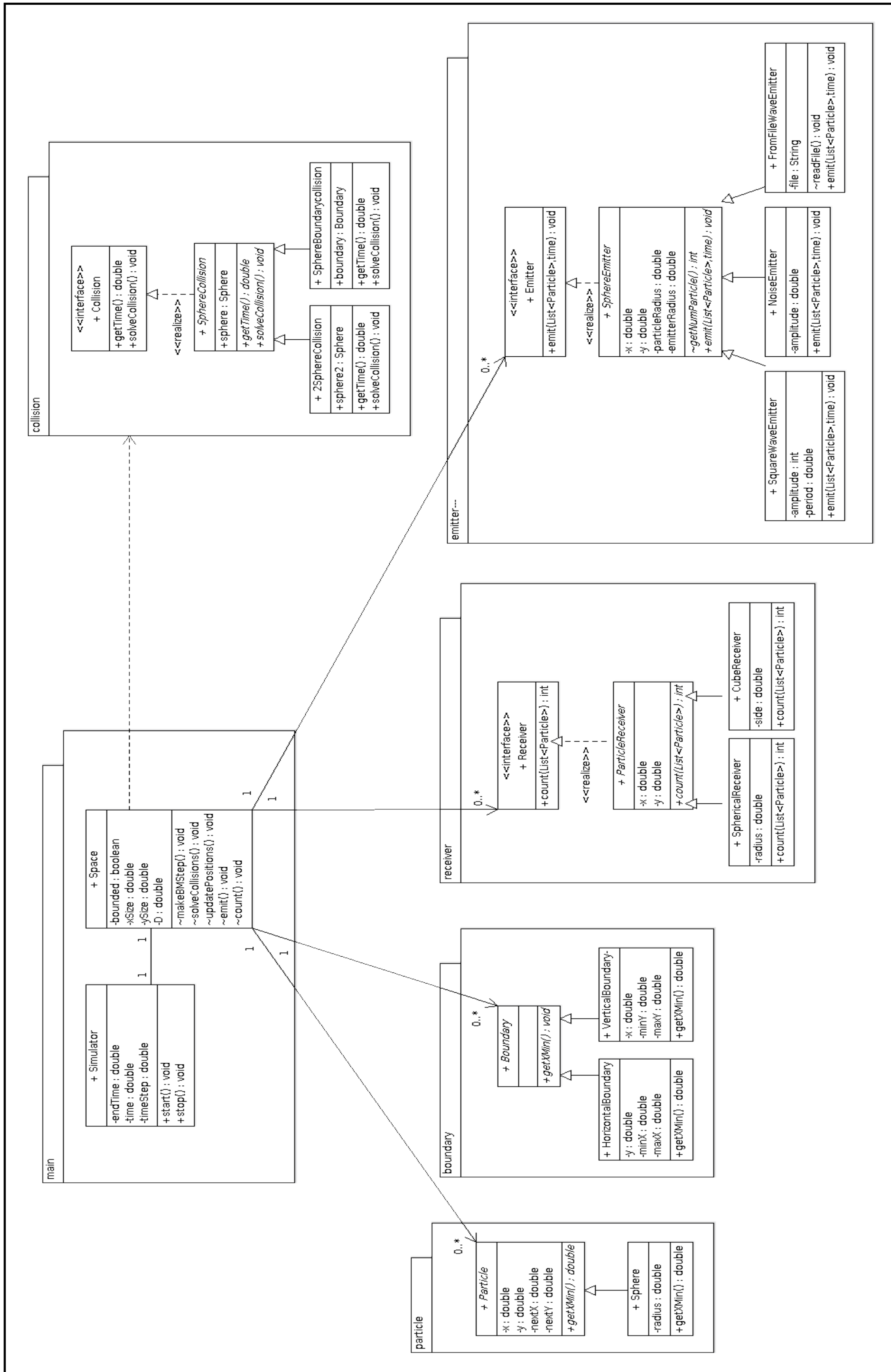


Figure 4.3. Class diagram.

5. DESIGN AND IMPLEMENTATION

5.1 INTRODUCTION

The requirements elicitation strategy explained in section 3.1, the fast prototyping and spiral development, have largely determined the project implementation.

The need to develop working prototypes in short periods of time (fast prototyping) combined with the fact that end users could use different OS (especially Linux and Windows) are the reasons for implementing the program in Java. Java is an OOL appropriate to implement the model proposed in chapter 4. Besides, Java is cross-platform, thereby reducing the prototype development time. Java is the programming language the student has the most experience with, fact that also helps in reducing the developing time.

For the implementation, a three-layer architecture has been chosen because it allows to separate very well the two layers (user interface layer and data layer) that will be very simple (and in the future may be completely different) from the domain layer that will contain the "intelligence" of the system. This is explained in the following section 5.2.

In section 5.3, the overall package structure is described and its implementation is detailed in section 5.4.

Finally in sections 5.5 and 5.6 there is a brief explanation of two prototypes that were developed but were not included in the current version of N3Sim, although they may be integrated into future releases. One is the inclusion of electrostatic forces in the collective diffusion, which is explained in section 5.5. The other is a prototype of a simple video player that reads a file generated by the application and shows the process of diffusion. This prototype is explained in section 5.6.

5.2 THREE LAYER ARCHITECTURE

This project combines two kinds of objectives outlined in section 1.5, a long-term one and a short-term one. In the long term the objective is to develop a program that contains whatever is necessary to model molecular communication based on collective diffusion. In the short term the goal is that the application can be used by a particular group of users to perform simulations to help them validate the theoretical studies in the field of molecular communication based nanonetworks.

But in the long run this application may take several paths that are yet to decide. The data and the user interface layers could be completely modified. N3Sim may become a model of a superior application such as the NS-3. As matter of fact, as explained in chapter 2, it has recently been published a molecular communication simulator which is indeed a module for the NS-2. Another possibility is to develop N3Sim as a stand-alone application with a graphical interface. Or it could be developed as a library that can be used by other developers.

The three-layer architecture is well suited for this project because it allows to separate the two layers that have to be very simple and that in the future may be completely different (user interface layer and data layer) from the domain layer. Currently, the user interface and data layers just read and write text files. The domain layer is the one that contains the *intelligence* of the system, the one that models the diffusion process. Therefore, in our case the three-layer architecture fulfills for the requirements of modifiability and extensibility.

Figure 5.1 shows a diagram of the layers of the project. Both the domain layer and the data layer have a controller class that makes them independent from the other layers.

5.3 PACKAGE STRUCTURE

Figure 5.1 shows the three-layer architecture together with the package structure of the simulator.

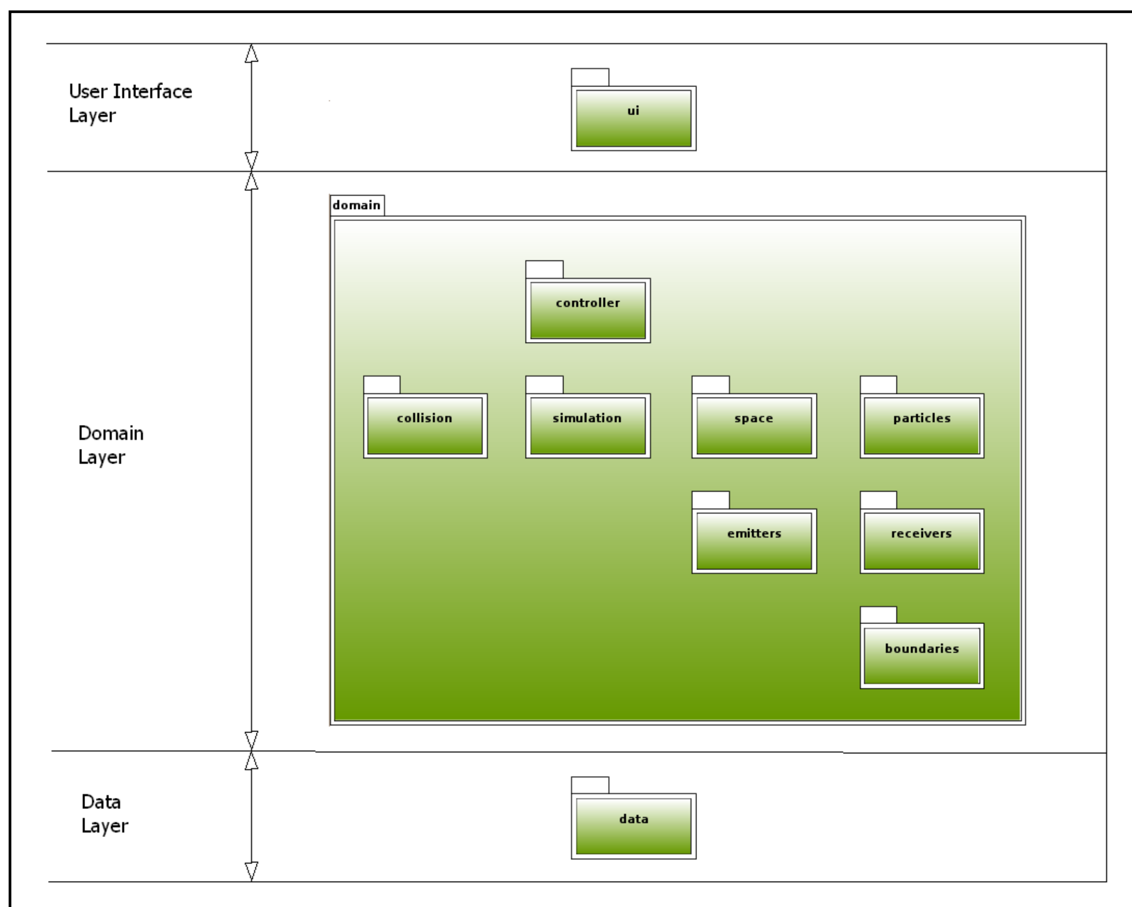


Figure 5.1. Three-layer architecture and package structure of N3Sim.

As discussed in the previous section, the project has focused its efforts on developing the domain layer because it contains the "intelligence" of the system, the molecular communication model. So, both the user interface layer and the data are very simple. These layers interact with the user and the system just by reading (user interface layer) and writing to file (data layer). They are implemented in a single package each.

The domain layer follows the structure described in the conceptual model presented in section 4.2, with the difference that a ***controller*** package is included to separate the domain layer from the user interface layer.

In the domain layer the main package is the package ***space***. It contains two classes: ***Space*** and ***Simulator***. ***Simulator*** has a start and stop method and an object of class space. It is the class that controls the movement of space (it controls the time and the operations to be applied to space at each time.)

The class ***Space*** contains the data structures with the components of the space and provides the necessary operations to modify it according the diffusion principles.

The remaining packages (particle, emitter, receiver, boundaries) are auxiliary packages used by the class space to model its components.

5.4 IMPLEMENTATION.

The previous section has already given an overview of the implementation. This section will explain in more detail the packages and classes of each layer.

5.4.1 USER INTERFACE LAYER.

As explained in section 3.1, the project requires a simple user interface that allows fast prototyping and is oriented to research users. To do this the chosen option has been to use a text configuration file editable by the user. In this file the values of all possible variables of the simulations are defined.

The application expects to receive the name of the configuration file (with full path if the configuration file is not in the same folder as the executable) as the first parameter. For instance:

```
$ java -jar NanoSim.jar myConfigFile.cfg
```

The variables in the configuration file are separated into four groups for clarity: simulation, space, transmitters and receivers variables. Annex IV includes a list of all possible variables of a simulation with a brief explanation.

In addition, to launch multiple simulations in a simple way, there is a keyword, ***param***, that can be used as value for any parameter. In this case, the execution command must include the variable values (in the same order they are placed in the configuration file). In this way the user can automate the launching of simulations combining the configuration file with scripts.

For example, let's imagine we want to run two simulations where the variable ***timestep*** takes values ***1*** and ***2*** and the results of each simulation must be stored in folders named ***timestep-1*** and ***timestep-2*** (results folders are defined by the variable ***outPath***). As the values of all other variables are the same for both simulations, the

same configuration file can be used just setting the values of ***outPath*** and ***timestep*** to ***param*** and running the two simulations like this:

```
$ java -jar N3Sim.jar configFile.cfg timeStep-1 1

$ java -jar N3Sim.jar configFile.cfg timeStep-2 2
```

These two commands can be put together in a script like the following:

```
PARAM1_LIST=(1 2)

for i in ${PARAM1_LIST[@]}; do

        java -jar N3Sim.jar configFile.cfg myTest-${i} $i

done
```

This system can be used in more complex simulation automations. For instance, if in the example above we want to repeat each simulation N times to obtain average values and eliminate noise, this can be done by modifying the script as follows:

```
#/bin/bash

        N=10

        PARAM1_LIST=(1 2 5 10)

        for (( j = 0 ; j < $N; j++ )); do

                for i in ${PARAM1_LIST[@]}; do

                        java -jar N3Sim.jar myConfigFile.cfg myTest-${i}-${j} $i

                done

        done
```

Annex II, shows a few examples of how to use scripts to automate simulations.

The user interface layer is implemented in a single package, ***ui***. Figure 5.2 shows the structure of the package. The class ***UI*** contains the method ***main*** and the class ***fileUI*** is an auxiliary class to read from file. The class ***UI*** communicates with the domain layer through the class ***Controller*** (see Controller package, section 5.4.3.7).

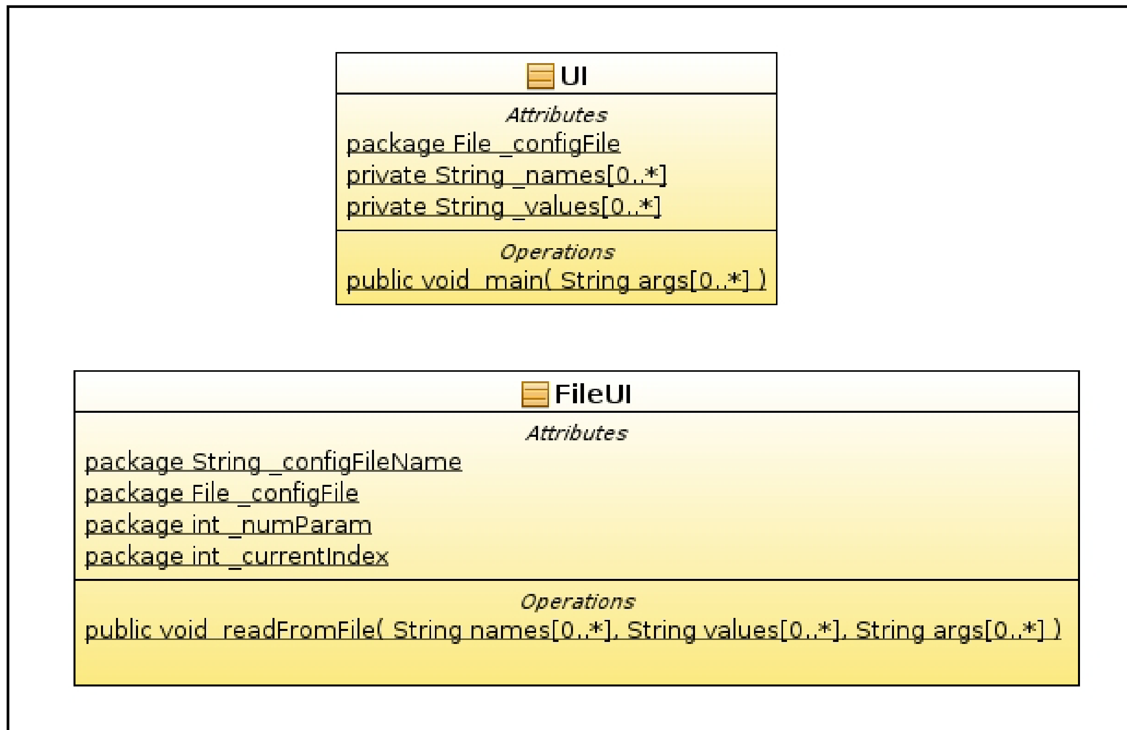


*Figure 5.2. Class diagram of package **ui**.*

Neither the user layer nor the rest of layers of the application control errors of the input (with minor exceptions). This decision has been taken to support the fast prototyping and because the user profile allowed to do so.

5.4.3 DOMAIN LAYER

5.4.3.1 OVERVIEW

The goal of the domain layer is to implement the process of diffusion. The layer has been implemented as a package (package ***domain***) containing seven subpackages:

Package ***controller***: to separate ***ui*** and ***domain*** layers. It offers operations to create the simulation space and run a simulation with appropriate parameters.

Package ***space***: is the heart of the simulator. It contains the class ***Space*** that represents the simulation space with its components (particles, emitters, receivers) and the class ***Simulation*** that *moves* the space, which means that it controls the modifications of the space by the diffusion process over time.

The packages ***receiver***, ***particle***, ***emitter,*** and ***boundary*** are auxiliary packages used by the class ***Space*** for the representation of the space.

The package **collision** is used to model the effect of collisions among suspended particles in the diffusion process.

5.4.3.2 PACKAGE SPACE

Figure 5.3 shows the class diagram of the package space. The aim of this package is to be the main package of the domain layer. It has two classes: ***Space*** and ***Simulator***. ***Simulator*** is responsible for the dynamic part, it controls the time, the operations applied to space and it is also responsible for the communication with the data layer. The class ***Space*** contains the representation of the simulation space: the space with boundaries and the sets of particles, transmitters and receivers.

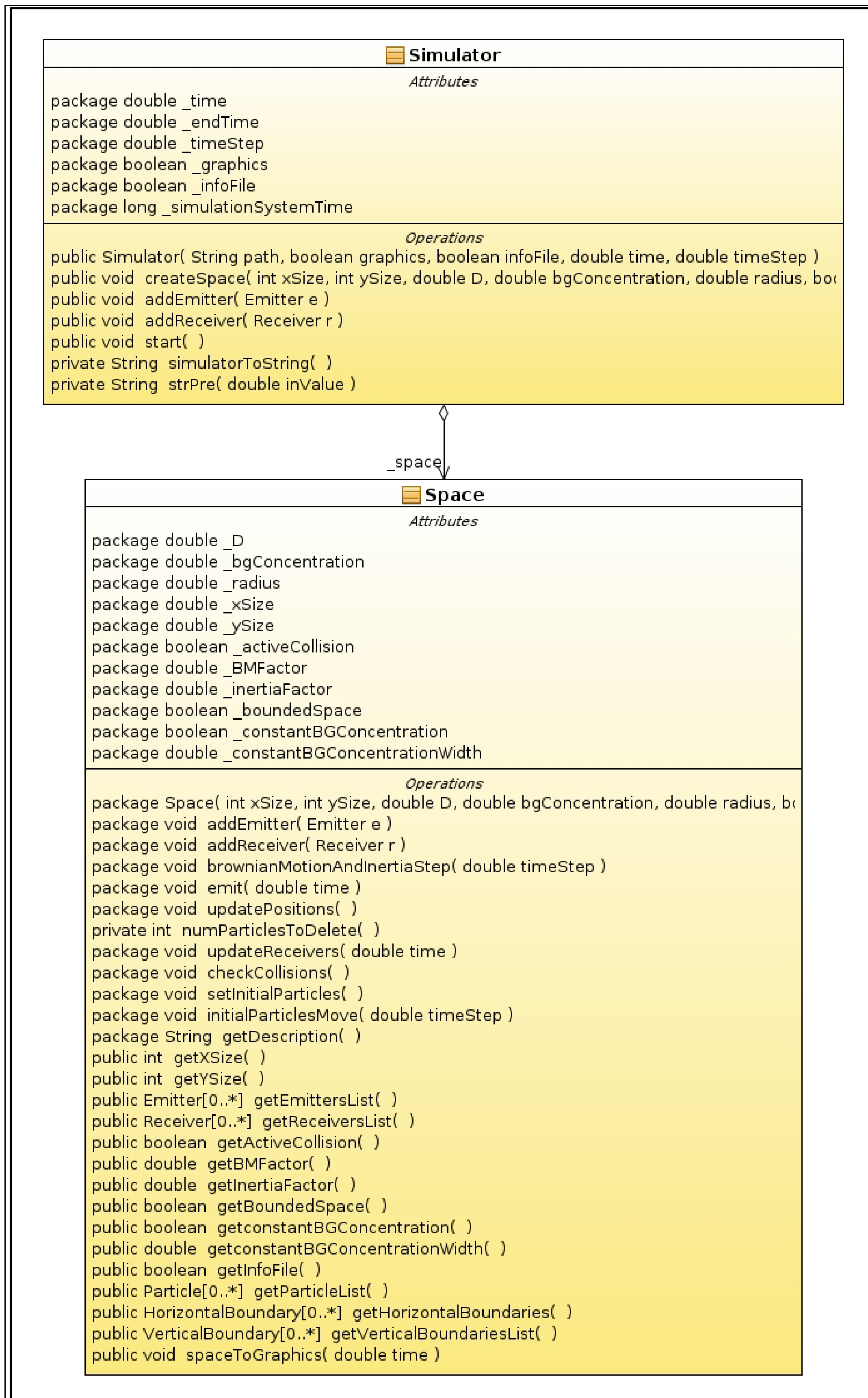


*Figure 5.3. Class diagram of package **space**.*

The class ***Simulator*** controls the time, so it has the attributes ***_time***, which is the current time, ***_endTime***, which is the total time of simulation and ***_timeStep***. The attribute ***_simulationSystemTime*** is the system time used to get information on the cost in time of different parts of the simulation. This information, and other, is stored in the file info.log.

Since the class ***Simulation*** is also responsible for communication with the data layer it has the two boolean attributes ***_infofile*** and ***_graphics***, to decide whether to create video and information files.

The ***Simulator*** class also has an attribute of type ***Space***. This means it contains the simulation space. Simulator applies the operations that the class ***Space*** offers to model the diffusion process and to obtain the necessary information, basically the measures of the receivers.

The ***Simulator*** class has a start and stop method because it is the future candidate to become a thread. This way it will be possible to run different simulations in parallel. This functionality is not in the requirements and has not yet been implemented, but it has been shown to be a future possibility and is a good structure to separate the dynamic and static part. The ***start*** method initializes the space and controls the progress of time. Time progresses by leaps of time, timesteps (value defined by variable ***_timestep***). Each timestep the method performs the appropriate operations to model the diffusion process and get the information needed which is sent to the data layer to be written to file.

The class ***Space*** models the space its elements. So it has a list of particles, ***_particleList***, a list of emitters, ***_emitterList*** and a list of receivers, ***_receiverList***.

The boolean attribute ***_boundedSpace*** controls whether the space will be open or limited. If it is limited then the ***_ySize*** and ***_xSize*** attributes define the limits of space which is a rectangle defined by the opposite corners (0, 0) and (_xSize, _ySize).

The space limits are implemented using the classes of the package ***boundary***. These classes represent boundaries or *walls*. Particles must rebound on these walls. This is done using the package ***collision*** where a new class, ***SphereBoundaryCollision*** implements the interface ***Collision***. This solution allows using the boundaries in the future to model obstacles (may be interesting to know how obstacles influence diffusion) or using to model more complex receivers o emitters. Currently, only horizontal and vertical boundaries have been implemented as they were the only necessary to model a rectangular space. The class ***Space*** has a list of vertical boundaries, ***_verticalBoundariesList*** attribute, and another list of horizontal boundaries, ***_horizontalBoundariesList*** attribute. Although so far there are only two of

each, using lists will allow an easier implementation of obstacles or other objects that may use boundaries. Just by inserting a boundary in the list of boundaries the simulator will calculate and solve any rebound of particles on the boundary since the mechanisms of creating walls and their behavior in the simulation are already implemented

Figure 5.4 shows the signal at the receiver for two scenarios : open and limited space. The conclusion is that for some simulations a limited space is not appropriate. But in order to work with a background concentration (requirement explained in section 3.3.5) an open space would mean an infinite number of particles. For this case it was proposed to simulate the operation of the open model in the closed model. The solution is to pretend that a certain amount of particles are allowed to go through the limits of the space (actually they are deleted from the set of particles).

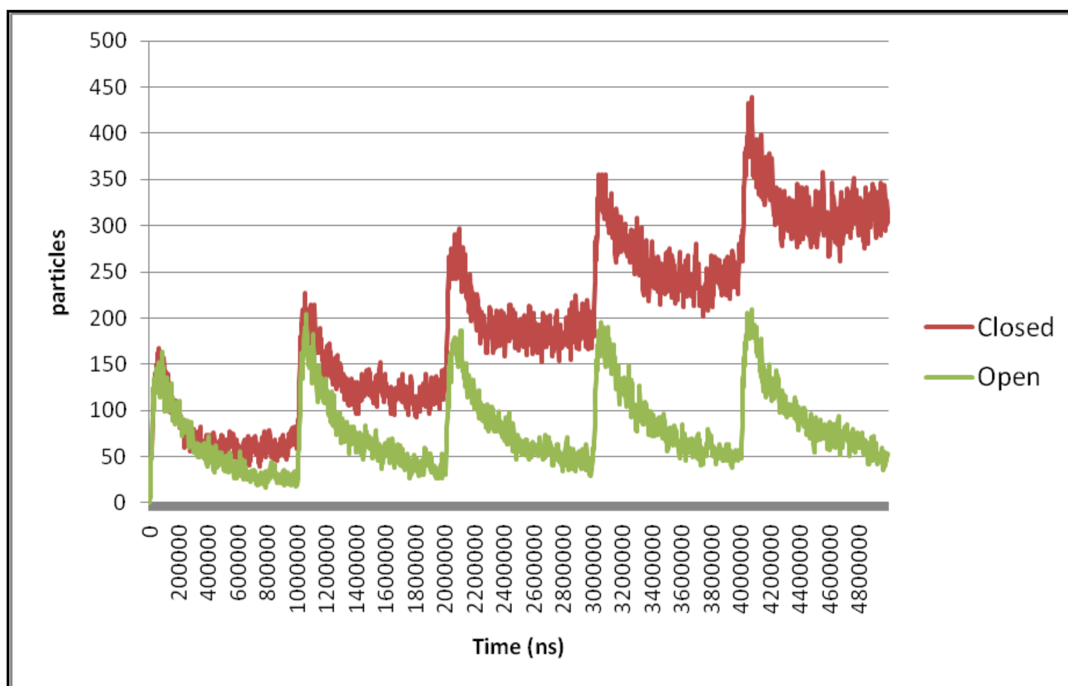


Figure.5.4. Comparison of signal received for open and closed space.

One way to achieve this effect is that the particles that rebound on the limits have a probability to go through the space limits. This way the results shown in figure 5.5 were obtained. The figure shows the signal in the case of open space and in cases of limited space with probability 0.5, 0.6, 0.7, 0.8 and 0.9 of crossing the border. The figure shows that this implementation allows the background concentration to remain stable. The peak signal and the time it arrives are not affected. Yet the tail of the signal is distorted and it is difficult to adjust it using probability.

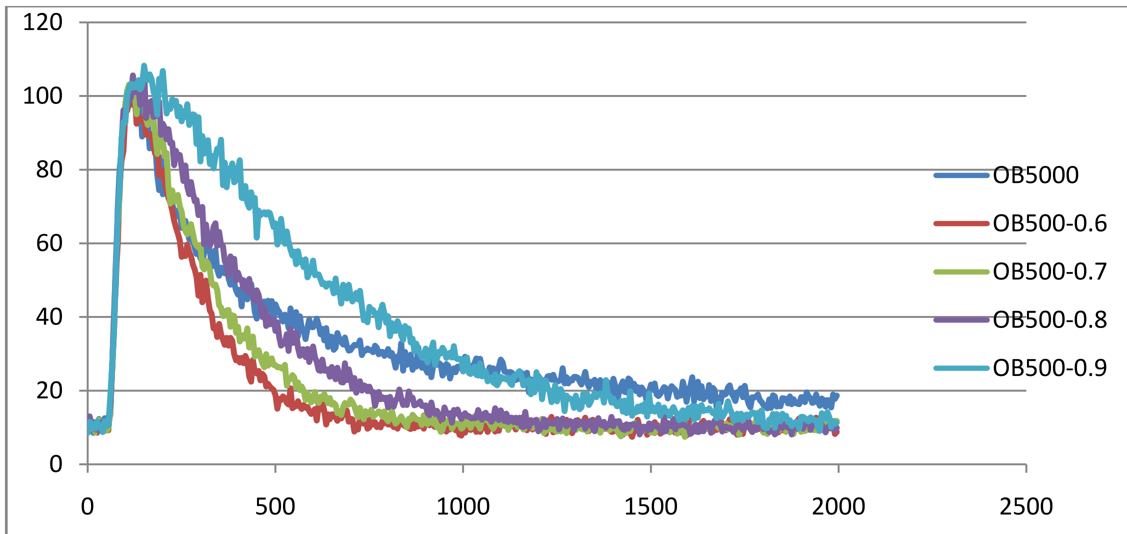


Figure 5.5. Received signal in the case of open space (OB5000) compared with signals received in a limited space which simulates open space by probability of particles to cross space limits.

Another way to simulate the open space in a limited space method is based on measuring the particle concentration in the areas close to the limits of the space. Then it is possible to calculate how many particles should cross the limits. This method has proven to be more accurate. The signal tail is much closer to the one obtained in open space and can still fit more by adjusting the value of the width of the area near the border to study. Figure 5.6 compares the results of received signal for open space and for limited space with different area widths (35, 75 and 100 nm in a limited space of 2000 x 2000 nm). As it can be observed, the differences with respect to the scenario of open space are almost imperceptible.

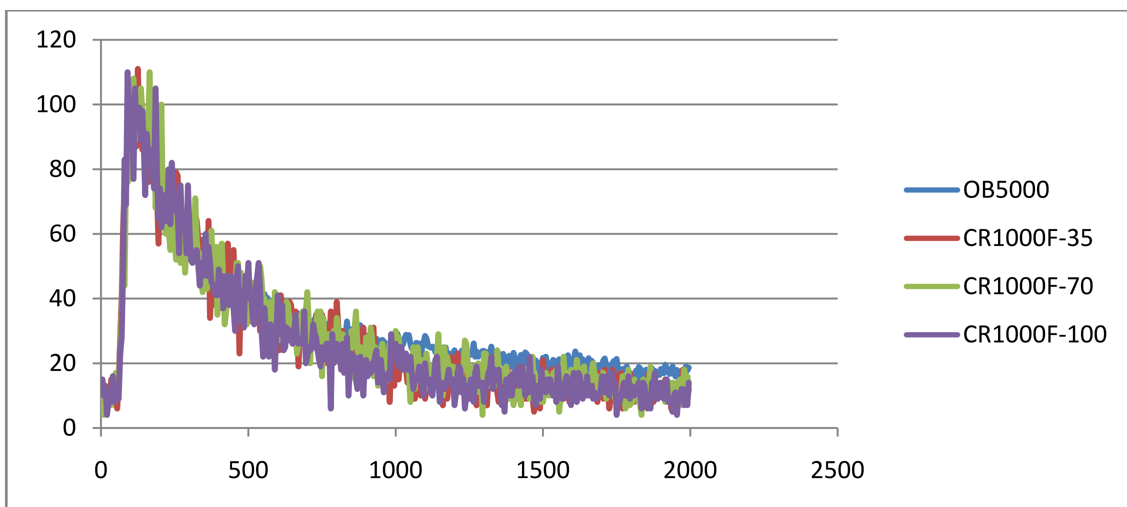


Figure 5.6. Received signal in the case of open space (OB5000) compared to signals received in an enclosed space which simulates open space.

5.4.3.3 PACKAGE PARTICLES

Figure 5.7 shows the class diagram for the package ***particles***. The package contains an abstract class, ***Particle***, from which other classes inherit. This class contains the basic methods and variables that the simulator needs to apply the diffusion process.

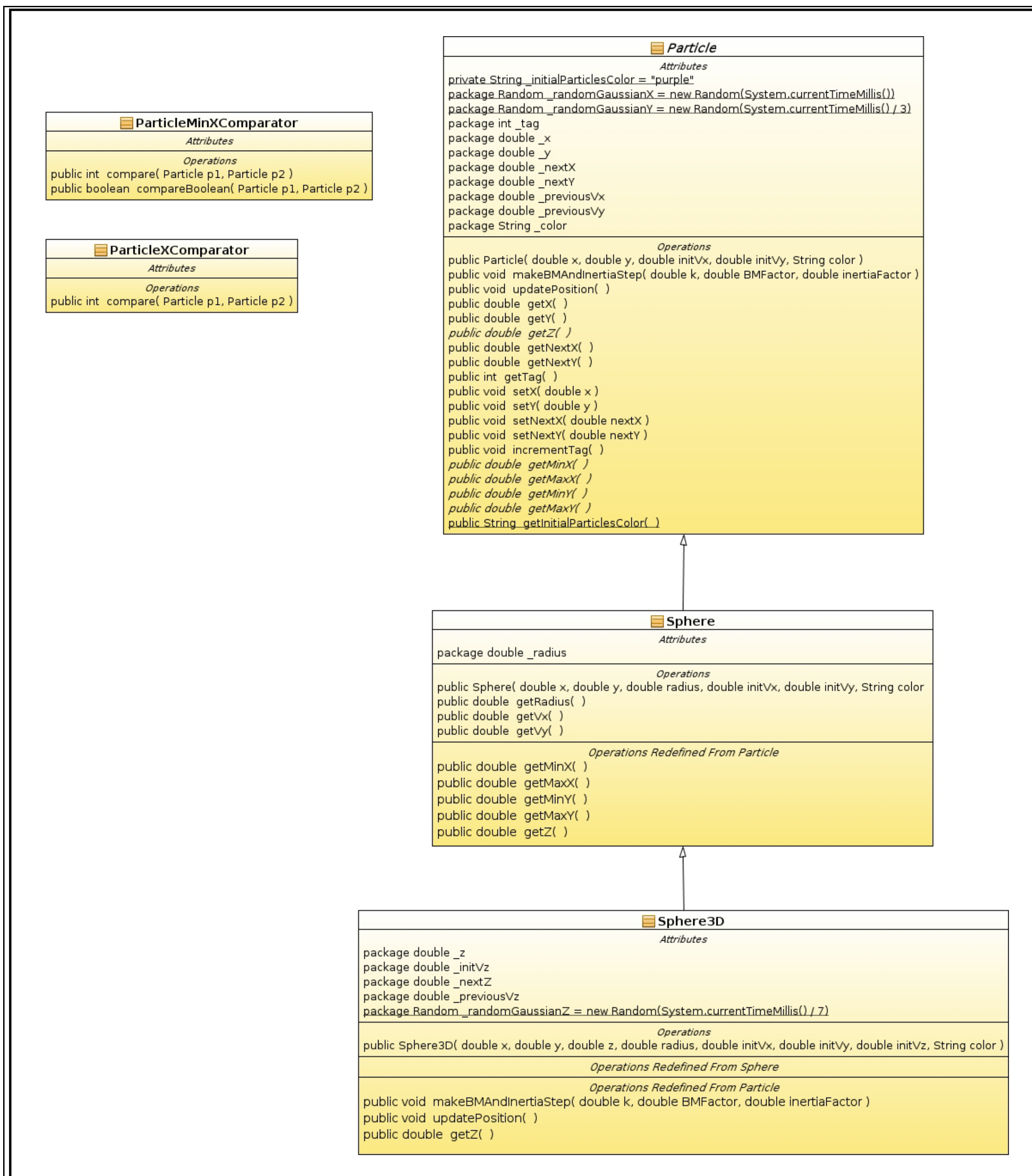


*Figure 5.7. Class diagram of package **particle.***

This class contains the attributes ***_x***, ***_y***, ***_nextX*** and ***_nextY*** and the method ***updatePosition*** that are used by all components of the diffusion process. The

attributes ***_x*** and ***_y*** represent the position at the beginning of the timestep and ***_nextY*** and ***_nextX*** represent the position at the end of the timestep. The idea is that, starting from the initial position, the various components of diffusion modify the final position according their own laws. Each timestep, once all components have performed diffusion, the simulator must call the method ***updatePosition*** so that the particles actually *move*, which means that particles change their position, ***_x*** takes the value of ***_nextX*** and ***_y*** takes the value of ***_nextY***.

In addition, in order to implement the inertia component of diffusion it is necessary to know the speed at the previous timestep. This value is stored in variables and ***_previousVy*** and ***_previousVx***. At the end of each timestep the method ***updatePositions*** updates these values.

Since it was decided to model the particles as spheres, the first class that inherits from the class ***Particle*** is the class ***Sphere***, which represents a sphere but in two dimensions. The only required attribute for this class is the radio, ***_radius***, used to calculate the collisions among particles.

Finally, for three dimensional scenarios the class ***Sphere*** is extended to the class ***Sphere3D***. For this case it is necessary to include the variables of the three dimensions and redefine the method ***updatePositions***.

In addition, there are auxiliary methods (***getMinX***, ***getMinY*** ...) and classes (***ParticleXComparator***, ***ParticleMinXComparator***) used to sort the list of particles and to estimate whether the paths of two particles cross.

5.4.3.4 PACKAGE EMITTERS.

The package ***emitters*** contains classes to model emitters that meet the functional requirements outlined in section 3.2.

Figure 5.8 shows the class diagram of the package ***emitters***. The package has an interface that has a single method: ***emit***. Using this interface allows that any class that implements this interface can be used by the simulator as an emitter. This is very useful for the future development of the simulator because it may be necessary to model more complex emitters as shown in the paper [2].

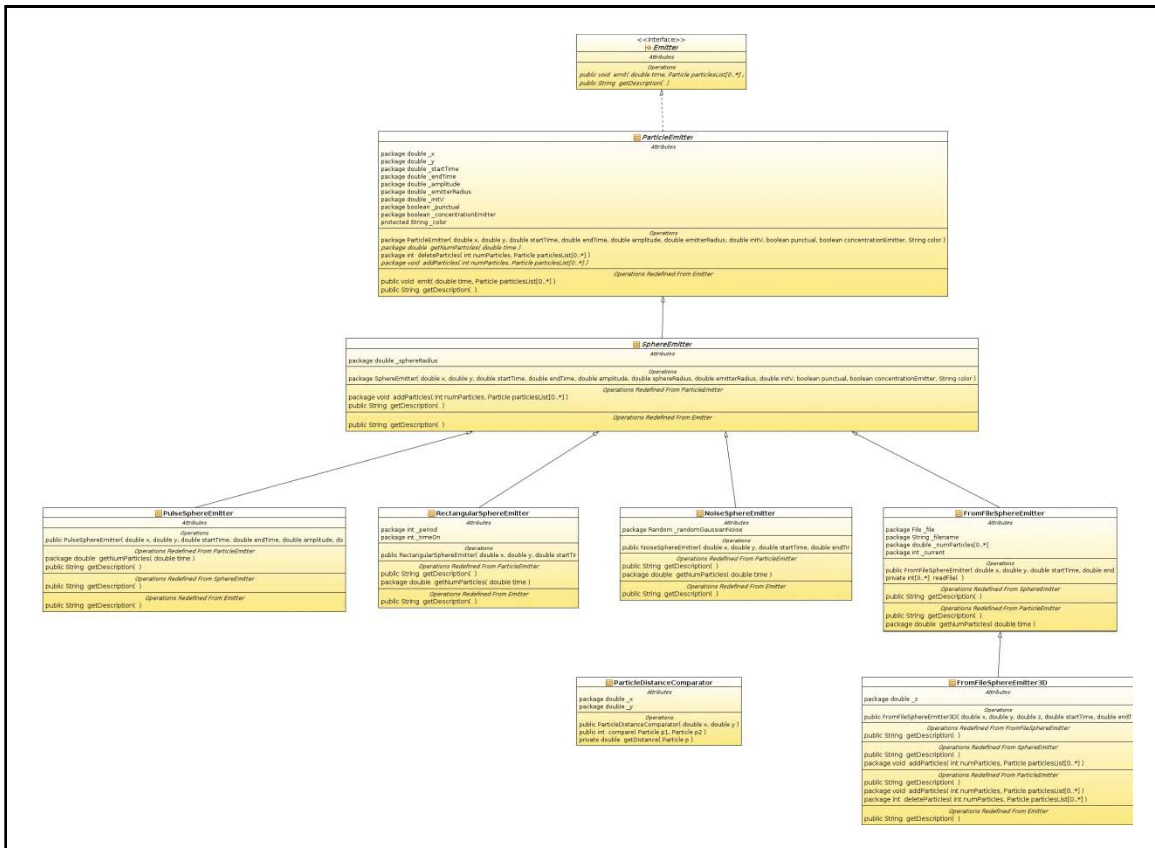


*Figure 5.8. Class diagram of package **emitter.***

The method ***emit*** takes two parameters, time and the list of particles in space. The emitter *knows* for a given time how many particles it must emit or absorb. The method does not return any value, but its results are reflected in the variations that occur in the list of particles.

The abstract classes ***ParticleEmitter*** and ***SphereEmitter*** implement the interface ***emitters*** in a general way, defining attributes and methods needed but do not specify the waveform of the emitter.

Emitters have a fixed location defined by the attributes ***_x*** and ***_y***. Emitters have too a radius of influence defined by the attribute ***_emitterRadius***. Emitters release or absorb particles only between times defined by the attributes ***_startTime*** and ***_endTime***.

The class ***SphereEmitter*** implements the method ***emit*** but using private methods (visible only in the package), one to absorb particles and another to release particles. This way, in order to create an emitter with a particular waveform it is only necessary to extend this class and to implement the method ***getNumParticles***. This method should return the number of particles to be released or absorbed or a given time, this is, the waveform. Following this explanation, four emitters with different

waveforms have been implemented. Three with a predefined waveform (***PulseEmitter, RectangularEmitter and NoiseEmitter***) and another that reads the waveform from a text file created by the user.

5.4.3.5 PACKAGE RECEIVERS

Figure 5.9 shows the class diagram of the package ***receivers***. As in the package ***emitters***, the package ***receivers*** has an interface that has a single method: ***count***. Using this interface allows that any class that implements this interface can be used by the simulator as a receiver. This is very useful for the future development of the simulator because it may be necessary to model more complex receivers as shown in the paper [2].

The interface emitter is implemented through the abstract class ***Receiver***. The attribute ***_name*** is used to identify the receiver. For each receiver the simulator creates a text file with the receiver's name and the extension *.csv*. The attributes ***_x*** and ***_y*** define the location of the receiver. The boolean attribute ***absorb*** determines whether the receiver absorbs the particles found in its area of influence or it just behaves as a transparent receiver.

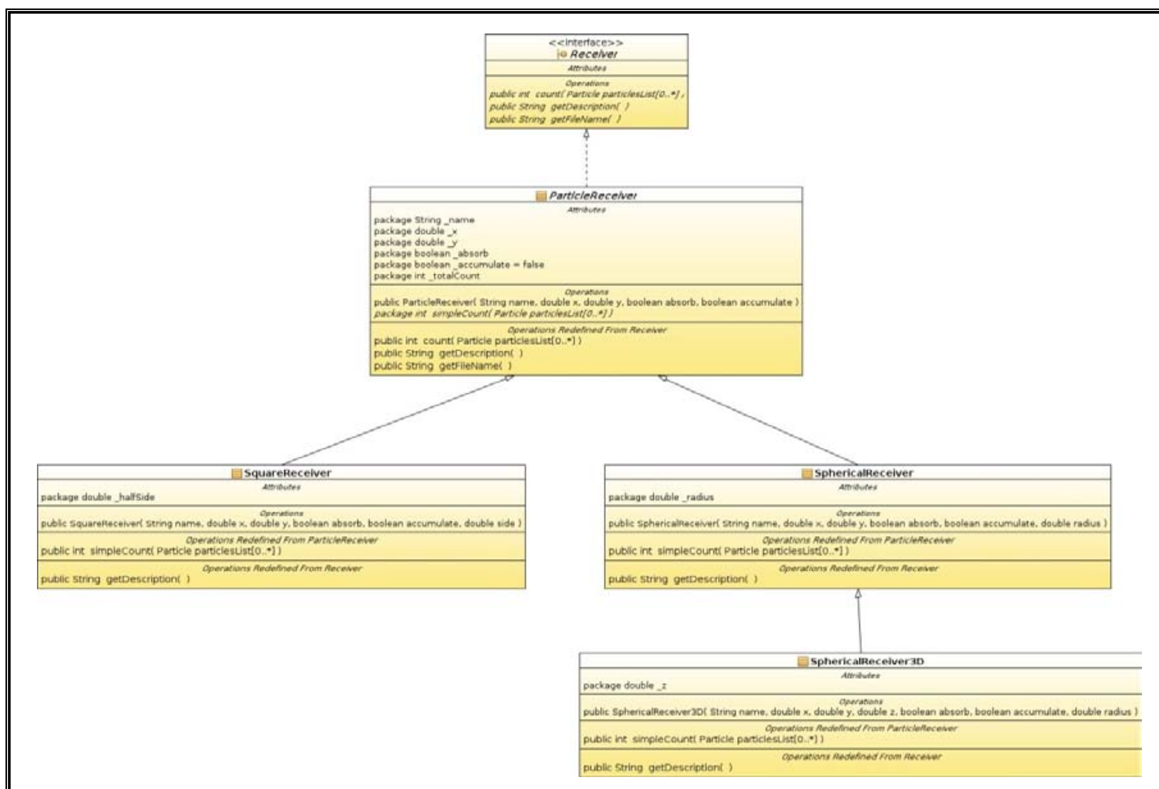


*Figure 5.9. Class diagram of package **receiver.***

The method ***count*** calculates the concentration in the receiver's area of influence by counting the number of particles that are located into its area. The method receives the list of particles as a parameter.

The two classes that directly extend the abstract class ***Receiver***, ***SphericalReceiver*** and ***SquareReceiver*** differ in the area of influence. The class ***SphericalReceiver*** has a circular area defined by the attribute ***_radius***, and the class ***SquareReceiver*** has a rectangular area defined by the attribute ***_halfSide***.

For three dimensional scenarios the class ***SphericalReceiver*** has been extended to the class ***SphericalReceiver3D***. In this case, it is necessary to include an attribute for the z location and redefine the method count.

5.4.3.6 PACKAGE BOUNDARIES

The package ***boundaries*** aims to model walls in the simulation space. These walls are used as space limits and may be used in the future to model obstacles. Another future use of boundaries may be as building blocks of more complex objects, especially considering the modeling of more sophisticated transmitters and receivers as described in [2].

The main function of these walls is that the particles should rebound on them. There is no method in the classes of the package to implement this functionality. Rebounds are modeled in the package ***collisions*** by the class ***SphereBoundaryCollision***.

Figure 5.10 shows the class diagram of this package. There is an abstract class with two methods, ***getYMin*** and ***getXMin***, to get the minimum y and x dimension. These methods are useful when sorting items to find collisions.

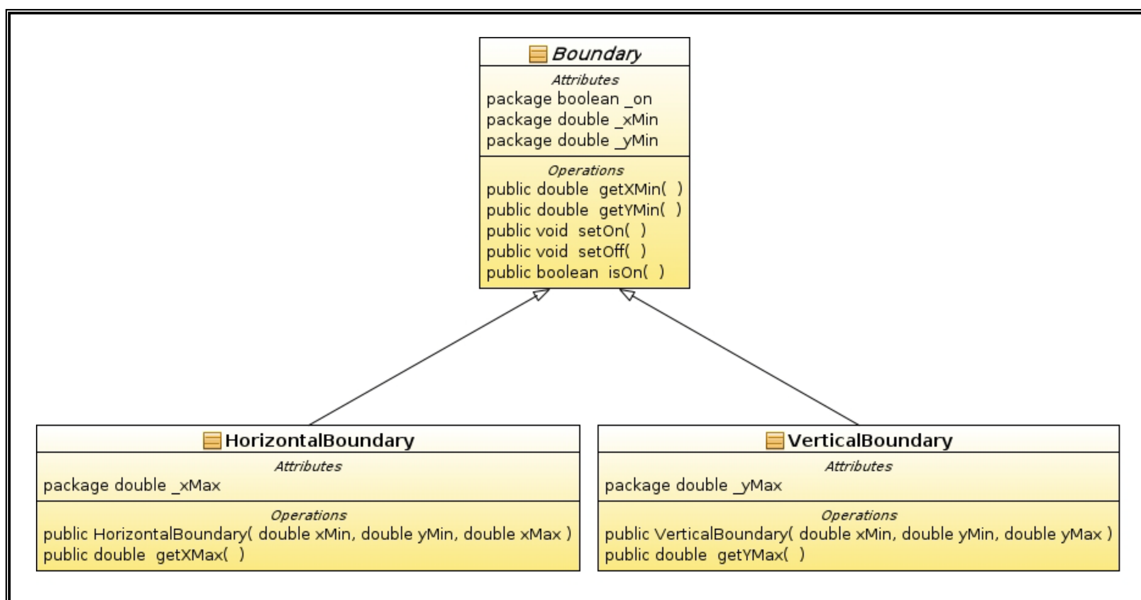


*Figure 5.10. Class diagram of package **boundaries**.*

5.4.3.7 PACKAGE CONTROLLER

The aim of this package is to separate the user interface and domain layers. The package has a single class, which this class provides the user interface layer methods to create a simulation, to add receivers and transmitters and to start the simulation.

5.4.3.8 PACKAGE COLLISIONS

Figure 5.12 shows the class diagram of the package ***collisions***. The package collisions has two goals. The first one is to model the collisions among particles and among particles and walls (the limits of the system) offering the necessary methods to calculate and solve them. The second goal is to compute and solve the set of collisions that will occur in the simulation space during a timestep.

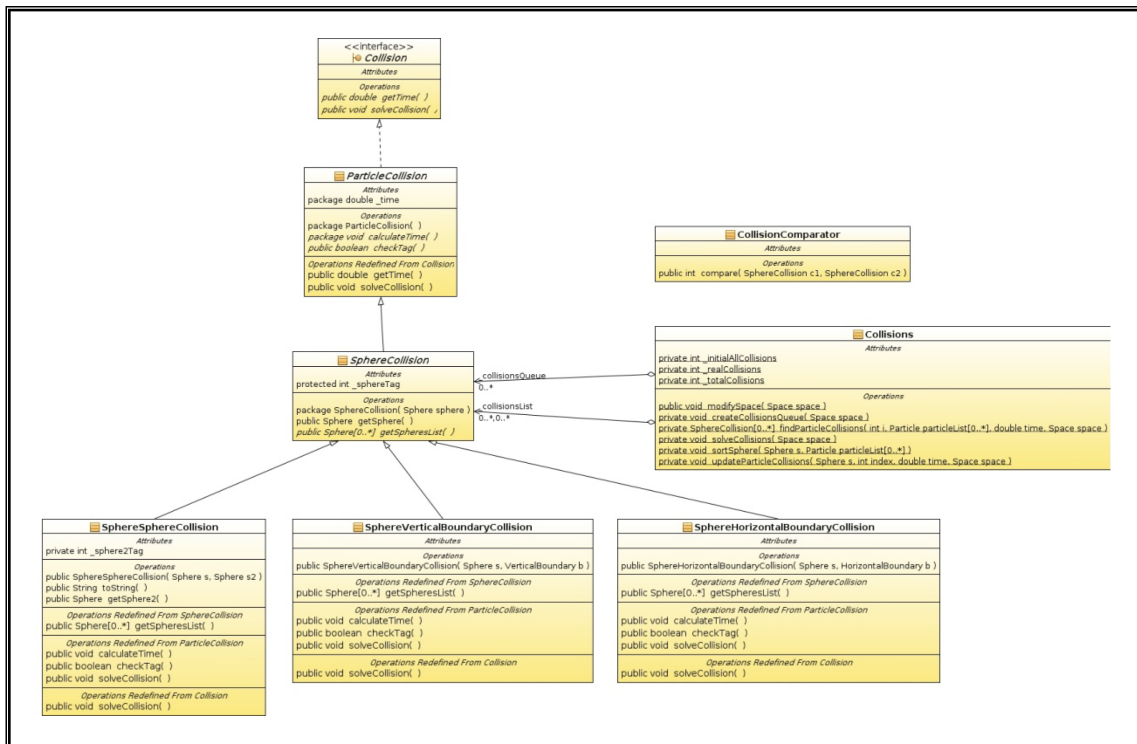


Figure 5.12. Class diagram of package **collisions**.

The static class ***Collisions*** is in charge of the second goal. To do so it has the method ***solveCollisions*** that receives as parameter the simulation space. This way the class ***Simulation*** (let's remember that it is the main class that controls the simulation process and contains an object of type Space) can use this method directly to solve the collisions.

This second goal is the most complex one from the point of view of algorithmics. Because of its extension and complexity, it is explained in a separate chapter, Chapter 6.

Regarding the first goal, the representation of collisions, it is done with the rest of the classes of the package. Like in other packages, an interface has been used that has the operations that the collisions solving algorithm needs. This way, in the future new types of collisions (i.e. collisions among particles and spheres) may be added just by implementing the interface.

The two operations of the interface are ***getTime***, which calculates the time at which the collision occurs, and ***solveCollision*** that modifies the path and speed of the particles involved in the collision. The first one is necessary because the collision algorithm must find all the possible collisions and resolve in the first place the one with lower collision time.

5.4.4 DATA LAYER.

The purpose of this layer is to write to file the simulation results and the simulation information. For this a single package with two classes has been implemented. Figure 5.13 shows the class diagram of the package ***data***.

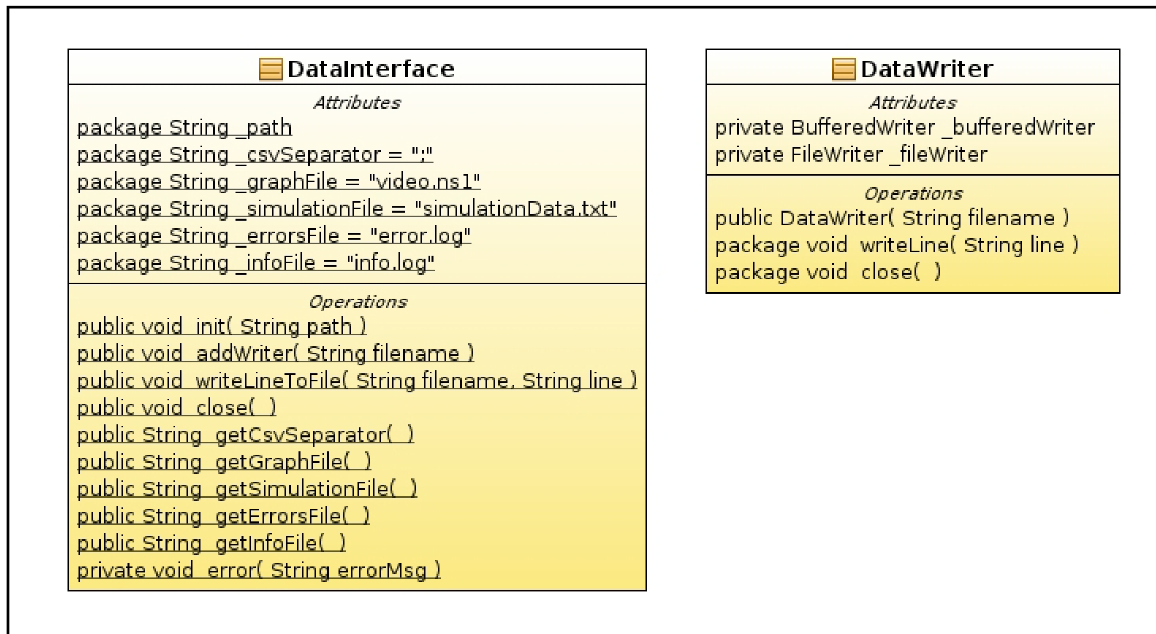


*Figure 5.13. Class diagram of package **data.***

The class ***DataInterface*** is the connection with the domain layer and uses the class ***DataWritter*** to write to file. For each file that should be handled the class ***DataInterface*** creates an instance of the class ***DataWriter*** associated with that file. By default error and simulation information files are created. The class ***DataInterface*** offers the method ***addWriter*** to create any other file needed by the simulator.

The class ***Datawriter***, besides the constructor method that opens the file and another method that closes it, provides the method ***writeLine*** to write a String to the file. The class ***DataInterface*** has a map that associates each file name to the instance of the class that corresponds to that file. ***DataInterface*** acts as an intermediary between the domain layer and the ***Datawriter*** instances.

5.5 ELECTROSTATIC FORCES

Although it was not within the initial requirements of the project, modeling the electrostatic forces among particles is the next step in the modeling of the collective diffusion. At the end of the project, the effect of such forces has been included in the simulator. However the results have not yet been validated by the group N3Cat, so it is considered a beta version.

This chapter briefly describes the theoretical bases and implementation of this module.

If the suspended particles have electric charges, they are subject to the laws of electrostatics. The electrical forces among them must be considered because these forces will cause particles to move. On the other hand, a particle moving within a fluid receives a resistance which is a function of particle's speed and other variables.

As an effect of electrostatic forces each particle is subject to the sum (vector) of the forces exerted on it by the rest of particles. According to Coulomb's law, these forces are worth:

$$F = K\frac{q^2}{r^2}$$

Where ***K*** is Coulomb's constant ($9{\cdot}10^9$ Nm2/C2), ***q*** is the particle's electric charge and ***r*** is the distance between particles.

On the other hand, if a force acts on a particle within a fluid, this particle will reach a maximum speed. In our case, as the Reynolds number is much less than 1, the Stokes formula can be applied:

$$F = 6\pi\eta r v$$

Where ***r*** and ***v*** are the radius and particle velocity respectively and **η** is the dynamic viscosity of the fluid. Solving for ***v*** we obtain:

$$v = \frac{F}{6\pi\eta r}$$

This way we know the speed that the electrostatic forces produce on particles. Multiplying by the time, the particle displacement is obtained, which is added to the displacement caused by Brownian motion.

Section 7.5 shows results obtained with this prototype.

5.6 N3SIM VIDEO PLAYER (N3SVIDEO).

N3SVideo is a small prototype built with C + +, OpenGL and Qt to help visualize the simulations made with N3Sim.

During the development of the project it was thought interesting to have an intuitive visual tool to help analyze the results of simulations.

The way it works is quite simple: if the graphics variable is set to true, N3Sim writes in a text file a representation of the simulation. This representation consists primarily of the boundaries and then, for each timestep, a list of the position of all particles. N3SVideo simply reads the file and, for each timestep, draws all the particles.

N3SVideo was helpful to better understand the simulations and to debug the application. However, it is in a very early stage and so far its development is not anticipated. Figure 5.14 shows some screenshots of the application.

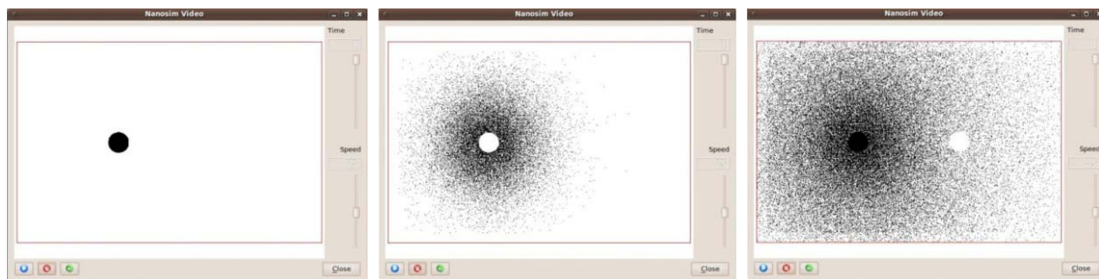


Figure 5.14. Screenshots of N3Sim Video that show three phases of a diffusion process.

6. COLLISION DETECTION ALGORITHM

6.1. INTRODUCTION

The problem of detecting collisions between n bodies with random movement is known and studied in algorithmics. It is well known because the cost of these algorithms is very high and it is a bottleneck of the applications that have to implement it.

Besides dealing with the interaction of n bodies, in itself a problem of cost $O(n^2)$, the problem of collisions has an added complication. Collisions must be solved in order of time, and each collision changes the trajectory of the particles involved. This means that there are new potential collisions.

Finding the next collision is, in principle, an algorithm with cost $O(n^2)$ because each particle must be checked for collisions with all the others. This, together with the high number of collisions that occur, becomes a bottleneck in the application.

On the other hand, we must bear in mind that the process is basically sequential. It is not possible to parallelize any significant part of the algorithm because until a collision is solved it is not possible to find the next one.

Yet, there are algorithms that try to reduce costs by leveraging the temporal and spatial coherence of collisions, as explained in section 6.2. Section 6.3 explains the implementation of the collision detection algorithm done in N3Sim. Finally in section 6.4, a study of the costs of the implemented algorithm is shown.

6.2. STATE OF THE ART

The CD (Collision Detection) problem is well known and studied in computer science because it is used in many graphical applications and simulators.

Collision detection algorithms can be divided into two groups: ***a priori*** and ***a posteriori***.

A priori means that from the current positions and velocities of particles the next collisions are calculated and solved in time order. So, the collision is found before it occurs.

The ***a posteriori*** method divides the trajectories into time frames. When moving to a new frame the application checks if any of the bodies overlap, overlapping indicates a collision.

In the ***a posteriori*** method, the time between frames must be small enough so that is unlikely to lose a collision, as shown in figure 6.1.

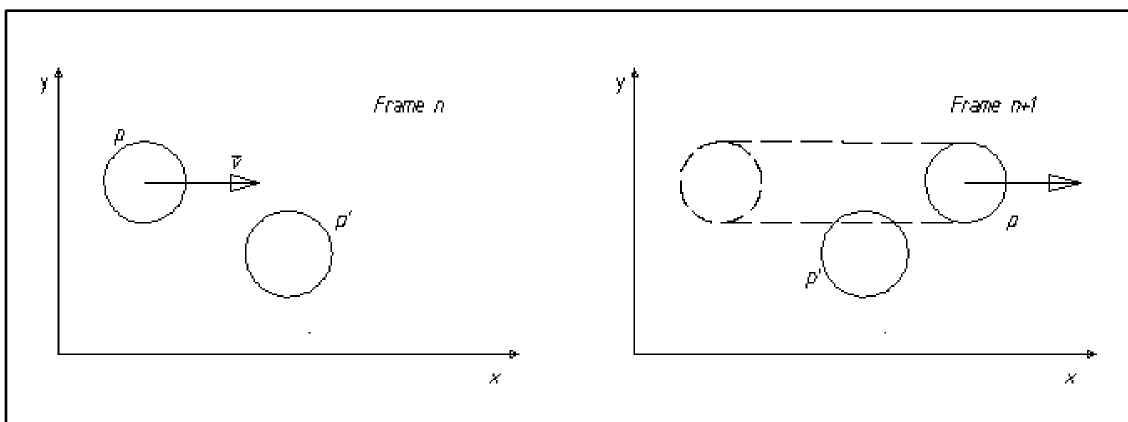


*Figure 6.1. If the time between frames in a **a posteriori** algorithm is too large some collisions may be missed. In the figure, the particles should collide but the algorithm does not detect it.*

The first non-functional requirement of this project is reliability. It is not easy to implement an ***a posteriori*** algorithm that guarantees in any condition that no collision is missing. So N3Sim will implement an ***a priori*** algorithm. Since it is a research project, the results could be in doubt if some collisions were missed.

The post method is however very useful in the case of bodies of complicated geometry and relatively slow moving. Calculating collision trajectories may be very expensive while finding the overlap of two bodies is a lot easier.

A well known example of an a posteriori algorithm is the ***Baraff's algorithm***, also known as ***sweep and prune***. An example of implementation can be found in [7]. This implementation is interesting because it shows the advantages between a *naive* algorithm and an algorithm that exploits the spatial and temporal coherence of collisions. Figure 6.2 compares the costs of collision of each of the two methods.

A naive solution to the collision detection problem is just to iterate through all pairs of objects, testing possible collisions, choosing the first one, solving it, move all

particles to the time of this collision and repeat the process until end time is reached. The time complexity of the algorithm is ***nc·O(n2)*** in all cases, where ***nc*** is the number of collisions and ***n*** is the number of objects.

Spatial coherence is the property that objects are more likely to collide with other objects in its neighborhood. This means that at any given time, the number of colliding object pairs is much smaller than the total number of pairs.

Temporal coherence means that the scene changes relatively little over small time intervals. This means that algorithms can typically reuse calculations from previous frames in order to avoid unnecessary recomputation.

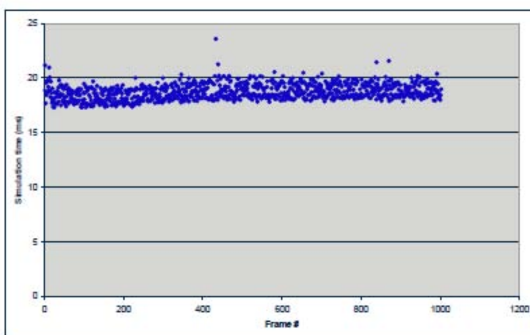


Figure 4.16: Trace of simulation time across a simulation run of the "spheres and pyramid" scene, using the naïve algorithm.

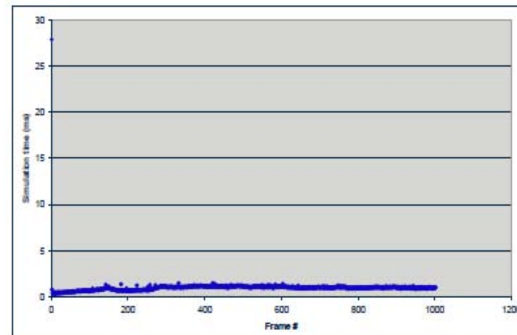


Figure 4.17: Trace of simulation time across a simulation run of the "spheres and pyramid" scene, using Baraff's algorithm.

Figure 6.2. Comparison of results using a naive algorithm and Baraff's algorithm. Extracted from the thesis of Sam Stokes [7]

Baraff's algorithm first takes advantage of spatial coherence. The goal is, for each frame, to detect all the collisions and then solve them. The algorithm sorts the bodies by the lower x coordinate of the body, as shown in Figure 6.3. Then, performing an appropriate iteration over the bodies list (must check each body with all the others), most of the pairs of objects can be easily dismissed. As explained in figure 6.3, for two bodies to overlap (which means they have collided) it is necessary (but not sufficient) that their projections on each axis overlap. When looking for collisions, each object is compared only with the followings in the sorted list. It is easy to see that at the time that the current object we are comparing does not overlap with another object, it is sure that it will not overlap with all the following objects of the sorted list. In this way we can rule out a priori a large number of pairs of objects.

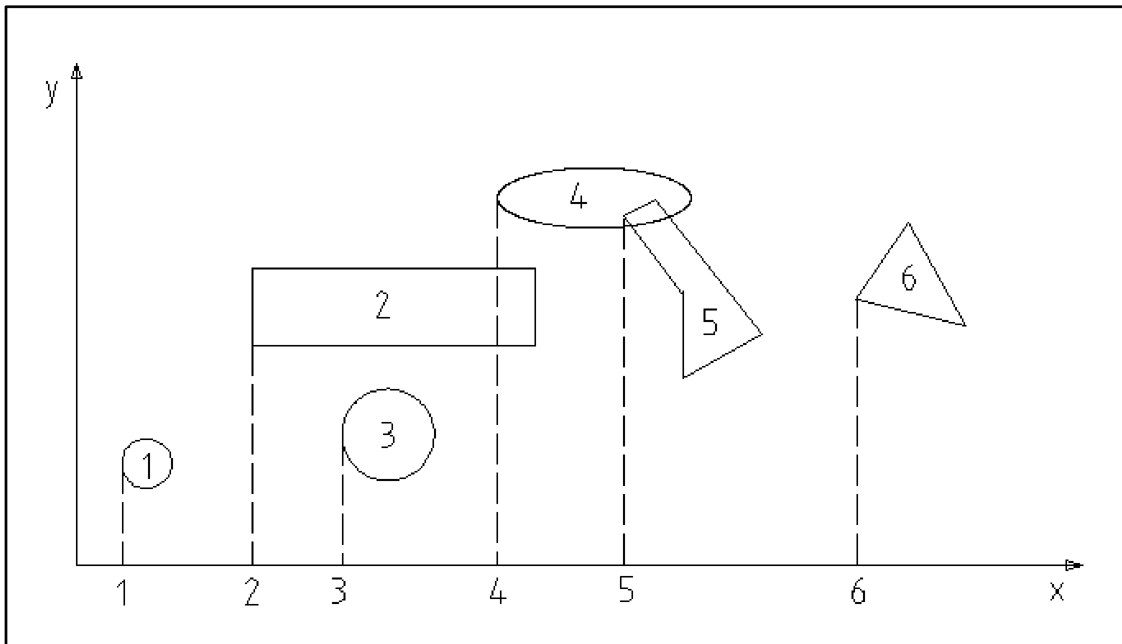


Figure 6.3. The position of a body in the sorted list is given by its minimum x dimension. As can be seen in the figure, for two colliding bodies is necessary but not sufficient, that their projections on each axis overlap.

Baraff's algorithm takes advantage of the temporal coherence in the following way. After solving the collisions of a frame, the bodies list must be sorted again because objects have changed their positions. Since the changes are small and the list is almost sorted, in this case is more efficient to use a sorting algorithm like insertion sort.

Other algorithms use a different approach. The algorithms in [4] and [5] divide the space into subspaces and work on each of them separately (or in relation to its neighbors). This way, the algorithms take advantage of spatial coherence. However, these algorithms are quite complicated because they must take into account new events arising when a body enters in another subspace. On the other hand, it seems that in these algorithms it is important to choose the appropriate cell size, and it does not seem to be an obvious problem at all. Given the need for fast prototyping, such algorithms have been ruled out.

6.3. IMPLEMENTATION

It has been decided to implement a variation of Baraff algorithm. This variation allows using it as an ***a priori*** algorithm. This variation is needed because, as discussed in the previous section, the ***a posteriori*** methods may miss collisions. On the other hand, Baraff's algorithm is a well known algorithm, efficient, proven and not too complex to implement.

Algorithms that divide space in subspaces have been discarded for N3Sim,at least for the first implementation, because they are much more complex. For instance, an important aspect in the effectiveness of these algorithms is the choice of the size of the subspaces. A formula to calculate the subspace side is given in [5] but there is no explanation on how it is obtained. Adjusting and finding the appropriate value for each simulation can be an additional problem that we have preferred to avoid. Keep in mind that the goal in implementing the algorithm was to have a prototype as soon as possible, although the efficiency was not the higher.

The modification of the Baraff's algorithm to use it as an ***a priori*** algorithm has two parts. In the first place, instead of searching the overlap between two objects, we will look for the overlap between their trajectories. Second, to use the temporal coherence is-necessary to save all possible collisions found into a data structure in which the collisions are sorted by time.

Figure 6.4 shows that for two particles to collide it is necessary (but not sufficient) that the projections of their trajectories overlap on each axis. It is just the same reasoning of the Baraff's algorithm (see figure 6.3), just changing bodies by the trajectories of the bodies.

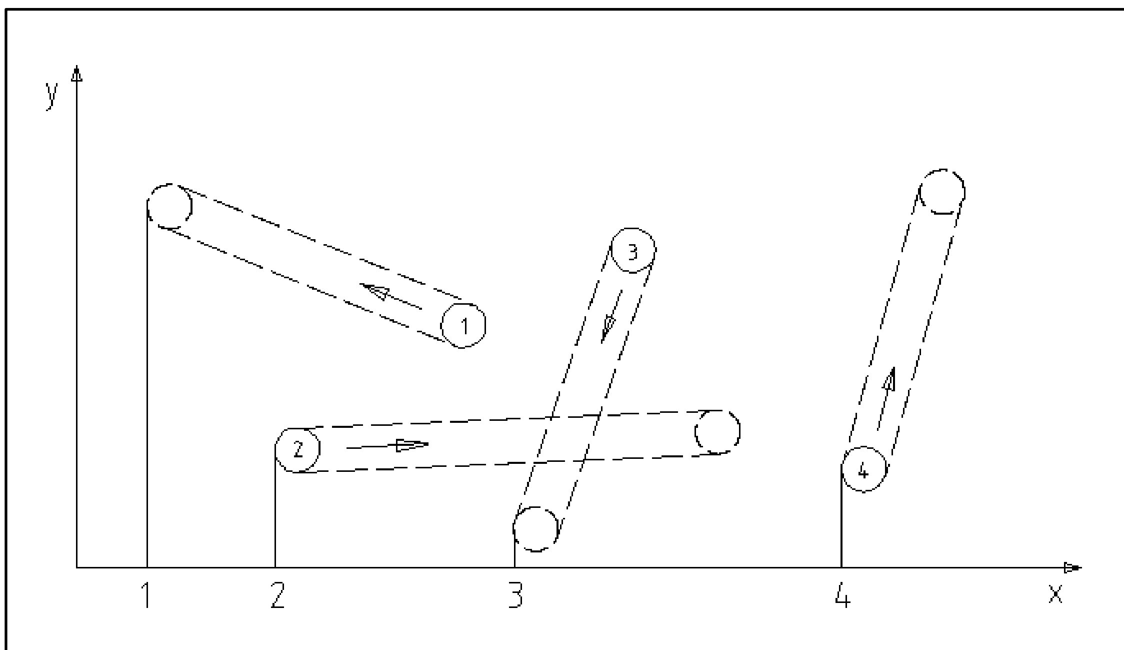


Figure 6.4. Spheres sorted by the minimum x coordinate of their trajectories. Notice that initial position of sphere 1 is greater than the initial position of sphere 2. But the minimum x dimension of sphere 1 is less than that of the sphere 2. It can be seen that for two particles to collide it is necessary that their projections on each axis intersect.

Following this analogy, the algorithm is implemented as objects but considering the trajectories of the spheres. The first step is to sort the particle list by the minimum x dimension of particle trajectories. As in Baraff's algorithm, each sphere is tested against the next ones in the list, discarding all of the following when the trajectories of two spheres and do not overlap.

Collisions are stored in a data structure type Min Heap, which we call ***collisions queue***, where the order criterion is the time of the collision. This way, the cost to obtain the next collision is always O(1).

Once the ***collisions queue*** has been created the algorithm consists in obtaining the next collision and solving it until the queue is empty.

Once a collision is solved, a new problem must be faced. The problem is that solving a collision means that the trajectories of the spheres involved in the collision change and this has two effects. First, the list of spheres is not sorted, and secondly the collisions that had been found that involved the spheres of the recently solved collision are no longer valid and these spheres may have new collisions too.

To solve the first problem it is evident that the list of spheres must be sorted again. As in Baraff's algorithm, an insertion sort algorithm is used because the list is

almost sorted (just need to order two spheres and they are almost at the right position in the list).

Regarding the second problem, in the first place all collisions involving the spheres of the solved collision must be deleted from the collisions queue. Search through the collisions queue to delete such collisions would have a high time cost. Instead of doing such search, a tag system is used. Each sphere has a tag, an integer that is incremented each time the sphere is in a collision that is solved. On the other hand each collision has two tags, one per sphere. These tags take the value of the tag of the sphere when the collision is created. Then, when a collision is retrieved from the collision queue to be solved, if the collision tags and the sphere's tag do not match it means that these spheres have been in a previous collision and then the current collision must be discarded and the next one must be solved. This way, the cost of discarding collisions is O(1).

Finding the new collisions for the spheres involved in a collision means to compare the sphere with the rest of the spheres in the sorted list (as it is done in the first part of the algorithm) and to save the new collisions into the collisions queue. The cost of this step is O(nlogn), O(n) to compare the sphere with the spheres in the list and O(logn) to save each collision in the collisions queue.

Finally the data structures that store collisions have been modified to improve the memory costs. The problem that leads to these changes was that in simulations of large timesteps and high number of collisions there were too many *false* collisions (collisions with invalid tags as explained in the previous section). These produced that the collisions queue grew too much.

To avoid this problem, a list of collisions is associated with each sphere. This list is deleted and recalculated when a particle has a collision. This way, the memory cost is kept constant.

6.4. COST ANALYSIS

The collision algorithm explained in the previous section can be outlined as shown in the table of costs in figure 6.6.

Stage	Operation	Time Cost
1	**Pre-processing**	
1a	Sort Sphere List	O(n)
1b	Create collisions queue	$O(n^2 logn)$
2	**Processing**	
2	While collisions queue is not empty	nc iterations
2a	Obtain first collision	O(1)
2b	Solve collision	O(1)
2c	Sort Sphere List	O(n)
2d	Delete invalid collisions	O(1)
2e	Find new collisions	O(nlogn)

Figure 6.6. Scheme of the collision detection algorithm and time costs associated with each stage (n = number of spheres; nc = number of collisions).

From the figure 6.6 it follows that the cost of a simulation timestep is:

$$C = O(n^2 \log n) + n_c O(n \log n)$$

As in the original Baraff's algorithm, there is a pre-processing cost. It is very high but then the cost of a single collision is reduced from O(n2) (for a naive algorithm) to O(nlogn).

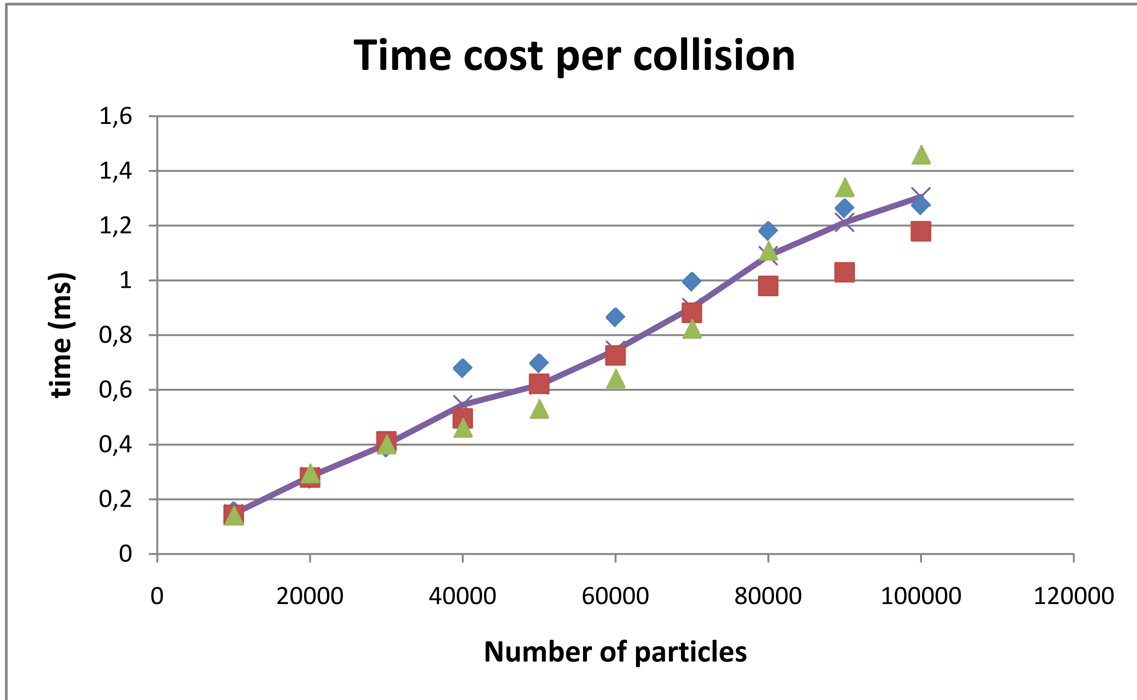


Figure 6.7. Time cost graph of N3Sim (ms per collision).

The number of collisions, nc, is aprox:

$$n_c \cong O(n \cdot c \cdot t)$$

Where ***c*** is the concentration of particles and ***t*** is the simulation time. If we consider a fixed space, the particle concentration is proportional to the number of particles, then

$$n_c \cong O(n \cdot c \cdot t) = O\left(n \cdot \frac{n}{area} \cdot t\right) = O(n^2 t)$$

and

$$C = O(n^2 \log n) + n_c O(n \log n) = O(n^2 \log n) + t \cdot O(n^3 \log n)$$

The memory cost depends on the number of collisions. As earlier outlined in this chapter, the number of collisions depends linearly on the number of particles, concentration and time. For N3Sim time is the time of a timestep, ***ts***, because is the time the algorithm will store the collisions.

$$C_{mem} = O(n \cdot c \cdot ts) = O(n^2 \cdot ts)$$

7. RESULTS.

7.1. BROWNIAN MOTION

The first feature that was implemented was the simulation of Brownian motion as a major component of diffusion.

Figure 7.1 shows a comparison between results obtained with the simulator and expected results according to the theory of Brownian motion. Specifically, compared with formula No. 17 [8]. Both graphs match except for the fact that the signal obtained with the simulator has noise.

This is because the formula used is based on Fick's laws that define the diffusion as a function of the concentration gradient. This means that it does not account for the individual movements of each particle, and because of that noise is not observed. N3Sim has been used to study the channel noise, as shown in section 7.3 and [9].

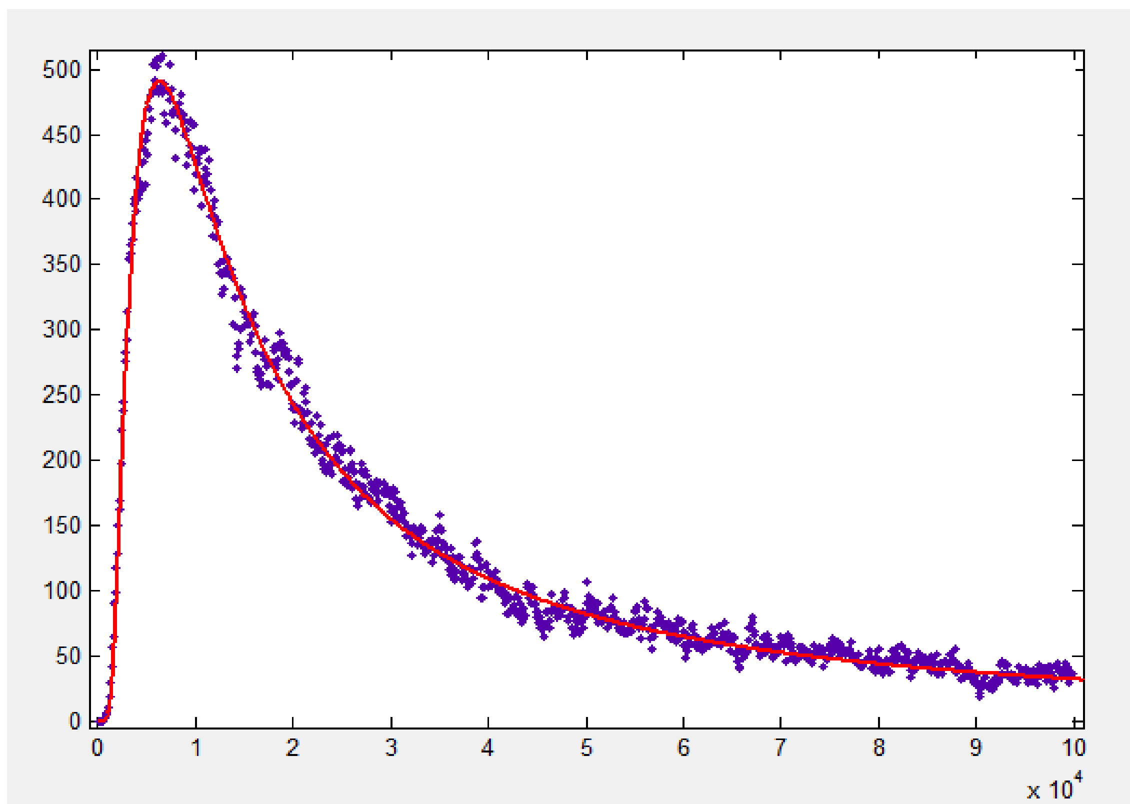


Figure 7.1. Comparison between the signal obtained with the simulator (blue) with the expected signal (red) according to Fick's laws of diffusion.

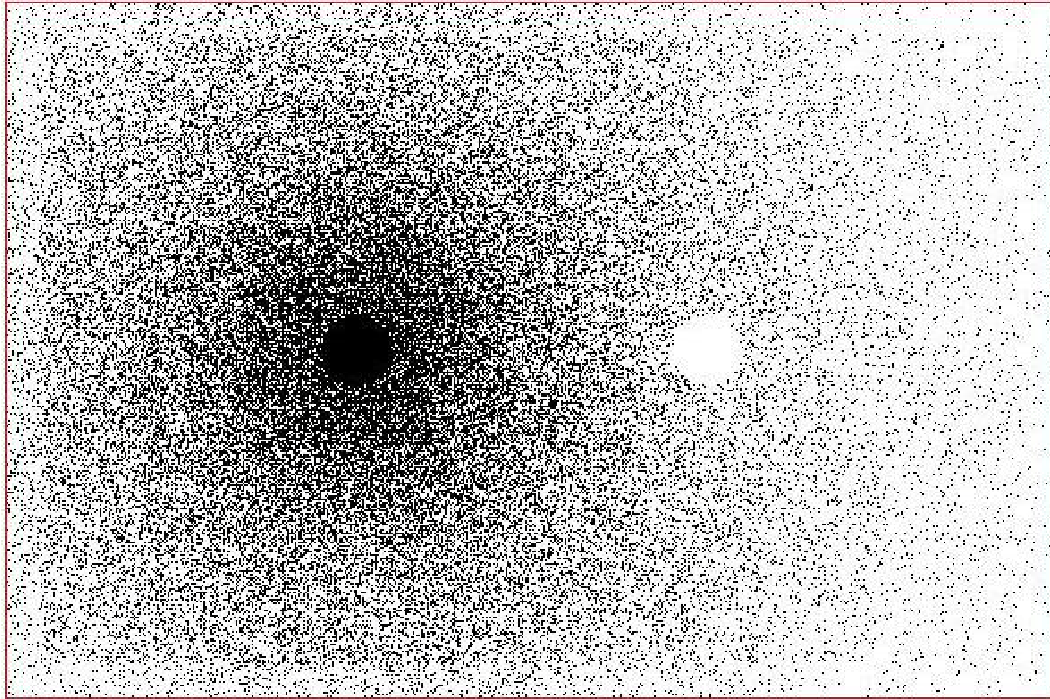


Figure 7.2. Diffusion pattern of Brownian motion.

7.2. SPACE LIMITS

One of the first problems to be solved was how to implement the limits of the system. Figure 5.1 of section 5.4.3.2 shows that an open space is more appropriate.

But in order to work with a background concentration (functional requirement specified in section 3.3.5) the space should be limited because open space with background concentration would mean an infinite number of particles. In this case it was proposed to simulate the operation of the open space in a limited model. This was achieved by allowing an amount of particles to go through the space limits.

Section 5.4.3.2 explains how this simulation was performed. Figure 7.3 shows a comparison between the received signal in an open space (in blue) and a limited space that simulates the open space (in red).

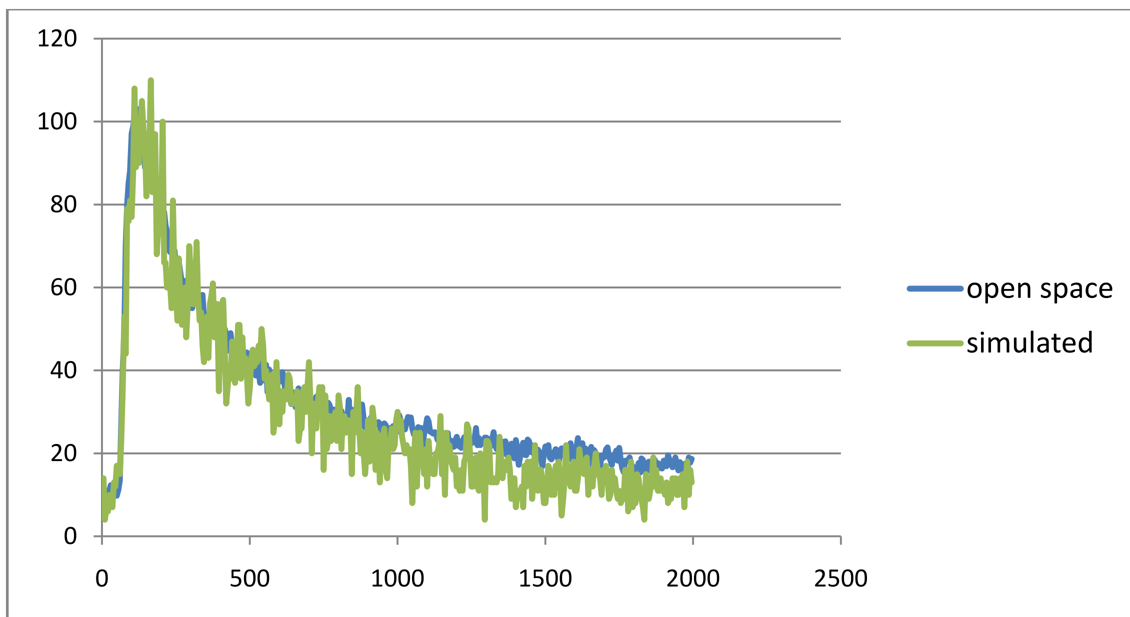


Figure 7.3. Comparison between the signal received at an open space (in blue) and a limited space that simulates the open space (in red).

7.3. NOISE

Noise is an important aspect to consider in a communication channel. In the case of molecular communication there is noise just for the fact of measuring concentrations in a medium that is not completely homogeneous. Figure 7.5 shows the signal received by a receiver in an environment where there is a constant total concentration. The noise is not observed by any external disturbance but for the same communication channel characteristics. Although the total concentration is constant, the local concentration depends on the random movement due to the brownian motion and therefore it is not constant.

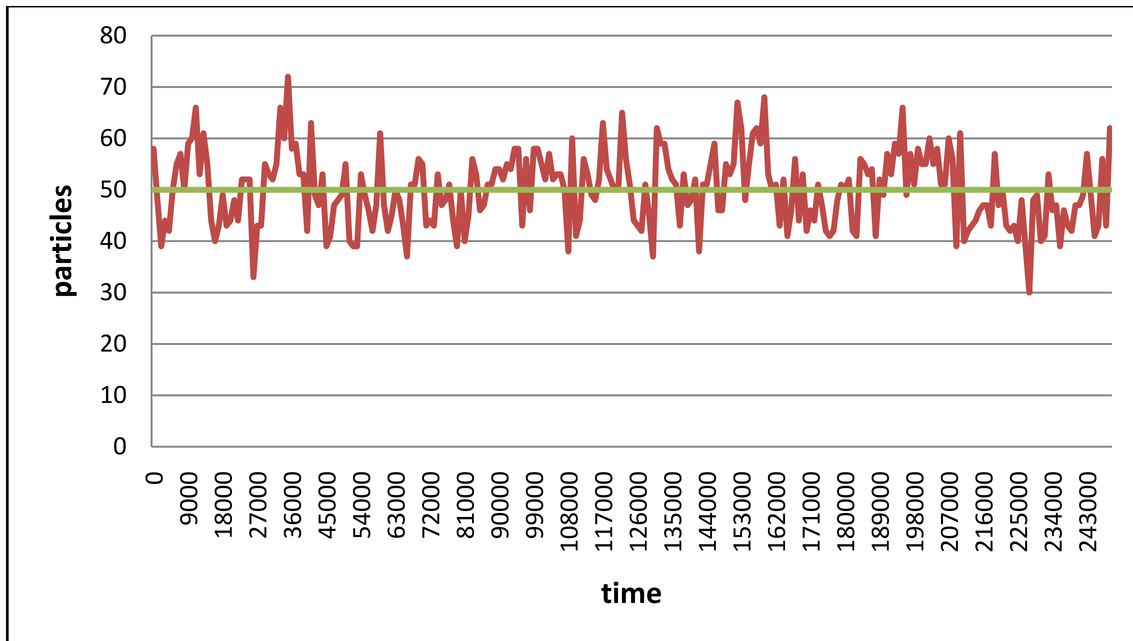


Figure 7.4. Noise of the background concentration.

Nora Garralda of N3Cat studied in [9] the noise in this channel of communication, using N3Sim.

7.4. COLLISIONS.

One of the conclusions obtained with N3Sim is that collisions only affect the values of diffusion when the particle concentration is very high.

Figure 7.5 compares the signals received in simulations with and without collisions with receivers at different distances.

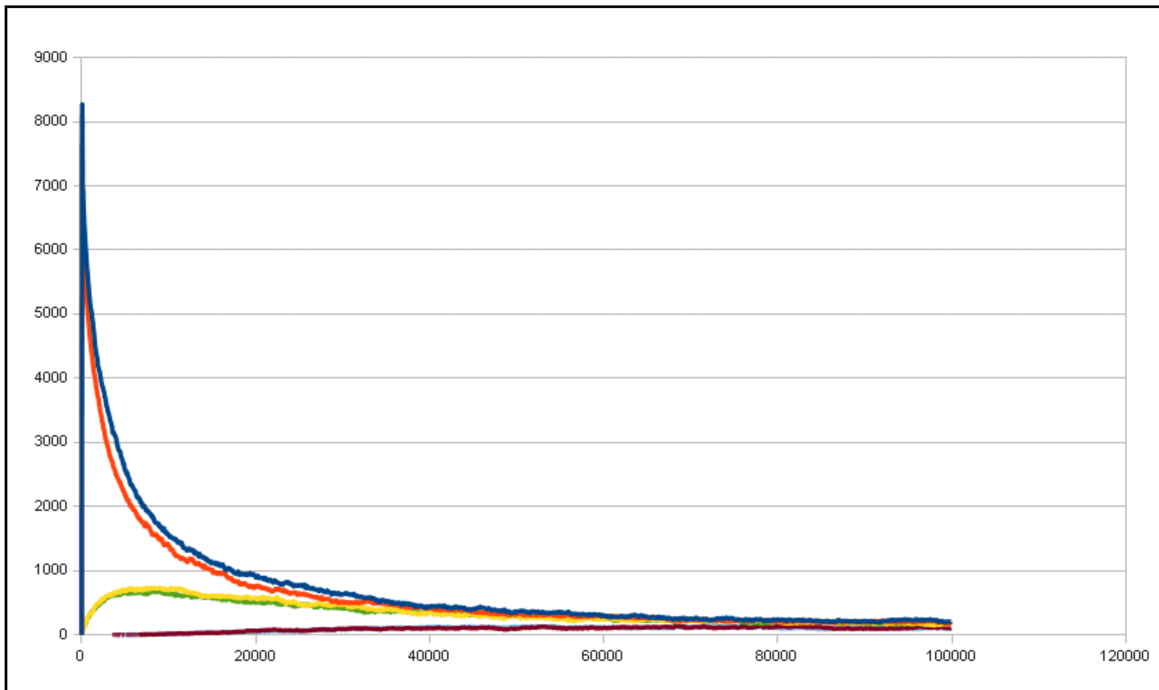


Figure 7.5. Received signals at 0 (red and blue), 200 (green and yellow) and 500nm from emitter for simulations with and without collisions respectively.

7.5 INERTIA

Inertia combined with initial velocity of particles produces a wave as shown in Figure 7.6.

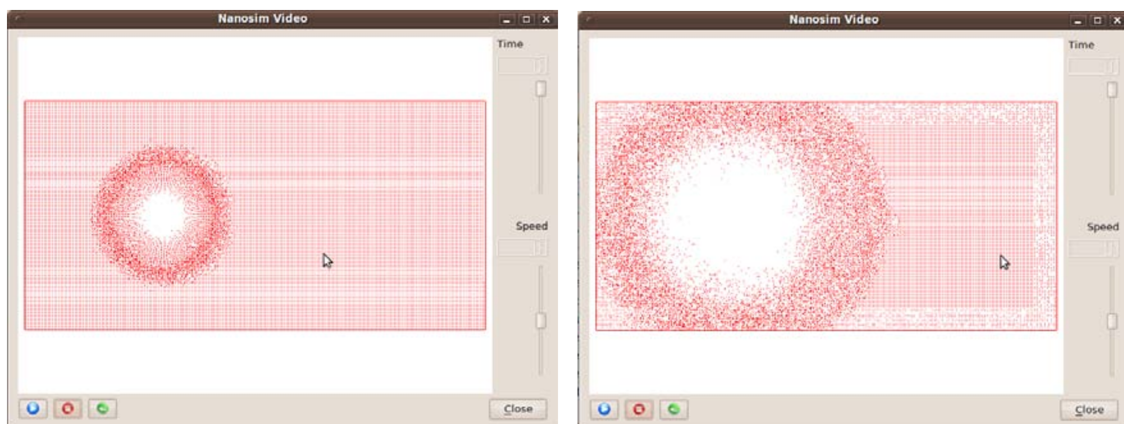


Figure 7.6. Inertia simulation with initial velocity.

7.5. ELECTROSTATIC FORCES.

The prototype that includes the effect of electric forces in the diffusion process shows that the pattern of diffusion is more homogeneous as shown in Figure 7.7. Perhaps this can lead to a decrease of noise. However, this prototype has not been validated yet and these effects have not been studied.

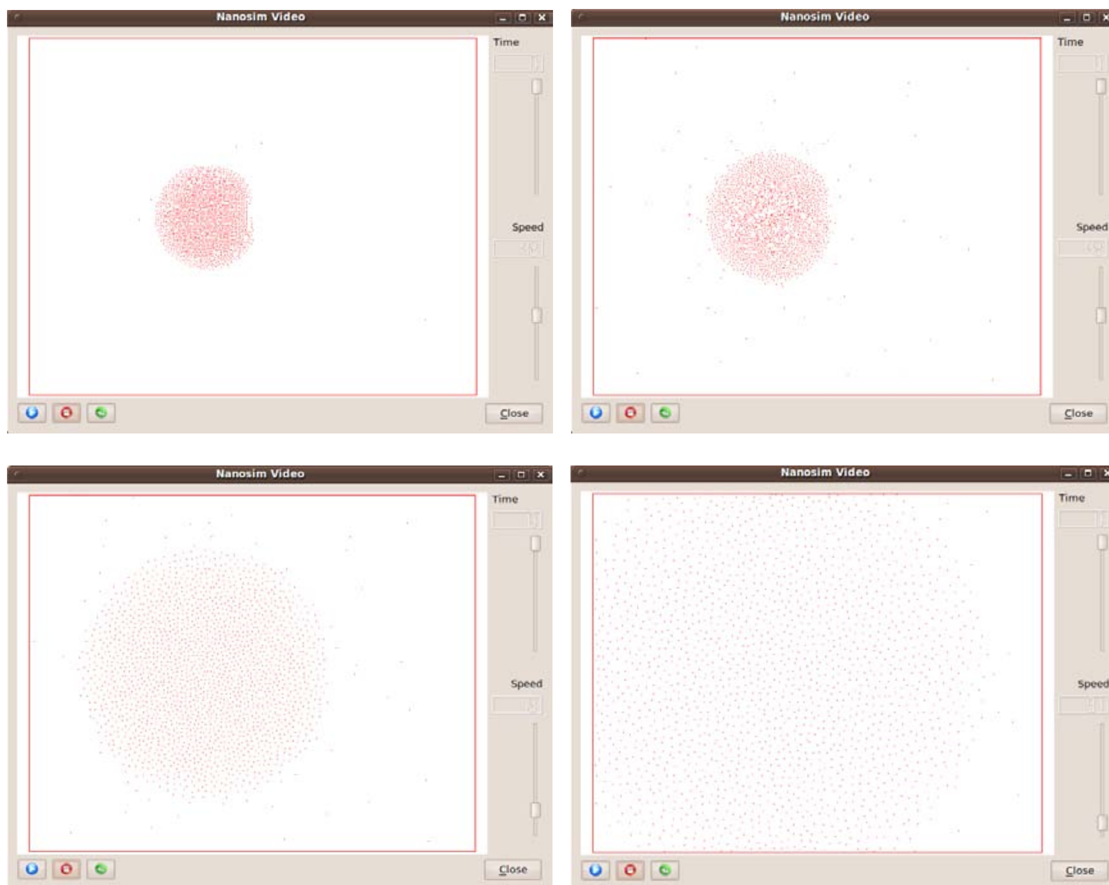


Figure 7.7 Four stages of the diffusion process by electrical forces.

8. FUTURE WORK.

In the short term, it would be interesting to develop the following tasks:

- Modify packages' structure of domain layer to improve modifiability.
- Validate implementation of electrical forces.
- Implement a new functionality: that users could create Boundary objects.
- Expand the application into three dimensions.

8.1. PACKAGE STRUCTURE

Spiral development, when adding functionalities to already functional prototypes, also creates a non-wanted effect against the modifiability requirement. For example: some functionalities that began as simples and they are included into a certain class, step by step they are growing up, until it is better to restructure the code so classes and modules do not grow up too much or in an inconsistent way what damages modifiability. It has also happened that functionalities starting as separated code parts, later on they have shown elements in common and it would be convenient to take them out from where they are and put them together to create new classes or modules. If these kinds of changes are not done, project's modifiability is going to be affected.

A study has been done at the end of the project about which would be recommendable changes on current implementation. The main change, explained in this section, is to modify the packages structure of the domain layer.

Figures 8.1 and 8.2 show the structure of current packages and the proposed structure respectively. As you can see, the structure appears quite altered but actually changes are reorganization of packages so it is not a change hard to do.

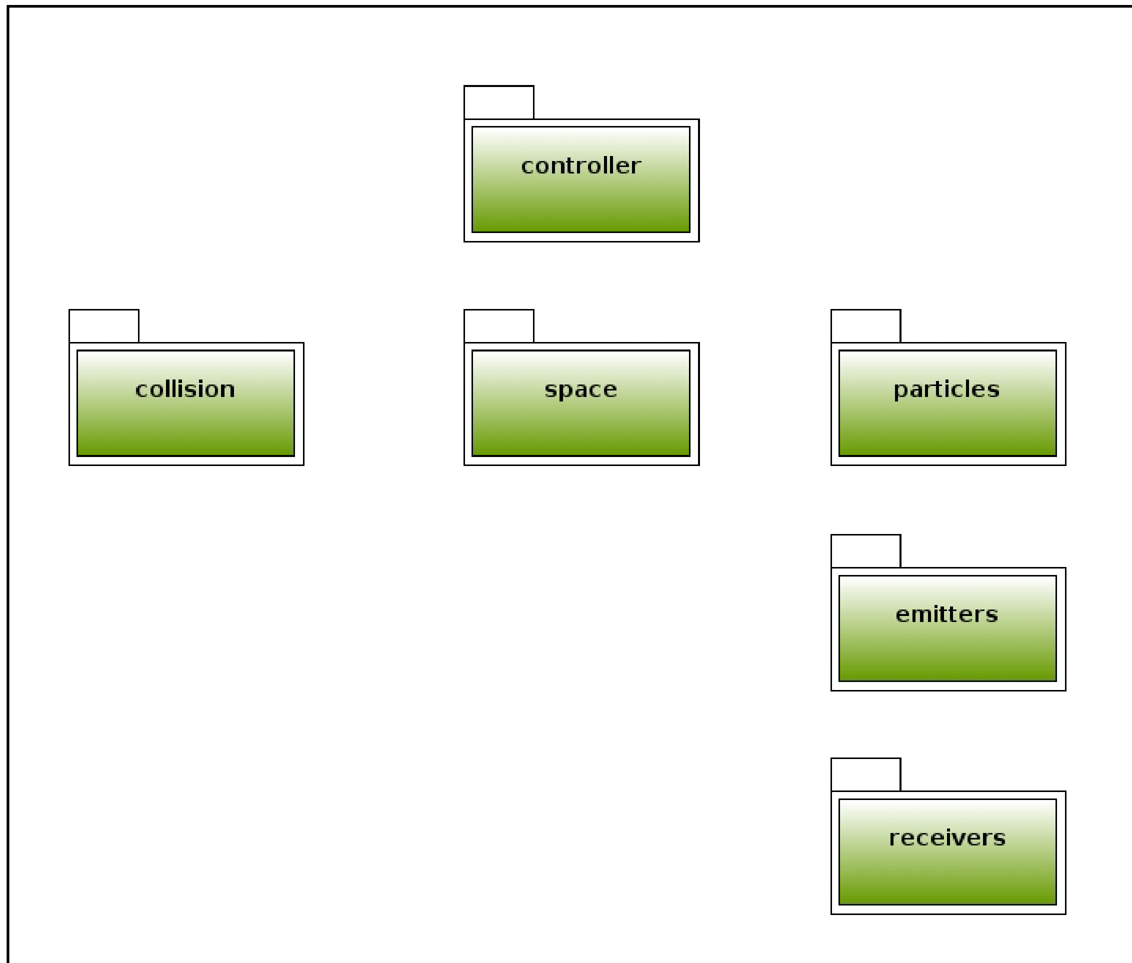


Figure. 8.1. Current package structure of the domain layer.

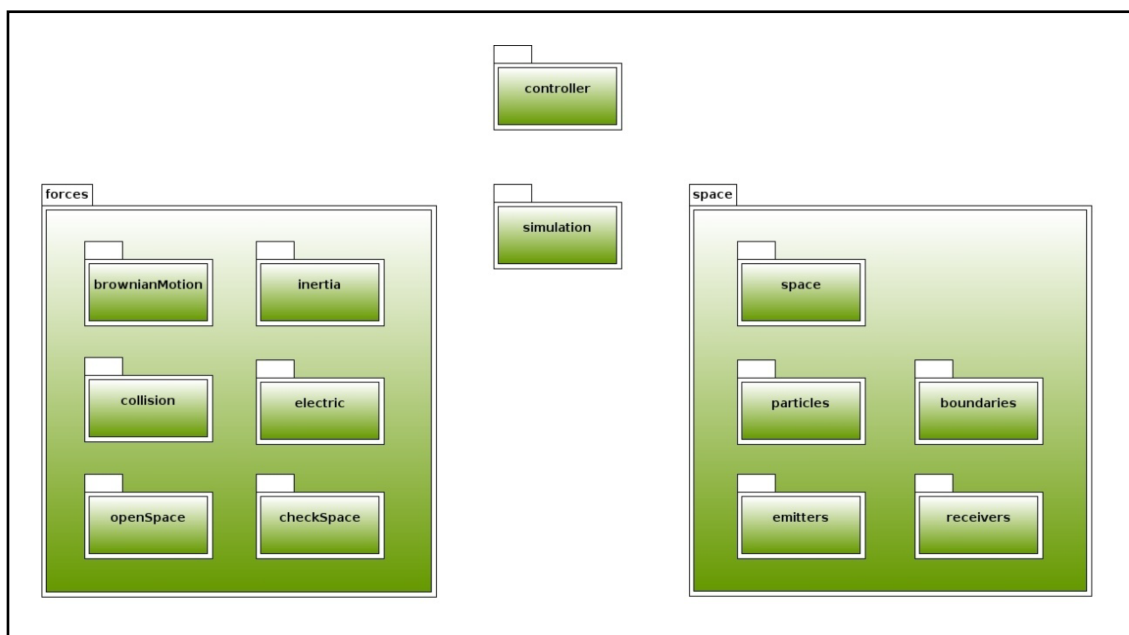


Figure. 8.2. Proposed package structure of the domain layer.

The main package at current structure is the package ***space*** that contains two classes: ***Simulator*** and ***Space***. The class ***Simulator*** is the one candidate to be a thread, it has a ***start*** method, a ***stop*** method and an object of space class. This class controls the space movement (with the control of the timestep and the operations that have to be applied to space at each time), while the class ***Space*** contains the data structures containing space components and all that necessary operations to modify it. The other packages (particle, emitter, receiver, boundaries) are auxiliary packages used by class ***Space***.

At current implementation the class ***Space*** has grown up too much because it has assumed the simulation of diffusion effects. This structure is heritage of the very first prototypes where data structures and operations were simples. It seems natural that those operations that modify the space were implemented in the class ***Space***. Over time, it has been observed that it complicates too much the class ***Space***, that it is a class with already many elements.

The suggested solution is the structure shown in figure 8.2. In this figure, the package ***space*** has been separated in two packages, ***simulation*** and ***space***. The package ***simulation*** would keep controlling time and it would have an attribute of the type ***Space***. The package ***space*** would have the data structures that contain the space elements: particles, emitters, receivers and boundaries.

The forces that model the effects of diffusion would have their own package. These forces would offer a static operation that would receive an object of the class ***Space*** as parameter. This way, it would be possible to apply the effects of diffusion to

the space from the class ***Simulator***, and the class ***Space*** would not have to know anything about diffusion laws (attributes and operations).

It will be necessary as well to do the same with attributes and operations referred to Brownian motion and inertia. And should be included the effect of electric forces as another package.

It would also be convenient to have two additional forces to model two actions. One is to simulate the infinite space. The other one is to do several tests about the space state. For example, to check that there is no particle out of the space limits (in case of limited space) or that two particles are not overlapping (if including collision effect). Currently these operations are responsibility of the class ***Space*** as well. This structure is more modifiable and extensible.

Part of this change has already been done; it has been created the package ***collision*** to model collisions among particles. However it is still necessary to pass the attribute **_activeCollisions (**which decides whether to apply the collision as part of the diffusion) to the class ***Simulator*** and make the call to the class ***Collisions*** directly from the class ***Simulator*** class instead of using the method ***checkCollisions*** of the class ***Space***.

It is still necessary to make the same change for Brownian motion, inertial forces, electrostatic forces and also to simulate infinite space.

8.2. ELECTRICAL FORCES.

If particles used for communication have electric charge, as is the case of signaling with calcium ions (Ca++), it is necessary to consider the effect of electric forces in the diffusion process. As explained in section 5.5, there is already a beta version of N3Sim that implements this functionality. However, it is necessary to perform simulations and study the results to validate the model. On the other hand, it is also necessary to receive feedback from users to modify the simulator so it best suits the needs of simulation.

8.3. BOUNDARIES.

A new functionality that would be interesting to implement is that the user can place boundaries into space in order to simulate obstacles. It is known that the existence of obstacles slows down the diffusion process. It may be interesting to have the right tools to characterize this effect through simulations.

When limits of system were modeled, the package ***boundary*** was created already considering this functionality. To implement it, it is only necessary to modify the user interface, so users can define boundaries, and implement the method that creates them and includes them in the boundaries list of the ***Space***.

There is no need for more changes in the simulator. When a boundary is in the boundaries list of the class ***Space***, the simulator will just make calculations of collisions with these boundaries.

In any case it might be interesting, and it would be discussed with users, to provide new functionalities to boundaries. For example, they could enable or disable or function as permeable membranes.

8.4. EXTENSION TO 3D.

Currently N3Sim has already been started to extend to three dimensions. As can be seen in the ***particle***, ***receiver*** and ***emitter*** packages, there are some classes that implement these elements in 3D. But so far it has only been modeled Brownian motion and in an open space

To finish N3Sim implementation in three dimensions is necessary to do the following:

- To implement the class ***Boundary*** in 3 dimensions. The boundaries are no longer straight lines but plans.
- To implement the algorithms of collisions between spheres and spheres and between spheres and boundaries in three dimensions.
- To implement emitters to hand out emitted particles in a volume of influence.
- To modify the user interface and the configuration file to pick up the additional variables needed for the above.

9. CONCLUSIONS.

This chapter evaluates the project from four points of view that correspond to the sections of the chapter. First, it evaluates the achievements of the objectives and requirements set out in chapters 1 and 3. Second, it values the development of the project itself. Third, results of the study of the communication channel are shown. Finally there is a personal assessment that reflects on what this project has brought to the student.

9.1. ACHIEVING GOALS.

The project achieved the objectives set out in section 1.4. The first objective was achieved because a simulator that models the diffusion process of the communication channel proposed in [2] has been built. And this simulator has been used to study and characterize the channel. The results of these studies were published in [5] [6] [7] [8] and [9]. In addition the simulator has been published under the GPL on the website of N3Cat, along with related publications and user guides.

To achieve the second objective the simulator has been implemented keeping in mind the non-functional requirements of modifiability and extensibility. It has been planned that the simulator can easily include future classes to model the transmission and reception processes that complete the definition of the communication channel [2]. As explained in section 5.4.3.4, to include a new model of emitter it is only necessary that the class that implements it has a method ***emit***. The same applies to receivers, receivers only have to implement a method ***count*** (5.4.3.5).

N3Cat
Nanonetworking Center in Catalunya

NaNoNetworking Center in Catalunya

UNIVERSITAT POLITÈCNICA DE CATALUNYA BARCELONATECH

Home People Projects Publications Events

Third NaNoNetworking Summit

N3Sim: A Simulation Framework for Diffusion-based Molecular Communication

N3Sim is a complete simulation framework for diffusion-based molecular communications, which allows the evaluation of the communication performance of molecular networks with several transmitters and receivers in an infinite space with a given concentration of molecules. Transmitters encode the information by releasing particles into the medium, thus varying the concentration rate in their vicinity. The diffusion of particles through the medium is modeled as Brownian motion, taking into account particle inertia and collisions among particles. Finally, receivers decode the information by sensing the local concentration in their neighborhood. The benefits of such a simulator are multiple: the validation of existing channel models for molecular communications and the evaluation of novel modulation schemes are just two examples.

- Download
- Quick Start Guide
- User Guide
- Parameters List

Publications

- Llatser, I., Alarcón, E., Pierobon, M., ***"Diffusion-based Channel Characterization in Molecular Nanonetworks"***, in Proc. of the 1st IEEE International Workshop on Molecular and Nano Scale Communication (MoNaCom), held in conjunction with IEEE INFOCOM, April 2011.
- Garralda, N., Llatser, I., Cabellos-Aparicio, A., Pierobon, M., ***"Simulation-based Evaluation of the Diffusion-based Physical Channel in Molecular Nanonetworks"***, in Proc. of the 1st IEEE International Workshop on Molecular and Nano Scale Communication (MoNaCom), held in conjunction with IEEE INFOCOM, April 2011.
- Llatser, I., Pascual, I., Garralda, N., Cabellos-Aparicio, A., Pierobon, M., Alarcón, E. and Solé-Pareta, J., ***"N3Sim: A Simulation Framework for Diffusion-based Molecular Communication,"*** IEEE TC on Simulation, No. 8, pp. 3-4, March 2011.
- Llatser, I., Pascual, I., Garralda, N., Cabellos-Aparicio, A., Pierobon, M., Alarcón, E. and Solé-Pareta, J., ***"Exploring the Physical Channel of Diffusion-based Molecular Communication by Simulation"***, submitted for publication, March 2011.

Aknowledgements

N3Sim has been developed by:

Figure. 9.1. Screenshot of N3Sim website.

9.2. PROJECT DEVELOPMENT.

Development of the project has been very dynamic; from February 2010 until July 2010 the group worked on the basis of weekly meetings usually involving four to eight people from N3Cat. During this period, the simulator was built and began the work to characterize the communication channel.

After this period, some improvements in the simulator were implemented and more work on the study the communication channel was done. Finally, some papers [5] [6] [7]. [8], the Master Thesis by Nora Garralda [9] and this report were written.

The way the project has been developed, with many prototypes, made me think and learn about the organization of a software project. Making good use of the features that give us the software development tools and organizing properly are two keys to a fast and quality development.

9.3. STUDY OF THE MOLECULAR COMMUNICATION CHANNEL.

Part of N3Sim working group has made a considerable effort studying the molecular communication channel. Using the simulator they were able to perform the experiments needed to obtain information on the basic parameters of a communication channel.

Below some of these results are discussed. You can delve into this subject in the papers [6] [7] and [8] and Garralda Nora's thesis [9].

The simulations proved that the molecular diffusion-based channel has the LTI (Linear Time Invariant) property. A Linear Time-Invariant (LTI) channel fulfills the superposition principle, as well as maintains its features over time.

The transmission of several pulse shapes has been simulated. From the simulations two conclusions arise: i) the optimal pulse shape is a spike, a very narrow pulse, which gives the lowest pulse width at the receiver and thus the highest achievable bandwidth, and ii) the shape of the transmitted pulses is not important, since the total energy received is not shape dependent.

The values of which follow the bandwidth and the gain of the channel have also been studied. In Figure 9.3 shows the values compared with electromagnetic communication channel [8].

Metric	EM channel	Molecular channel
Pulse delay	$\Theta\left(r\right)$	$\Theta\left(r^2\right)$
Pulse amplitude	$\Theta\left(1/r^2\right)$	$\Theta\left(1/r^3\right)$
Pulse width	$\Theta\left(1\right)$	$\Theta\left(r^2\right)$

Figure 9.2. Molecular vs Wireless EM Channel Comparison [8].

It has also been studied and characterized the noise that is inherent in the Brownian motion diffusion (Particle Counting Noise). It has also concluded that the observations are consistent with the theories discussed in [4].

9.4. PERSONAL ASSESSMENT.

My evaluation cannot be other than excellent. Participation in this project has given me many things. I have learned about programming tools like Java and Eclipse and SVN. I have worked with a great team with whom I have learned and discussed about programming, physics, chemistry, biology and nanotechnology. And I have actively participated in a project whose outcome is already used in research.

I take this opportunity to express my gratitude to all members of the group N3Cat and especially those who have worked on the project N3Sim: Dr. Josep Solé Pareta, Dr. Ian F. Akyildiz, Dr. Eduard Alarcón-Cot, Dr. Albert Cabellos-Aparicio, Ignacio Llatser Martí, Massimiliano Pierobon, and Nora Garralda Torres. I have been very fortunate to have the chance to work with this great team.

10. REFERENCES

[1] I. F. Akyildiz, F. Brunetti, and C. Blazquez, ***"Nanonetworks: A new communication paradigm"***, Computer Networks, vol. 52, no. 12, pp. 2260–2279, 2008.

[2] M. Pierobon and I. F. Akyildiz, ***"A physical end-to-end channel model for nanonetworks"***, IEEE Journal on Selected Areas in Communications, vol. 28, no. 4, pp. 602–611, 2010.

[3] A. Einstein ***"On the movement of small particles suspended in a stationary liquid demanded by the molecular kinetic theory of heat."***, Ann.d.Phys. 17(1905)549-560.

[4] Pierobon, M. and Akyildiz, I. F., ***"Diffusion-based Noise Analysis for Molecular Communication in Nanonetworks"***, to appear in IEEE Transactions on Signal Processing, 2011.

[5] Llatser, I., Pascual, I., Garralda, N., Cabellos-Aparicio, A., Alarcón, E., ***"N3Sim: A Simulation Framework for Diffusion-based Molecular Communication"***, IEEE TCSIM Newsletter, Issue 08 March 2011, pg. 3.

[6] Llatser, I., Pascual, I., Garralda, N., Cabellos-Aparicio, A., Pierobon, M., Alarcón, E. and Solé-Pareta, J., ***"Exploring the Physical Channel of Diffusion-based Molecular Communication by Simulation"*** submitted for publication at IEEE GLOBECOM 2011, March 2011.

[7] Garralda, N., Llatser, I., Cabellos-Aparicio, A., Pierobon, M., ***"Simulation-based Evaluation of the Diffusion-based Physical Channel in Molecular Nanonetworks"***, in Proc. of the 1st IEEE International Workshop on Molecular and Nano Scale Communication (MoNaCom), held in conjunction with IEEE INFOCOM, April 2011.

[8] Llatser, I., Alarcón, E., Pierobon, M., ***"Diffusion-based Channel Characterization in Molecular Nanonetworks"***, in Proc. of the 1st IEEE International Workshop on Molecular and Nano Scale Communication (MoNaCom), held in conjunction with IEEE INFOCOM, April 2011.

[9] Nora Garralda ***"Diffusion-based Physical Channel Identification for Molecular Nanonetworks"***, Master Thesis 2010, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya.

[10] Ho Kyung Kim, Leonidas J. Guibas, and Sung Yong Shin, ***"Efficient Collision Detection among Moving Spheres with Unknown Trajectories"***, Algorithmica (2005) 43: 195–210.

[11] Hu Qi-tu, Deng Xiao, Zhang Xiao-ling, ***"Simulation on random motion of numerous spheres with collision-event-driven approach"***, Aug. 2009, Volume 6, No. Bbb 8 (Serial No.57), Journal of Communication and Computer, ISSN 1548-7709, USA.

[12] Baraff, D. (1992), ***"Dynamic Simulation of Non-Penetrating Rigid Bodies, (Ph. D thesis) "***, Computer Science Department, Cornell University, pp. 52–56

[13] Sam Stokes ***"Collision detection in the simulation of rigid body motion"*** Computer Science Tripos Part II , Robinson College, September 29, 2005

[14] E. Gul, B. Atakan, O. B. Akan, ***"NanoNS: A Nanoscale Network Simulator Framework for Molecular Communications"***, Nano Communication Networks Journal (Elsevier), Vol. 1, No. 2, pp. 138-156, June 2010.

[15] Tenn Francis Chen, Gladimir V. G. Baranoski ***"BSim: A System for Three-Dimensional Visualization of Brownian Motion"***, School of Computer Science , University of Waterloo , Ontario, Canada.

ANNEX I : PUBLICATIONS DERIVED FROM THIS PROJECT

- Llatser, I., Alarcón, E., Pierobon, M., ***"Diffusion-based Channel Characterization in Molecular Nanonetworks",*** in Proc. of the 1st IEEE International Workshop on Molecular and Nano Scale Communication (MoNaCom), held in conjunction with IEEE INFOCOM, April 2011. (*)
- Garralda, N., Llatser, I., Cabellos-Aparicio, A., Pierobon, M., ***"Simulation-based Evaluation of the Diffusion-based Physical Channel in Molecular Nanonetworks",*** in Proc. of the 1st IEEE International Workshop on Molecular and Nano Scale Communication (MoNaCom), held in conjunction with IEEE INFOCOM, April 2011. (*)
- Llatser, I., Pascual, I., Garralda, N., Cabellos-Aparicio, A., Pierobon, M., Alarcón, E. and Solé-Pareta, J., ***"N3Sim: A Simulation Framework for Diffusion-based Molecular Communication,"*** IEEE TC on Simulation, No. 8, pp. 3-4, March 2011. (*)
- Llatser, I., Pascual, I., Garralda, N., Cabellos-Aparicio, A., Pierobon, M., Alarcón, E. and Solé-Pareta, J., ***"Exploring the Physical Channel of Diffusion-based Molecular Communication by Simulation",*** submitted for publication, March 2011.
- Nora Garralda ***"Diffusion-based Physical Channel Identification for Molecular Nanonetworks"***, Master Thesis 2010, Departament d'Arquitectura de Computadors, Universitat Politècnica de Catalunya.

(*) Available at http://www.n3cat.upc.edu/n3sim

ANNEX II : QUICK START GUIDE

OVERVIEW

The goal of this quick start guide is to give the reader the minimum instructions to install N3Sim (version0.7) and run a simple simulation. Sections Installation, Running N3Sim and Output explain the basic steps to run a simulation. In section Next Step: Automating Simulations, short indications to automate multiple simulations are given. In order to get a deeper knowledge of N3Sim, reading the User Guide is highly recommended.

CONTENTS

- Installation.
- Running N3Sim.
- Output.
- Next Step: Automating Simulations
- Example 1
- Example 2

INSTALLATION

- The only prerequisite is Java JRE 1.6.
- Download the jar file N3Sim.jar and the configuration file N3Sim.cfg from here and save them in your N3Sim folder.

RUNNING N3SIM

- Edit the values of the parameters in the downloaded configuration file (N3Sim.cfg) in order to suit your needs. A list of parameters together with a short explanation of them is available here.
- Type the following command from the N3Sim folder in your console:

  > java -jar N3Sim-0.7.jar myConfigFile.cfg

  Tip: for heavy simulations it might be a good idea to increase the Java memory to a higher value, such as 1024 MB.

  > java -jar -Xms1024m -Xmx1024m N3Sim-0.7.jar myConfigFile.cfg

- When the simulation is completed, you just need to check your results (see the next section Output).

OUTPUT

Once the simulation has finished, the application will have created a folder under the N3Sim folder with the name specified by the variable **outPath** in the configuration file. Inside this folder, the following files may be found:

- A file *simulation.txt* with the parameters and values used in the simulation.
- A file *receiver_name.csv* for each receiver in the simulation. These files have two columns; the first one contains the time steps in nanoseconds, and the second one represents the number of particles measured by the receiver at the corresponding time step.
- Debugging files *info.log* (optional) and *error.log*.
- Video file *graphic.ns1* (optional), to be used with NSVideo.

NEXT STEP: AUTOMATING SIMULATIONS

- Read example 1 and/or example 2.
- Modify your configuration file and build your script.
- Run the script.
- Check your results.

EXAMPLE 1

Let's assume that we want to run several 2-dimensional simulations with particle radius 1, 2, 5 and 10, and we want to repeat each of them 10 times. When using the key word *"param"* as value for any of the parameters in the configuration file, the program will read its value from the parameters list of the execution command (in the same order that they appear at the configuration file). In this example, we give the value *"param"* to the variables **outPath** (so that each simulation has a different folder) and **sphereRadius,** as shown in Figure 1. Then, a script such as the one shown in Figure 2 will execute all the required simulations automatically.

```
### N3Sim CONFIG FILE

### SIMULATION PARAMS

outPath=param
graphics=false
infoFile=true
activeCollision=false
BMFactor=1
inertiaFactor=0
time=200000
timeStep=1000

### SPACE PARAMS

boundedSpace=true
constantBGConcentration=false
constantBGConcentrationWidth=0
xSize=2500
ySize=2000
D=1
bgConcentration=10
sphereRadius=param

### EMITTER PARAMS

emitters=1
emitterRadius=100
x=1000
y=500
startTime=1000
endTime=2000
initV=0
punctual=false
concentrationEmitter=false
color=white
emitterType=1
amplitude=1000

### RECEIVER PARAMS

receivers=1
name=rx500
x=1500
y=500
absorb=false
accumulate=false
end=true
```

Figure 1. *Configuration file used to launch simulations with different values of the parameter sphereRadius.*

```
#!/bin/sh

# This script will launch 10 simulations for each value of the PARAM1_LIST.
# Simulations results will be stored in folders named "myTest-i-j", where i
# is the effective value of the PARAM1_LIST for each simulation and j
# is the 1 to 10 repetition for each i.

N=10
PARAM1_LIST=(1 2 5 10)

for (( j = 0 ; j < $N; j++ )); do

for i in ${PARAM1_LIST[@]}; do

java -jar N3Sim.jar myConfigFile.cfg myTest-${i}-${j} $i

done

done
```

Figure 2. *Script to automate simulations for a list of values of a parameter and repeat each simulation 10 times.*

EXAMPLE 2

Let's assume that we want to run several 2-dimensional simulations combining several values for the particle radius (values 1, 2, 5 and 10) with several values for the diffusion coefficient D (0.5, 0.6, 0.7, 0.8 and 0.9). When using the key word *"param"* as value for any of the parameters in the configuration file, the program will read its value from the parameters list of the execution command (in the same order that they appear at the configuration file). In this example, we give the value *"param"* to the variables **outPath** (so that each simulation has a different folder), **sphereRadius** and **D**, as shown in Figure 3. Then, the script shown in Figure 4 will execute all the required simulations automatically.

```
### N3Sim CONFIG FILE

### SIMULATION PARAMS

outPath=param
graphics=false
infoFile=true
activeCollision=false
BMFactor=1
inertiaFactor=0
time=200000
timeStep=1000

### SPACE PARAMS

boundedSpace=true
constantBGConcentration=false
constantBGConcentrationWidth=0
xSize=2500
ySize=2000
D=param
bgConcentration=10
sphereRadius=param

### EMITTER PARAMS

emitters=1
emitterRadius=100
x=1000
y=500
startTime=1000
endTime=2000
initV=0
punctual=false
concentrationEmitter=false
color=white
emitterType=1
amplitude=1000

### RECEIVER PARAMS

receivers=1
name=rx500
x=1500
y=500
absorb=false
accumulate=false
```

Figure 3. *Configuration file used to launch simulations combining different values of the parameters **sphereRadius** and **D**.*

```
#!/bin/sh

# These script will launch simulations combining values of PARAM1_LIST
# with values of PARAM2_LIST.
# Simulations results will be stored in folders named "myTest-i-j"
# where i and j are the values of the parameters.

PARAM1_LIST=(1 2 5 10)
PARAM2_LIST=(0.6 0.7 0.8 0.9)

for i in ${PARAM2_LIST[@]}; do

for j in ${PARAM2_LIST[@]}; do

java -jar N3Sim.jar myConfigFile.cfg myTest-${i}-${j} $i $j

done

done
```

Figure 4. *Script to automate simulations combining a list of values of a parameter with a list of values for another parameter.*

ANNEX III : USER GUIDE

OVERVIEW

N3Sim is a simulation framework for diffusion-based molecular communication in nanonetworks, a bio-inspired paradigm based on the use of molecules to encode and transmit information.

N3Sim simulates a set of nanomachines which communicate through molecular diffusion in a fluid medium. The information to be sent by the emitter nanomachines modulates the rate at which they release particles to the medium. The variation in the local concentration generated by the emitters propagates throughout the medium. The receivers are able to estimate the concentration of particles in their neighborhood by counting the number of particles in the environment. From this measurement, they can decode the transmitted information.

This is the user guide for the version 0.7 of N3Sim. It includes a general description of the simulator in sections Current Features and N3Sim Block Diagram. Installation and execution instructions may be found in sections Installation, Running Simulations, Output and Automating Simulations. Finally, more specific concepts about the simulator are explained in the remaining sections.

Contents

- Current Features
- Roadmap
- N3Sim Block Diagram
- Installation
- Running Simulations
- Output
- Automating Simulations
- 3-Dimensional Simulations
- Time and Time step
- Anomalous Diffusion Components
- Simulation Space
- Simulating an Infinite Space
- Emitters
- Receivers

CURRENT FEATURES

N3Sim is a prototype still under development. Some of its features are therefore under testing and are not available in the current released version (v0.7). For instance, the influence of electrostatic forces in diffusion is not included in v0.7, and 3-dimensional simulations are only possible when the particles move according to Brownian motion in an unbounded space.

The current features of N3Sim are:

- Brownian motion and components of anomalous diffusion: particle inertia and elastic collisions for 2-dimensional space. (A prototype that includes the electrostatic forces among particles is currently under testing.)
- All components of diffusion can be activated/deactivated.
- Bounded or unbounded space.
- Initial background homogenous concentration (finite or infinite space).
- Any number of emitters and receivers.
- Predefined released patterns for each emitter.
- Input file (.txt) defined by the user containing a released pattern for each emitter.
- 3-dimensional simulation (only for Brownian motion an unbounded space).

ROADMAP

The current stable version of N3Sim is v0.7. Some of the new features that will be included in next versions are:

- v0.8: electric forces as a component of anomalous diffusion.
- v0.9: obstacles in the simulation space.
- v1.0: complete 3-dimensional simulations.

N3SIM BLOCK DIAGRAM

Figure 1 shows a block diagram of the steps needed to run a simulation.

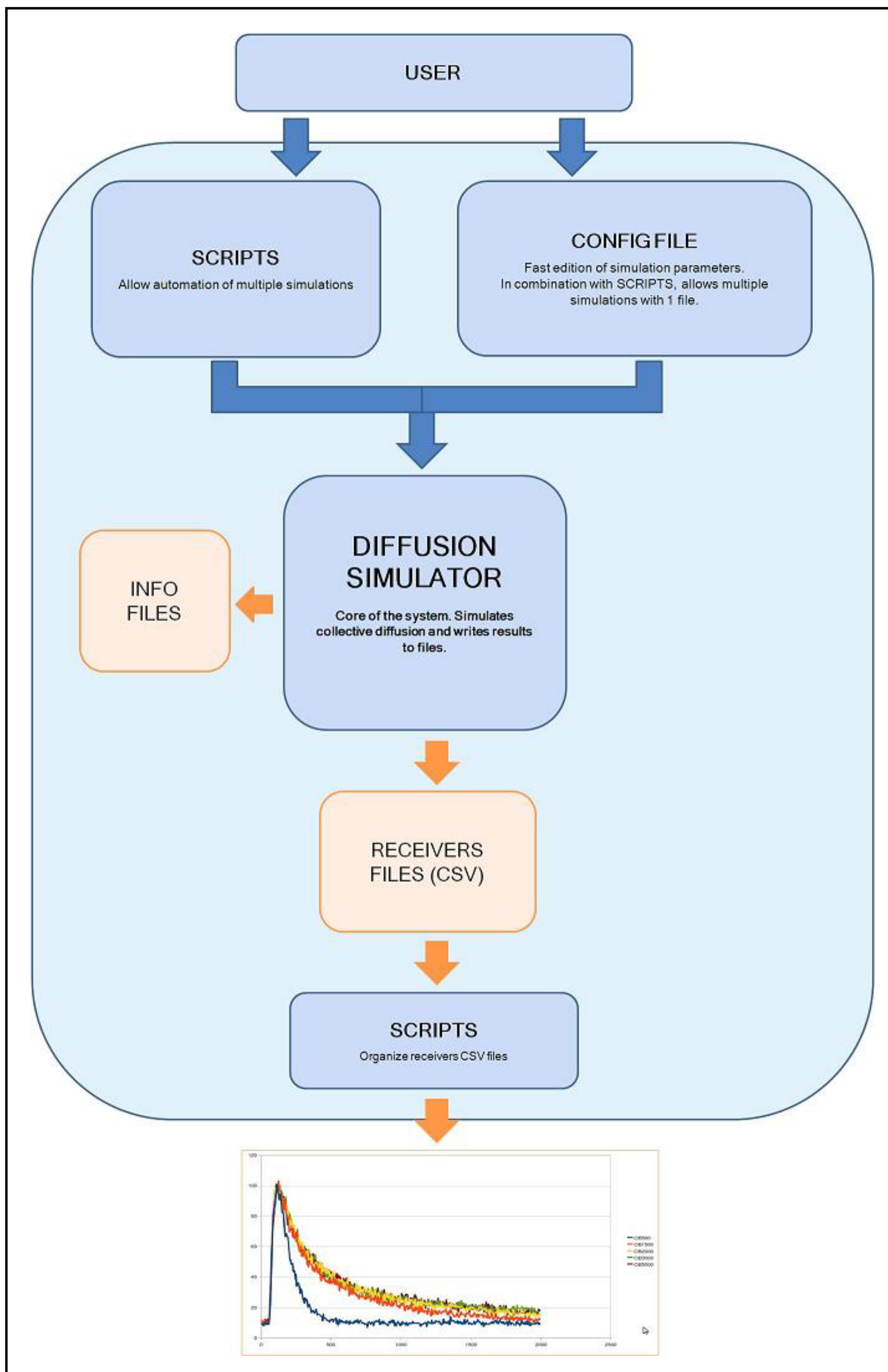


Figure 1. N3Sim block diagram

First, the user communicates with the simulator using a configuration file and, optionally, a script. In the configuration file, the user specifies the values of all the simulation parameters. These parameters include the number and location of emitters and receivers, the waveform of the emitter, the size of the emitted particles and the diffusion coefficient of the medium, amongst others. Some of these parameters can be left unspecified and can be defined using a script; this allows the user to run multiple simulations automatically with only one configuration file and one script. This is useful to easily evaluate the influence of a specific parameter (e.g., the number of transmitted particles) in the system output.

Next, the diffusion simulator takes the configuration file and the automation scripts as input and performs the actual simulation. The diffusion simulator writes its results to files. To do so, it creates a new folder with the simulation name and in this folder creates one file per receiver and some other files with the simulation information. The receiver files are text files in csv format, they contain the concentration measured by each receiver as a function of time. Last, another set of scripts may be used to organize the results from several receivers and graphically represent them into a single plot.

INSTALLATION

N3Sim is a Java runnable jar file. It is OS-independent as long as the Java VM is installed in the system. In order to use N3Sim, Java 1.6 or higher is required. If you do not have Java in your system, you may download it from here.

In order to install N3Sim, just download the executable jar file and the configuration file from here.

RUNNING SIMULATIONS

In order to run a simulation with N3Sim, you first need to edit the values of the simulation parameters. To do so, just open the text file N3Sim.cfg previously downloaded, which has all the parameters set to their default values. You may edit them in order to fit your requirements and then save the configuration file with the desired name. A list of parameters together with a brief explanation of them is available here. A more comprehensive explanation of some of the parameters can be found in the following sections of this guide.

Once the configuration file has the desired values, enter the following command in a Unix shell or Windows command prompt from your N3Sim directory::

```
> java -jar N3Sim-0.7.jar myConfigFile.cfg
```

For simulations with a high number of particles, you may want to increase the amount of Java memory up to, e.g., 1024M.

```
> java -jar -Xms1024m -Xmx1024m N3Sim-0.7.jar myConfigFile.cfg
```

This will start the simulation. Once it is complete, just check its results (see section Output).

OUTPUT

Once the simulation has finished, the application will have created a directory under the N3Sim folder with the name specified by the parameter **outPath** in the configuration file. Inside this folder, the following files may be found:

- A file *simulation.txt* with the parameters and values used in the simulation.
- A file *receiver_name.csv* for each receiver in the simulation. These files have two columns; the first one contains the time steps in nanoseconds, and the second one represents the number of particles measured by the receiver at the corresponding time step.
- Debugging files *info.log* (optional) and *error.log*.
- Video file *graphic.ns1* (optional), to be used with NSVideo.

AUTOMATING SIMULATIONS

Often user is interested in performing multiple identical simulations in order to compute the average results, or to check how the results change depending on the values of one or more parameters.

N3Sim provides the following mechanism to facilitate the execution of multiple simulations. In the configuration file, one or more parameters can take the key value *"param"*. In this case, the program expects to receive these parameters from the execution command (in the same order that they are in the configuration file.) For instance, if the value "param" is given to the parameters **time** and **timeStep**, the simulation may be executed as follows:

```
> java -jar N3Sim-0.7.jar myConfigFile.cfg 2000 5
```

which is equivalent to give the values **time=**2000 and **timeStep=**5 in the configuration file.

In this way, scripts can be used to launch multiple simulations with different parameters. An example: let's imagine we want to run several 2-dimensional simulations combining different values of particle radius (1, 2, 5 and 10 nm) with several values of diffusion constant D (0.5, 0.6, 0.7, 0.8 and 0.9 nm^2/ns).

By using the word *"param"* as the value for any parameter, the program will read its value from the parameters list of the execution command (in the same order that they appear in the configuration file).

In this example, we give the value *"param"* to the parameters **outPath** (so that each simulation has a different folder), **sphereRadius** and **D**, as shown in figure 2. Then, the script shown in figure 3 will run the multiple simulations.

```
### N3Sim CONFIG FILE
### SIMULATION PARAMS
outPath=param
graphics=false
infoFile=true
activeCollision=false
BMFactor=1
inertiaFactor=0
time=200000
timeStep=1000

### SPACE PARAMS
boundedSpace=true
constantBGConcentration=false
constantBGConcentrationWidth=0
xSize=2500
ySize=2000
D=param
bgConcentration=10
sphereRadius=param

### EMITTER PARAMS
emitters=1
emitterRadius=100
x=1000
y=500
startTime=1000
endTime=2000
initV=0
punctual=false
concentrationEmitter=false
color=white
emitterType=1
amplitude=1000

### RECEIVER PARAMS
receivers=1
name=rx500
x=1500
y=500
absorb=false
accumulate=false
end=true
```

Figure 2. Configuration file used to launch simulations combining different values of the parameters ***sphereRadius*** and **D**.

```
#!/bin/sh

# These script will launch simulations combining values of PARAM1_LIST
# with values of PARAM2_LIST.
# Simulation results will be stored in folders named "myTest-i-j",
# where i and j are the values of the parameters.

PARAM1_LIST=(1 2 5 10)
PARAM2_LIST=(0.6 0.7 0.8 0.9)

for i in ${PARAM2_LIST[@]}; do

for j in ${PARAM2_LIST[@]}; do

java -jar N3Sim.jar myConfigFile.cfg myTest-${i}-${j} $i $j

done

done
```

Figure 3. Script to automate simulations combining lists of values of two parameters.

3-DIMENSIONAL SIMULATIONS

N3Sim was primarily developed for 2-dimensional scenarios. In the current version, 3-dimensional simulations are possible only under the following conditions:

- Unbounded simulation space
- No collisions among the emitted particles
- Emitter type 5 (see section Emitters)
- Receiver type 3 (see section Receivers)

A specific configuration file for 3-dimensional scenarios, N3Sim3D.cfg, is available. Although the general N3Sim configuration file can also be used for 3-dimensional simulations, it is easier to use the specific 3D configuration file. In this file, the parameters have a predefined value that fulfills the above conditions for 3-dimensional simulations (e.g., **boundedSpace** = *false*), and the parameters that may be changed by the user are underlined for easier identification. The template configuration file for 3-dimensional simulations may be downloaded here.

TIME AND TIMESTEP

The total time of the simulation is set with the parameter **time**. Time is discrete in the simulator, which advances time by steps (time steps). At each time step, emitters release particles, the simulator moves the released particles according the laws of diffusion, and receivers measure the concentration at their location. Both the parameters **time** and **timeStep** must be integers and their unit is nanoseconds.

An important question is: which is the best time step for a simulation? In order to answer it, several criteria must be taken into account. First, it must be noticed that the signal measured by receivers will have a resolution of one time step. The user must thus decide which is the maximum timestep so that the obtained results are useful.

On the other hand, in general, the higher the time step the shorter the simulation running time. However, there are several other factors which limit the maximum value of the time step, which are detailed as follows.

First, when modeling Brownian motion and/or inertia, if the time step is too large particles may move in a single time step a long distance, compared to the emitter-receiver distance. The mean distance that particles travel in a time step is sqrt(2*DiffusionConstant*timestep), according to the theory of Brownian motion . In order for the simulation to give reasonably correct results, this value should not exceed the 25% of the distance between emitter and receiver (or the minimum distance between an emitter and a receiver, if there is more than one of them).

For simulations that include collisions among particles, it is still true that the longer the time step, the shorter execution time. However, using a longer time step may cause the number of collisions which happen in every time step to rapidly increase. In this cases, using a smaller time step may result in a less memory consuming simulation and faster execution times.

A useful criteria to check whether the time step used is too large is to perform the same simulation with a smaller time step. If the results are not the same, it means that the time step used in the first simulation was too large. The smaller valid time step is of about 0.0001 nanoseconds.

ANOMALOUS DIFFUSION COMPONENTS

N3Sim models anomalous diffusion with four components: Brownian motion, particle inertia, collisions among particles and electrostatic forces.

Electrostatic forces are not yet included in the current version. There is a prototype that includes this component of diffusion, but it is still under testing and has not been released yet.

All the other components can be enabled or disabled. Collisions are activated by the parameter **activecollisions**, which may be found in the configuration file. Brownian motion and particle inertia may be enabled/disabled through the parameters **BMFactor** and **inertiaFactor**, respectively.

Brownian motion models the basic diffusion process which causes the random movement of the emitted particles at every time step. Brownian motion is modeled as a Gaussian random parameter with zero mean and whose root mean square displacement in each dimension after a time t is sqrt(2*DiffusionConstant*timestep). N3Sim multiplies this displacement by the parameter **BMFactor**. Therefore, if **BMFactor** is zero no Brownian Motion is performed.

The parameter **inertiaFactor** accounts for the inertia of particles. At every time step, N3Sim adds a displacement equal to the velocity of the previous time step multiplied by this factor. So, as with Brownian motion, if the **inertiaFactor** is zero, the particles will have no inertia.

SIMULATION SPACE

The simulation space contains the particles, emitters and receivers. Particles are modeled as spheres and its radius is set with the parameter **sphereRadius**.

The simulation space can be bounded or unbounded. This is controlled by the parameter **boundedSpace**. If it is true, a rectangular bounded space is simulated. The left/bottom corner of the space is (0,0) and the right/top corner is set at the values of parameters **xSize** and **ySize**. In this case, the emitted particles rebound at the space limits (even if the parameter **activeCollision** is set to false).

The simulation space can be filled before the simulation starts with an initial background concentration. This is set with the parameter **bgConcentration,** which must be an integer. The units of the background concentration are the number of particles per 10000 nm^2. If **bgConcentration** is not zero, the simulation space must be bounded (**boundedSpace** = true).

SIMULATING AN INFINITE SPACE

When running simulations with a bounded space, an undesired effect may appear. As emitters release particles, the background concentration will increase indefinitely. Figure 4 shows the signal at a receiver for the same simulation with bounded and with unbounded space. With bounded space (blue line) the tail of the signal is higher than with unbounded space (red line), because of the increase in the background concentration over time.

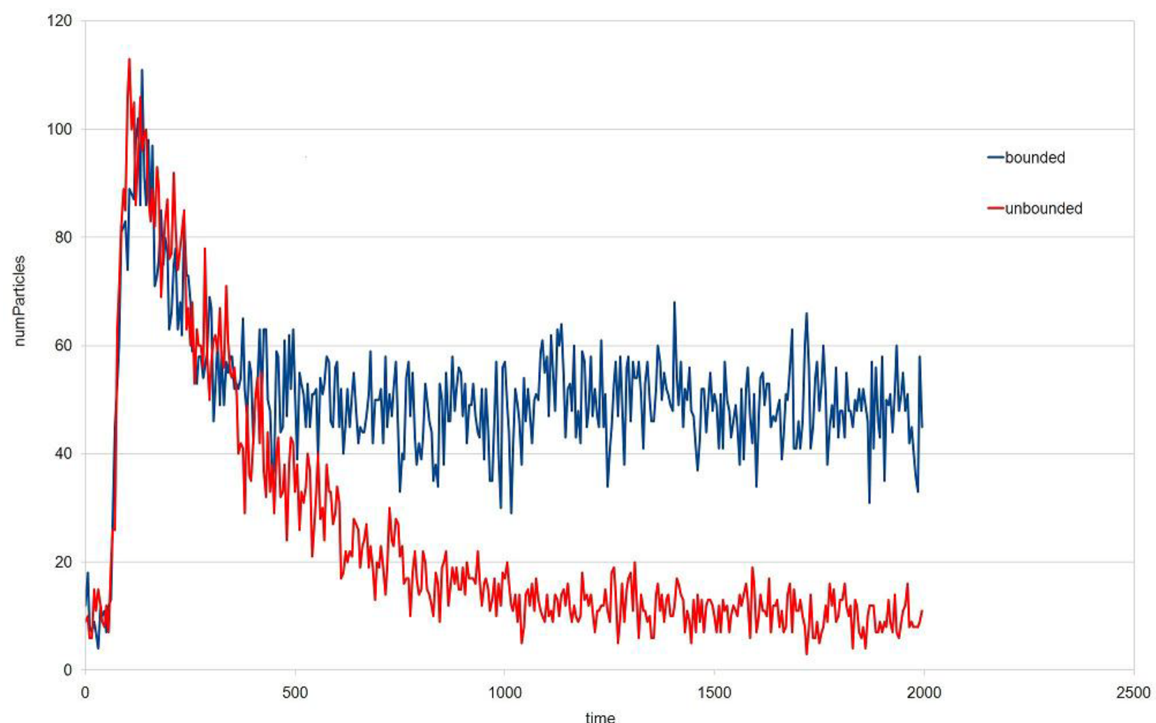


Figure 4. Comparison of a simulation with bounded and with unbounded space.

However, if the initial background concentration is set to a non-zero value, the simulation space must be bounded. In order to avoid this problem, N3Sim offers a mechanism to simulate an infinite space while using a bounded space. This mechanism deletes some particles close to the space bounds, as if they were leaving the space bounds. To activate this mechanism, the parameter **constantBgConcentration** must be set to *true*.

A few considerations to be taken into account in order to get the best results follow First, the space dimensions and the proper location of emitters and receivers is crucial. If, for instance, we want to run a simulation with one receiver and one emitter, the distance between the receiver and the space bounds, and between the emitter and the space bounds, must be at least twice the distance between the emitter and the receiver. Otherwise, the simulation of an infinite space will not be realistic.

Second, an important parameter used to adjust this mechanism is **constantBGConcentrationWidth**. As a rule of thumb, the value of this parameter must be the 2% of the minimum value between the parameters **xSize** and **ySize** (see section Simulation Space). If this value is increased, the tail of received signals will be artificially reduced; and if it is decreased, the tail will be increased. In any case, if the size of the space and the location of emitters and receivers have the values indicated in the section above, this parameter has very little influence and the simulation of infinite space is good enough for most cases.

EMITTERS

The location of an emitter is defined by the parameters **x** and **y** in the configuration file. Every emitter has an influence area, which means that the emitter can only release or absorb particles within this area. This area is a circle (or a sphere in 3 dimensions) and its radius is the value of the parameter **emitterRadius**.

If the emitter tries to emit/absorb more particles than the capacity of its influence area, N3Sim will emit/absorb as many as possible and a message will be written at the *error.log* file. This message will include the actual number of particles released/absorbed.

Each emitter has an associated waveform that defines the number of particles to be released/absorbed at each time step. Moreover, every emitter has a start time and end time (parameters **startTime** and **endTime**). The emitter will emit/absorb particles, according to its waveform, only when the simulation time is greater or equal than **startTime** and lower than **endTime**.

There are five types of emitters. Three of them have a predefined waveform, and they have a parameter named **amplitude**. The amplitude is the number of released particles every 100 ns. For instance, if the time step is of 400 ns and 1000 particles must be released at every time step, the parameter **amplitude** must be set to a value of 250.

The three emitters with a predefined waveform are:

- *Type 1*: emits a fixed number of particles at every time step.
- *Type 2*: emits particles following a rectangular waveform. For this emitter, two additional parameters exist: **period** and **timeOn**. These parameters indicate, respectively, the period of the square wave and amount of time within a period in which the emitter is releasing particles (as many as indicated by the parameter **amplitude**).
- *Type 3*: emits particles following a white noise waveform.

Type 4 emitters read the waveform of the signal to be transmitted from a text file. In this text file, every line represents a time step, beginning at **startTime**. At every time step, the number of particles indicated by the corresponding line are emitted. If the total number of lines of the text file is lower than the number of time steps between **startTime** and **endTime**, zeros are assumed. The parameter **file** indicates the filename where the waveform is defined (including relative or absolute path). Another parameter for this emitter is ***scaleFactor***. Emitters multiply the number of particles specified in the file

by this factor in order to obtain the number of particles to emit, which is useful to automate simulations.

Finally, *Type 5* emitters are equivalent to type 4, but in a 3-dimensional simulation space. For this emitter, the z location must be set using the parameter **z** in the configuration file.

RECEIVERS

The receive location is defined by the parameters **x** and **y** in the configuration file. Receivers count the number of particles in its area (or volume) at every time step. This number is written to the receiver result files (see section Output)

There are three types of receivers. First, *Type 1* receivers are square receivers, and their size is defined by the square side (parameter **side**). Second, *Type 2* receivers are circular, their size is defined by their radius (parameter **rradius**). Last, *Type 3* receivers (to be used in 3-dimensional simulations) are spherical receivers and the counting volume is also defined by its radius (parameter **rradius**).

Receivers may just count the particles inside their area (or volume), or they may also absorb these particles. This is controlled with the parameter **absorb**. Finally, the **accumulate** parameter controls whether the receiver output corresponds to the instantaneous particle count (**accumulate**=*false*), or to the accumulated particle count since the beginning of the simulation (**accumulate**=*true*).

ANNEX IV : LIST OF PARAMETERS

OVERVIEW

This document contains a short description of all the parameters used in the configuration file of N3Sim (version 0.7). For a more detailed explanation of parameters please read the User Guide.

The units of the parameters are nanometers (nm) and nanoseconds (ns), unless otherwise stated. Also the values of the parameters are assumed to be real numbers, unless specified.

In the configuration file, parameters are organized in four sections:

- Simulation parameters
- Space parameters
- Emitter parameters
- Receiver parameters

SIMULATION PARAMETERS

- **outPath**: name of the folder where the result files will be stored. Both absolute and relative (to the execution directory) paths may be used.
- **graphics** (true/false): if the value is *true*, N3Sim will create a visualization file to be used with NSVideo. NSVideo is still an unreleased prototype. Therefore, this parameter should be set to *false*.
- **infoFile** (true/false): if the value is *true*, N3Sim will create a file with information of the simulation at each time step, i. e., number of particles, number of collisions, time instants, etc. It may be used for debugging purposes.
- **activeCollision** (true/false): if set to *true* collisions among the emitted particles are modeled. Otherwise, the emitted particles are assumed to be transparent to each other and never collide.
- **BMFactor (0 to 1)**: the particle displacements due to Brownian motion are multiplied by this factor. The default value is 1.
- **inertiaFactor (0 to 1)**:controls the amount of inertia of the emitted particles. If set to zero, particles have no inertia. Otherwise, at each time step, the particle speed is equal to its speed in the previous time step multiplied by **inertiaFactor**.
- **time** (integer, ns): total time of the simulation.
- **timeStep** (integer, ns): duration of each time step.

SPACE PARAMETERS

- **boundedSpace** (true/false): if set to *false*, an unbounded space is simulated. Otherwise, a rectangular bounded space is simulated. The bottom left corner of the space has the coordinates (0,0) and the top right corner coordinates correspond to the values of the parameters **xSize** and **ySize**. If an initial background concentration is set (**bgConcentration** greater than zero), **boundedSpace** must be set to *true*.

- **constantBGConcentration** (true/false): if set to *false*, it has no effect on the simulation. If set to *true* (this can only be done when **boundedSpace** is also set to *true*), N3Sim will simulate an infinite space.
- **constantBGConcentrationWidth** (integer, nm): value used to adjust the simulation of infinite space when the space is bounded. Must be an integer between 0 and 10% of min(xSize, ySize). 5% is an acceptable value in most cases.
- **xSize** (nm, integer): horizontal size of the simulation space (see parameter **boundedSpace**).
- **ySize** (nm, integer): vertical size of the simulation space (see parameter **boundedSpace**).
- **D** (nm/ns^2): diffusion coefficient.
- **bgConcentration** (integer): number of particles per 10000 nm^2. The simulation space will be filled with this initial background concentration before the simulation starts. If **boundedSpace** is *false*, **bgConcentration** must be set to zero.
- **sphereRadius** (nm): radius of the emitted particles.

EMITTER PARAMETERS

- **emitters**: number of emitters in the simulation

Every emitter has the following parameters:

- **emitterRadius** (nm): radius of the influence area of the emitter. The influence area determines the area where particles are released/absorbed. If there are not enough particles to absorb, all the particles in the area will be absorbed and a warning message will be logged in the file *error.log*. Moreover, if the particles to be released do not fit within the influence area, as many as possible will be released and a message will also be logged in the file *error.log*.
- **x** (nm): horizontal position of the emitter.
- **y** (nm): vertical position of the emitter.
- **startTime** (ns): the emitter releases/absorbs particles when the current time is greater or equal than **startTime**.
- **endTime** (ns): the emitter releases/absorbs particles when the current time is less than **endTime**.
- **initV** (nm/ns): initial velocity of the released particles, in nm/ns. If the parameter **punctual** is set to false, the initial particle direction is the line from the emitter location to particle location. If **punctual** is set to *true*, then **initV** must be set to zero.
- **punctual** (true/false): if set to *true*, particles are released at exactly the emitter location. If set to *false*, particles are emitted in a random location within the influence area. If **activeCollision** is set to *true*, **punctual** must be set to *false*.
- **concentrationEmitter** (true/false): if set to *false,* the **amplitude** parameter (see below) indicates the number particles released by the emitter at every time step. If set to *true,* **amplitude** indicates the number particles present in the influence area of the emitter at every time step.
- **color**: color of the emitted particles. Used by the application NSVideo. Since NSVideo is still an unreleased prototype, the value *white* must always be used.
- **emitterType** (integer, 1 to 5): there are 5 types of emitters. Depending on the emitter type, some other parameters must be included in the configuration file. A short description of each emitter type and its additional parameters follows:
    - *Type 1*: emits a fixed number of particles at every time step. Additional parameters for type 1 include:
        - **amplitude**: number of released particles every 100 ns. E.g., if the time step is of 400 ns and 1000 particles must be released at every time step, **amplitude** must be set to 250.

    - *Type 2*: emits particles following rectangular waveform. Additional parameters for type 2 include:
        - **amplitude**: see type 1.
        - **period** (ns): wave period.
        - **timeOn** (ns): part of the wave period in which the particles are released (as many as the parameter **amplitude** indicates).
    - *Type 3*: emits particles following a white noise waveform. Additional parameters for type 3 include:
        - **amplitude**: see type 1.
    - *Type 4*: reads the waveform of the signal to be emitted from a text file. In this text file, every line represents a time step, beginning at **startTime**. At every time step, the number of particles indicated by the corresponding line are emitted. If the total number of lines of the text file is lower than the number of time steps between **startTime** and **endTime**, zeros are assumed. Additional parameters for type 4 include:
        - **file**: name of the text file (including relative or absolute path).
        - **scaleFactor**: the number of particles specified in the file is multiplied by this factor (useful to automate simulations).
    - *Type 5*: equivalent to type 4 emitter, but in a 3-dimensional simulation space. In addition to the parameters for type 4 emitters, the following parameter must be defined:
        - **z** (nm): z coordinate of the emitter location.

RECEIVER PARAMETERS

- **receivers**: number of receivers in the simulation

Every receiver has the following parameters:

- **name**: receiver name. The measurements of this receiver will be saved to a file named *name.csv*.
- **x** (nm): horizontal position of the receiver.
- **y** (nm): vertical position of the receiver.
- **absorb** (true/false): if set to *true*, after the receiver measures the particles in its influence area at every time step, they will be deleted.
- **accumulate** (true/false): if set to *true*, the output signal measured by the receiver at each time step corresponds to the accumulated number of particles measured from the simulation start. If set to *false*, it corresponds to the number of particles measured in the current time step.
- **receiverType**: (integer, 1 to 3): there are 3 types of receivers. Depending on the receiver type, some other parameters must be included in the configuration file. A short description of each receiver type and its additional parameters follows:
    - *Type 1*: the detection area of the receiver is a square. Additional parameters for type 1:
        - **side** (nm): side of the square that defines the detection area.
    - *Type 2*: the detection area of the receiver is a circle. Additional parameters for type 2:
        - **rradius** (nm): radius of the circle that defines the detection area.
    - *Type 3*: the detection volume of the receiver is a sphere (to be used only in 3-dimensional simulations). Additional parameters for type 3:
        - **z** (nm): z coordinate of the receiver location.
        - **rradius** (nm): radius of the sphere that defines the detection volume.