CX3D short tutorial

Version 0.5

Frédéric Zubler

August 7, 2009

Contents

1 Preliminaries **4**

1.1 Introduction 4

1.2 Four layers of abstraction 4

1.3 Imports 6

1.4 Particular non cellular classes 6

1.5 Triangulation requirement 7

2 Cell and cell modules **8**

2.1 First example : simple division 8

2.2 Writing a Cell Module : 10

3 Physical objects, cell elements and local biology modules **11**

3.1 Moving 11

3.2 LocalBiologyModule 12

3.3 Soma random walk 12

3.4 Branching 14

4 Substances **16**

4.1 Extracellular Substances 16

4.2 Artificial extracellular Substances 18

4.3 Intracellular Substances 20

4.4 Membrane Substances 22

4.5 Gene Substances 24

5 Synapses 25

6 Programming interface 27

6.1 ini.cx3d . . . 27

6.2 ini.cx3d.simulations . . . 28

6.3 ini.cx3d.physics . . . 28

6.4 ini.cx3d.localBiology . . . 30

6.5 ini.cx3d.cells . . . 31

THIS TUTORIAL IS UNDER CONSTRUCTION !

Installation :

If you plan to use CX3D for developing your own models, we strongly recommend that you use a Java Integrated Development Environment, such as Eclipse (http://www.eclipse.org). Below we just describe the steps to follow to run the example simulations provided in the original CX3D source code (in particular, the file `DividingCell.java`):

For Windows

Download the zip file from the webpage. Double click the downloaded file, and extract it into a folder. Press `Start, Run,` and enter `cmd`. A command line window will appear where you enter the two following lines:

```
cd «ExtractionDirectory»
java ini/cx3d/simulations/tutorial/DividingCell
```

(where `«ExtractionDirectory»` is the name of the directory where you put the extracted files).

For Mac OS X

Download the zip file from the webpage. A folder `ini` appears in your `Downloads` folder. You can move it in the folder of your choice, say `Documents`. Open a terminal (in `/Applications/Utilities/`). A window appears, in which you type the following lines:

```
cd /Documents
java ini/cx3d/simulations/tutorial/DividingCell
```

1 Preliminaries

1.1 Introduction

CX3D is a Java package for simulation of development of neural tissues. It allows cell division, migration, extension of neurites to form axonal and dendritic arbors, synapses formation, signaling molecules production and detection.

Beside the implementation of these biological processes, CX3D also takes into account the physical properties and interactions of the cells : neurons are decomposed into discrete elements, each one with a volume, an elasticity, a friction coefficient to simulate the mechanics of the cells. The program also allows diffusion of chemicals both in the extra-cellular matrix and inside the cells.

In this tutorial, we assume that the reader has some basic knowledge of Java.

1.2 Four layers of abstraction

We designed our software so that the user doesn't have to care about the technicalities of the implementation, and can concentrate on designing the growth rules he/she wants to simulate. Therefore, we have defined four abstract layers in the representation of a cell (Figure 1). The user interacts with the two upper layers by writing small modules describing the model's specification, he or she has to call some methods in the third layer, but doesn't interact with the fourth one at all. Each layer corresponds to a distinct java package, presented here from the most "biological" one to the most "utilitarian" one:

1. **ini.cx3d.cell**: in particular the class `Cell`, from which there is a unique instance per neuron. The user specifies biological properties concerning the entire neuron in small classes implementing the `CellModule` interface, that are inserted into the cell instances. Typically a cell cycle or a gene network would be coded at that level.

2. **ini.cx3d.localBiology**: the discrete elements composing a cell: `SomaElement` for the cell body, and a collection `NeuriteElement` for the neurites. Both are subclasses of the abstract `CellElement`. Local biological behavior like movement, branching, production or detection of a guidance cues are coded in classes implementing the `LocalBiologyModule`, which are inserted into specific cell elements.

3. **ini.cx3d.physics**: this third layer represents the physical properties of the cells like volume, friction, elasticity, etc. It also performs the simulation of the diffusion processes. Each soma element instance is associated with an instance of `PhysicalSphere`, and each neurite element with a `PhysicalCylinder`. They both derive from the abstract `PhysicalObject`, which in turn derives from `PhysicalNode`. This latter class represents a discrete volume of space, and the extracellular substances (instances of `Substance`)

it contains. To have the physical objects derive from the class representing pace, guaranties that each object created automatically comes with a definition of the space it is contained in.

4. **ini.cx3d.spatialOrganization**: Written by Dennis Goehlsdorf, the fourth and most technical level defines the boundaries between the physical nodes, as well as the neighboring relation between physical objects. It consists of a 3D kinetic and dynamic Delaunay triangulation [1].

There are other packages in CX3D: **ini.cx3d.simulation**, which contains two important classes : `Scheduler` and `ECM`, which are described below (Section 1.4), **ini.cx3d.graphics** contains the GUI, **ini.cx3d.synapses** the classes for making synaptic connections, **ini.cx3d.utilities.export** for saving an XML description of a grown network in the NeuroML level 3 scheme, **ini.cx3d.utilities**, especially with the class `Matrix`, containing useful static methods for vector and matrix operations.

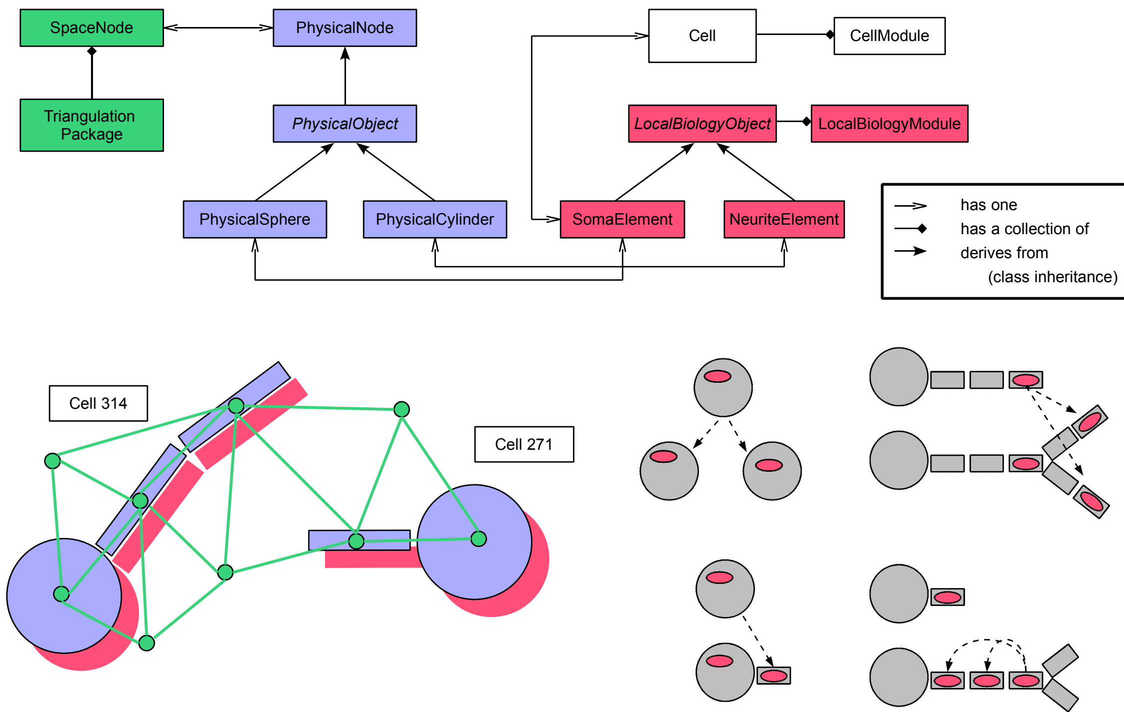

Figure 1: **The four layers of abstraction in our cell representation**: **(green)** The neighborhood definition : each physical object of the simulation contains a vertex. All vertices are linked in a graph structure that defines for each one of them, based on its localization, a subset of the other vertices that are considered as its neighbors. This definition is based on a Delaunay triangulation. **(blue)** The physical layer, that represents the space with the chemicals it contains and the methods for diffusion, as well as the mechanical properties of the cells. **(red)** the local biological properties of the neurons, associated with each physical objects **(white)** The higher level biological processes, relative to the entire cell. **(lower right)** possible automatic copy modes for local biology modules: when soma divide, when new neurites are being formed from the soma, when neurite branches, when neurite extend.

[1] *kinetic* means that the points can be moved, *dynamic* that they can be added or removed.

1.3 Imports

In the examples provided in this tutorial, we will always import only the classes that are explicitly needed, but it might be convenient to directly import the packages containing the classes describing the three upper layers of cell representation, the classes needed for running the simulation, the class containing several parameters and a utility class for vectorial operations:

```
import ini.cx3d.cells.*;
import ini.cx3d.localBiology.*;
import ini.cx3d.physics.*;

import ini.cx3d.simulations.*;
import ini.cx3d.Param;
import static ini.cx3d.utilities.Matrix.*;
```

1.4 Particular non cellular classes

Extracellular matrix

The class `ini.cx3d.simulation.ECM` is used when one wants to add additional points in the triangulation (see below), or register some chemicals or add some boundary conditions. It is useful to have a reference for it, and so many of your simulations will start with :

```
ECM ecm = ECM.getInstance();
```

Many processes in CX3D might use a pseudo random arguments (when you add noise for instance). It is nevertheless often useful to be able to reproduce exactely one simulation run. For this we can use the method `ECM.getRandomDouble()`. All CX3D classes use this method for their random processes. By specifying an initial seed, you ensure that several runs will give identical outcomes : `ECM.setRandomSeedTo(long seed)`;

Running the simulation

For actually running the simulation, we use the class `ini.cx3.simulation.Scheduler` that runs each object in the simulation (basically it calls the `run()` method that each CX3D objects contains). The scheduler contains two important static methods : `simulate()`, that runs the simulation indefinitely, and `simulateOneStep()`, that -big surprise- simulates only one time step. The latter is usually put inside a for loop, and is used if you want to have an external control on the simulation (see section 2.1).

Parameters

The class `ini.cx3d.Param` contains many parameters default value (like the default diameter of a neurite for instance), some `java.awt.Color` instances, with different degrees of transparency, and -probably the most

important- ,the time discretization scale of the simulation (`Param.SIMULATION_TIME_STEP`). By default it is set to 10^{-2} hours (36 seconds). Currently we don't dynamically change this time step.

Linear algebra

The class `ini.cx3d.utilities.Matrix` contains several `static` methods for performing operations on vectors (`double[]`) and matrices (`double[][]`). Unlike other java packages, you don't have to instantiate any matrix object. You directly call call one of the methods, giving the arrays as argument, and get a new array back. Be careful, there is no dimensions check! Of course, you don't have to use this class if you have your own habits. But it appears in the examples below.

1.5 Triangulation requirement

Although Dennis Goehlsdorf's triangulation package is extremely stable, and allows points to lie on a line, or on a plane (which is usually not the case for the Delaunay triangulation), you cannot *start* with points on a line or on a plane. It is therefore a good idea to first insert a couple of `PhysicalNode` at random position (something like 20). In case where you want to use diffusion, it is also necessary to have additional points, in this case much more are needed.

```
for (int i = 0; i < 18; i++) {  // eighteen extra PhysicalNodes
    ecm.getPhysicalNodeInstance(randomNoise(500,3));
}
```

Note :

We use the method `randomNoise(x,n)`, which gives an array of length n of `double` randomly chosen between -x and +x. It is our first use of the static methods of the class `ini.cx3d.utilities.Matrix`.

2 Cell and cell modules

2.1 First example : simple division

A cell contains four layers of objects, but luckily you don't have to declare them all and link them to one another. Instead you can use the convenient class `CellFactory`:

```
double[] cellOrigin = {0.0, 3.0, 5.0};                    // future cell's location
Cell cell = CellFactory.getCellInstance(cellOrigin);      // create the four layers
```

you can now access the local biology layer, and the physics layer with :

```
SomaElement soma = cell.getSomaElement();
PhysicalSphere sphere = soma.getPhysicalSphere();
```

We are now ready to design our first simulation. Suppose we are interested in cell division. Division is done at the cell level, and propagates down so that each layer is being copied. We end up with two cells, each one with a soma element, a physical sphere and a space node. The mother cell becomes one of the daughters, and the other daughter cell is returned by the method used for triggering division :

```
Cell c2 = cell.divide();
```

This method can also take arguments: a double, defining the volume ration between the two daughters (1.0 means equal size) and/or an array of 3 doubles specifying the division axis in global coordinates. (It is also possible to work with local spherical coordinates, see later).

The division conserves the total volume, so in order to make several cycles of divisions, you have to increase the volume after each division. Since it is a physical property, we have to call a method in the physical sphere. We could directly use the assessor `PhysicalObject.setVolume(.)`, but usually we want a smooth increase in volume, and we prefer to call one of the following methods:

```
sphere.changeDiameter(100);          // rate of 100 um/h
sphere.changeVolume(100);            // rate of 100 um^3/h
```

The model we propose is very simple, and it goes as follows: at each time step, we check the diameter of our cell. If it is above the threshold of 20 μm, we make it perform a division, otherwise it increases gradually it's volume. The class can be found in the file `ini.cx3d.simulations.tutorial.DividingCell`; it is also reproduced here:

```
package ini.cx3d.simulations.tutorial;

import ini.cx3d.Param;
import ini.cx3d.cells.Cell;
import ini.cx3d.cells.CellFactory;
import ini.cx3d.localBiology.SomaElement;
```

```
import ini.cx3d.physics.PhysicalSphere;
import ini.cx3d.simulations.Scheduler;

public class DividingCell {

    public static void main(String[] args) {

        double[] cellOrigin = {0.0, 3.0, 5.0};
        Cell cell = CellFactory.getCellInstance(cellOrigin);
        cell.setColorForAllPhysicalObjects(Param.RED);
        SomaElement soma = cell.getSomaElement();
        PhysicalSphere sphere = soma.getPhysicalSphere();

        for (int i = 0; i < 50000; i++) {                // at each time step,
            Scheduler.simulateOneStep();                 // run the simulation
            if(sphere.getDiameter()<20){                 // and check the diameter:
                sphere.changeVolume(350);                // .. increase...
            }else{
                Cell c2 = cell.divide();                 // .. or divide.
                c2.setColorForAllPhysicalObjects(Param.BLUE);
            }
        }
    }
}
```

Notes :

1) It is an asymmetrical division, since only one of the daughter cell continues to divide. To illustrate this we start with a red cell, and make sure that after division the non dividing daughter cell becomes blue.

2) `Cell.setColorForAllPhysicalObjects(.)` changes the color of all `PhysicalObjects` composing a cell, whereas `PhysicalObject.setColor(.)` changes the color of only one particular physical object. In this case, since the cell contains only one physical object (a `PhysicalSphere`), both methods would have the same effect.

3) We use colors difined in `ini.cx3d.Param`, but you can of course use any other standard `java.awt.Color`.

Exercises :

1) Modify the previous example so that every cell continues to divide, and not only one of the daughters.

2) Modify the previous example so that the cell division axis stays in the XY plane (in this case, you'll need to introduce extra `PhysicalNode` because the triangulation doesn't like to have all points on a plane..).

2.2 Writing a Cell Module :

In the previous example, we gave external instructions to the cells at each time step on how to behave. To let the cells act independently during the simulation, we usually design small modules that we incorporate into the cells. A cell module is a class that implements the `CellModule` interface, or that derives from the `AbstractCellModule` class.

The `CellModule` interface contains the following methods :

```
public Cell getCell()
        // returns the Cell this module belongs to
public void setCell(Cell cell)
        // sets the Cell this module belongs to
public void run()
        // where the simulation action is specified (this method is called at each time step).
public boolean isCopiedWhenCellDivision()
        // specifies if the second daughter cell receives a copy of this module after division.
public CellModule getCopy()
        // the module inserted in the second cell (if the previous method returns true).
```

A cell module is inserted or removed from a cell with the method `Cell.addCellModule(CellModule m)` and `Cell.removeCellModule(CellModule m)`.

Exercises :

1) Rewrite the simulation of section 2.1 with a `CellModule`.
(solution in `ini.cx3d.simulations.tutorial.DivisionModule`).

2) How can you change from symmetrical divisions (both daughter cells continue to divide) to asymmetrical divisions (only one daughter cell divides) by changing one single word in the file?

3) Modify the class to stop division (but not the simulation) after ten division cycles.

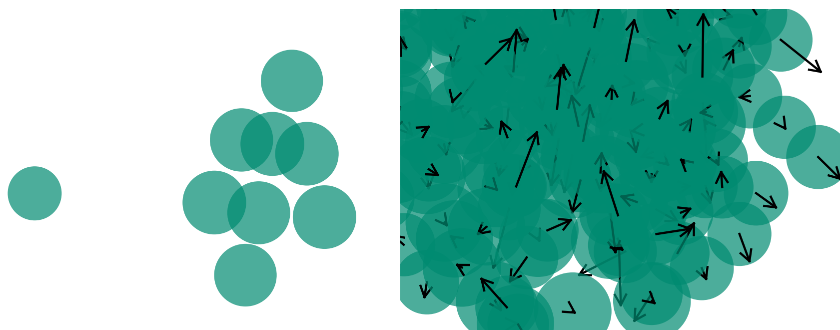

Figure 2: **ini.cx3d.simulations.tutorial.DivisionModule**: Starting with one cell, symmetrical cycles of divisions. The black arrows denote the total force acting on each physical sphere because of the high cell density.

3 Physical objects, cell elements and local biology modules

3.1 Moving

Each `PhysicalObject` contains a single point mass, defining its position. To move the object, you have to move its point mass. For that, you should NEVER use `PhysicalObject.setMassLocation(.)`, because it doesn't update the physical properties. Instead, you use the following method:

```
PhysicalObject.movePointMass(double speed, double[] direction)
```

The speed units are microns/hour; a displacement is computed depending on the time step. The norm (or magnitude) of the vector `direction` is not taken into account, but only its direction (if all components are zero, then there will be no movement).

There are two important things to remember for `PhysicalCylinder`. First that you can only move the terminal cylinder of a chain, and second that whether the displacement of its point mass results in a lengthening or a shortening of the cylinder, you don't change the tension (since it's internal spring is also lengthened or shortened), which corresponds to neurite elongation or retraction respectively, depending on the direction you use to move the point mass. To ensure elongation or retraction, you can use one of the following:

```
PhysicalCylinder.extendNeurite(double speed, double[] direction)
    // the movement is only performed if it results in an elongation
PhyicalCylinder.retractCylinder(speed)
    // obviously, you can't choose the direction.
```

If the retraction is longer than the neurite length, the neurite disappears (with the modules it may contain).

It is possible to call these methods from the local biology layer, through their counterpart:

```
CellElement.move(double speed, double[] direction)
NeuriteElement.elongateTerminalEnd(double speed, double[] direction)
```

```
NeuriteElement.retractTerminalEnd(double speed, double[] direction)
```

If a cylinder is to long, it will be split into two. Note that the terminal cylinder - and its neurite element- remains terminal. The new cylinder is inserted proximally. There is one exception: in case of bifurcation, two new cylinders are added distally to the previously terminal one.

3.2 LocalBiologyModule

Cell elements, namely `SomaElement` and `NeuriteElement` can also contain modules, dictating their behavior during simulation. Such a module is a class implementing the `LocalBiologyModule` interface or extending the `AbstractBiologyModule` class. A local biology module is added or removed with the methods `CellElement.addLocalBiologyModule(.)` and `CellElement.removeLocalBiologyModule(.)`.

The `LocalBiologyModule` interface contains the following methods :

```
public CellElement getCellElement()
public void setCellElement(CellElement cellElement)

public void run()
    //where the simulation  action is specified (this method is called at each time step).
public boolean isCopiedWhenNeuriteBranches();
    // specifies if the module is copied into new branches
public boolean isCopiedWhenSomaDivides();
    // specifies if the module is copied when the soma divides
public boolean isCopiedWhenNeuriteElongates();
    // specifies if the module is copied in each segment of a neurite
public boolean isCopiedWhenNeuriteExtendsFromSoma();
    // specifies if the module is copied in new neurites.
public CellModule getCopy()
    // the module inserted in another element when one of the previous method returns true.
public boolean isDeletedAfterNeuriteHasBifurcated()
    // specifies if deleted when in a terminal neurite element that bifurcates
```

3.3 Soma random walk

Now we are ready to use what we have just learned to design a module for somata to perform a smoothed random walk (at each time step the *change* of direction is random, and not the direction itself):

```
package ini.cx3d.simulations.tutorial;

import static ini.cx3d.utilities.Matrix.add;
import static ini.cx3d.utilities.Matrix.normalize;
import static ini.cx3d.utilities.Matrix.randomNoise;
import ini.cx3d.cells.Cell;
import ini.cx3d.cells.CellFactory;
import ini.cx3d.localBiology.AbstractLocalBiologyModule;
import ini.cx3d.simulations.Scheduler;
```

```java
public class SomaRandomWalkModule extends AbstractLocalBiologyModule {

    double[] direction = randomNoise(1.0, 3); // initial direction

    public AbstractLocalBiologyModule getCopy() {
        return new SomaRandomWalkModule();
    }

    public void run() {
        double speed = 50;
        double[] deltaDirection = randomNoise(0.1, 3);
        direction = normalize(direction);
        direction = add(direction, deltaDirection);
        super.cellElement.move(speed, direction);
    }

    public static void main(String[] args) {
        for(int i = 0; i<5; i++){
            Cell c = CellFactory.getCellInstance(randomNoise(40, 3));
            c.getSomaElement().addLocalBiologyModule(new SomaRandomWalkModule());
        }
        Scheduler.simulate();
    }
}
```

Notes :

1) This class extends `AbstractLocalBiologyModule`, which contains a field `cellElement` that is declared as protected, so that the subclasses can access it.

2) For the random walk, we consider a 3d vector direction to which we add some random noise each time step, to slightly change the direction in which the soma moves at the constant speed of 50 microns/hours. (If you were to change the simulation time stape Δt, you should scale the amount of noise with $\sqrt{\Delta t}$).

3) This class uses two new static methods of `Matrix`: `add(.)` for vectorial addition and `normalize(.)` for normalizing a vector (setting the length to 1.0).

4) In the main method, we create 5 cells with this module. You can change this by taking any other number, but if you choose to have exactly 4 cells, you have to add at least one `PhysicalNode` instance because the triangulation doesn't allow 4 vertices to move (1, 2, 3, 5, 6, ... are all ok, but not 4!). You can turn on the Delaunay display, it makes beautiful geometrical figures !!

Exercises :

1) Make the moving cells divide (use the `DivisionModule` class of section 2.2). Note: if you want both daughter cells to move, i.e. to contain an instance of `SomaRandomWalkModule`, you have to overwrite the `isCopiedWhenSomaDivides()` method, making it return `true`.

3.4 Branching

New neurites (chains of `PhysicalCylinder` instances with their associated `NeuriteElements`) can be created from a `SomaElement` or from a `NeuriteElement`. Initially, there is only one element in the chain.

From a soma, we use the method `SomaElement.extendNewNeurite()`, which can also take as arguments an array of double (specifying the direction where the new neurite points to) and/or a double specifying the diameter of the new cylinder. This method returns the new neurite element.

From a neurite element, we have two possibilities: to make a *bifurcation* or a *side branche*. `NeuriteElement.bifurcate()` adds two new neurite elements at the tip of a distal neurite element (if it is not terminal, a runtime exception is printed to the standard error output). The method can take arguments to specify the directions and diameters of the new branches. By default, there is a 60° angle between the two daughter branches. `NeuriteElement.branch()` extends a side branch starting from the center of a neurite element (that doesn't have to be terminal), which is thus split into two.

We illustrate these methods in the file `ini.cx3d.simulations.NeuriteBranchingModule`, a local biology module that produces an arbor, with bifurcation segments in red and blue, and side-branche segments in violet:

The run method contains the following code:

```
public void run() {
    //some parameters
    double speed = 100;
    double probabilityToBifurcate = 0.005;
    double probabilityToBranch = 0.005;
    // same movement principle as in the previous example
    double[] deltaDirection = randomNoise(0.1, 3);
    direction = add(direction, deltaDirection);
    direction = normalize(direction);
    neuriteElement.getPhysical().movePointMass(speed, direction);
    // evaluate bifurcation
    if(Math.random()<probabilityToBifurcate){
        NeuriteElement[] nn = neuriteElement.bifurcate();
        nn[0].getPhysical().setColor(Color.red);
        nn[1].getPhysical().setColor(Color.blue);
        return;
```

```
    } // evaluate branching
    if(Math.random()<probabilityToBranch){
        NeuriteElement n = neuriteElement.branch();
        n.getPhysical().setColor(Color.green);
        return;
    }
}
```

`direction` is a filed like in the previous example, except that it not initially set to a random value. It is set to point in the direction of the `PhysicalCylinder` associated with the `NeuriteElement` the module is set into:

```
public void setCellElement(CellElement cellElement) {
    if(cellElement.isANeuriteElement()){
        neuriteElement = (NeuriteElement)cellElement;
        // to start the elongation in the direction of the cylinder :
        direction = neuriteElement.getPhysicalCylinder().getAxis();
    }else{
        cellElement.removeLocalBiologyModule(this);
        System.out.println("Sorry, I only work with neurite elements");
    }
}
```

To have the module copied in the first neurite element of the new branches created after bifurcation or side-branching, we make the method `isCopiedWhenNeuriteBranches()` return true. To avoid having the module persisting in an element that has bifurcated (and hence that is no longer terminal), we make `isDeletedAfterNeuriteHasBifurcated()` return true as well.

The simulation is started with one cell, which we make extend a neurite, in which we insert the module:

```
public static void main(String[] args) {
    for (int i = 0; i < 18; i++) {
        ECM.getInstance().getPhysicalNodeInstance(randomNoise(1000,3));
    }
    for(int i = 0; i<1; i++){
        Cell c = CellFactory.getCellInstance(randomNoise(40, 3));
        NeuriteElement neurite = c.getSomaElement().extendNewNeurite();
        neurite.getPhysicalCylinder().setDiameter(2);
        neurite.addLocalBiologyModule(new NeuriteBranchingModule());
    }
    Scheduler.simulate();
}
```

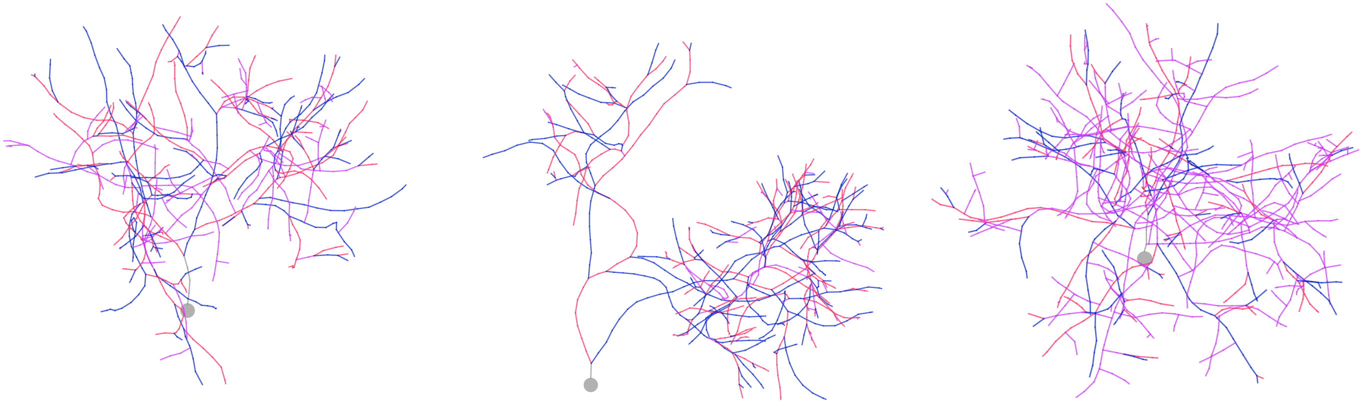

Figure 3: **ini.cx3d.simulations.tutorial.NeuriteBranchingModule**: Arborization of a single neurite. Segments issued from a side branch are violet, segments from the left and right daughter branch at a bifurcation are red and blue respectively. **(left)** Probability to bifurcate (P_b) equal to probability to make a side branch (P_s). **(midlle)** $P_b > P_s$. **(right)** $P_s > P_b$.

4 Substances

There are three different kinds of substances in CX3D:

1) Extracellular substances, that can diffuse in the extracellular space. They are represented by instances of the class `ini.cx3d.physics.Substance`, and stored in the `PhysicalNode` instances.

2) Intracellular substances and membrane-bound substances, that are confined to cell structures. For this we use instances of `ini.cx3d.physics.IntracellularSubstance`, which are stored in instances of `PhysicalObject`.

3) Gene substances, representing protein expression of genes. They are represented by instances of `ini.cx3d.cell.GeneSubst` and stored in instances of `Cell`.

4.1 Extracellular Substances

Extracellular substances are represented by the class `Substance`, who's most important fields are the `id`, the `diffusionConstant` and the `degradationConstant`. They are stored in the `PhysicalNodes`, that take care of diffusion and degradation. Cell elements and local biology modules never directly interact with the substances, but rather with the `PhysicalNode` they are associated to[2], through the following methods :

```
public double getExtracellularConcentration(String id)
```

[2]Recall that `PhysicalObject` extends `PhysicalNode`

```
// Returns the concentration of an extracellular Substance at this PhysicalNode.

public double getConvolvedConcentration(String id)
// Returns the weighted average concentration of a Substance at this PhysicalNode
// and all its neighbors, weighted with respect to the volumes of all nodes
//(attenuates the fluctuations due to the irregular triangulation,
// but more expensive to compute; usually not needed).

public double getExtracellularConcentration(String id, double[] location)
// Returns the concentration of an extra-cellular Substance
//(obtained by interpolation between the PhysicalNodes surrounding this point)

public double[] getExtracellularGradient(String id)
// Returns the gradient at the PhsicalNode's location

public void modifyExtracellularQuantity(String id, double quantityPerTime)
// Modifies the quantity (increases or decreases) of a Substance at this PhysicalNode's location
```

At the begining of a simulation that involves extracellular substances, you declare them and register them to ECM (it is not mandatory, but if you do so, you'll be able to visualize there concentration at each physical node, and you can specify the diffusion and degradation constants). For instance:

```
Substance netrin = new Substance("Netrin",1000, 0.01); // id, diffusion cst, degrad cst
ECM.getInstance().addNewSubstanceTemplate(netrin);
```

Imagine that we want to model cell aggregation. We have each soma release a certain chemical cue, and in the mean time move up the gradient of concentration of the same chemical. The run method of the `LocalBiologyModule` is something like :

```
public void run() {
    PhysicalObject physical = super.cellElement.getPhysical();
    // move
    double speed = 100;
    double[] grad = physical.getExtracellularGradient(substanceID);
    physical.movePointMass(speed, normalize(grad));
    // secrete
    physical.modifyExtracellularQuantity(substanceID, 1000);
}
```

<u>**Exercises :**</u>

1) Write a simulation with two different types of cells, secreting a cell type specific substance, that aggregate. (Solution: `ini.cx3d.simulations.tutorial.SomaClustering`)

2) With the run method proposed above, somata move all the time. Modify the method so that they only move if the gradient stays in the same direction for a while.

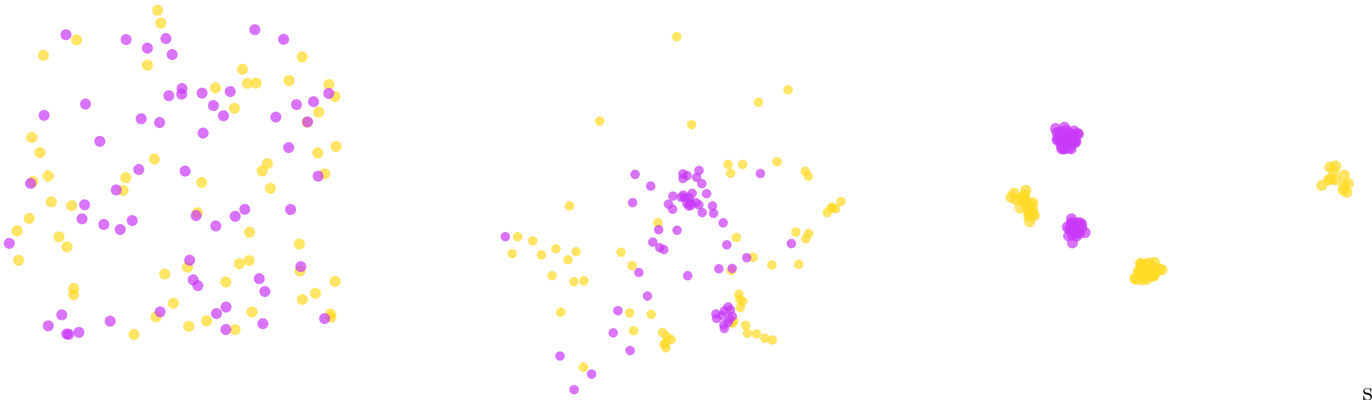

Figure 4: **ini.cx3d.simulations.tutorial.SomaClustering**: 60 yellow cells secreting a "yellow" diffusible substance and 60 violet cells secreting a "violet" diffusible substance follow the gradient of concentration of their respective substance, which leads to cell clustering.

4.2 Artificial extracellular Substances

Diffusion is computationally very expensive. For testing, when you do several simulations, it is possible to define fixed concentration profiles, linear or gaussian, along the X or the Z axis. To get the concentrations and the gradients of theses "artificial" substances you use the same methods in `PhysicalNode` than for real Substances.

ECM's method for adding artificial substances :

```
public void addArtificialGaussianLayerZ(
        Substance substance, double maxConcentration, double zCoord, double sigma)
// defines a gaussian concentration of a Substance along the Z axis (ie uniform along X,Y axis)

public void addArtificialLinearGradientZ(
        Substance substance, double maxConcentration, double zCoordMax, double zCoordMin)
// Defines a linear artificial concentration between two points along the Z axis.
// Outside this interval the value will be 0. Between the boundaries the value is the linear
// interpolation between the maximum value and 0.

public void addArtificialGaussianLayerX(
        Substance substance, double maxConcentration, double xCoord, double sigma)

public void addArtificialLinearGradientX(
        Substance substance, double maxConcentration, double xCoordMax, double xCoordMin)
```

We demonstrate the functioning of these artificial substances by using them as guidance cues and branching factor in the extension of a neurite arbor (demo in the file `ini.cx3d.simulations.NeuriteChemoAttraction`).

In the `main()` methods, we define an artificial concentration of a particular substance `A`, with a gaussian profile along the Z axis. In this case we use a different constructor for `Substance`; indeed the concentration is fixed by the gaussian function, and so the diffusion and degradation constant are not taken into account. Instead we

define a color for the visualization:

```
ECM ecm = ECM.getInstance();
Substance attractant = new Substance("A", Color.red);
ecm.addArtificialGaussianLayerZ(attractant, 1.0, 400.0, 160.0);
```

We then instantiate a cell, the soma of which extends a neurite like we have described previously. In this neurite, we insert a module that chooses the gradient as growth direction, and as branching signal. The `run()` method goes like this:

```
public void run() {
    PhysicalObject physical = super.cellElement.getPhysical();
    double concentration = physical.getExtracellularConcentration(substanceID);
    double[] grad = physical.getExtracellularGradient(substanceID);

    // some parameters
    double concentrationThreshold = 0.3;
    double bifurcationCoefficient = 0.005;
    double oldDirectionnWeight = 1.0;
    double gradientWeight = 0.2;
    double randomnessWeight = 0.6;

    // 1) movement
    if(physical.getExtracellularConcentration(substanceID)> concentrationThreshold)
        grad = new double[] {0.0, 0.0, 0.0};

    double[] newStepDirection = add(
        scalarMult(oldDirectionWeight, direction),
        scalarMult(gradientWeight, normalize(grad)),
        randomNoise(randomnessWeight,3));
    double speed = 100;
    physical.movePointMass(speed, newStepDirection);

    direction = normalize(add(scalarMult(5,direction),newStepDirection));

    // 2) branching based on concentration:
    if(ecm.getRandomDouble()<concentration* bifurcationCoefficient){
        ((NeuriteElement)cellElement).bifurcate();
    }
}
```

Note :

1) The method elongates the neurite by moving according to the gradient. But instead of taking directly the gradient as movement direction, we make a weighted sum combining the previous movement steps, some random perturbation and the gradient. You can change the different weights to see the effects on the branching pattern.

2) For the neurites not to stay absolutely at the peak of concentration, we disable the gradient detection (we set it to zero) in a region around the peak (i.e. if the concentration is higher than a defined threshold).

Exercises :

1) Define a vertical ("X") and and horizontal ("Z") artificial gaussian gradient, and modify the class to attract the neurite at the intersection of the two substances, and ramify there.

Figure 5: **ini.cx3d.simulations.tutorial.NeuriteChemoAttraction**: The extracellular space is pre-patterned with an artificial concentration profile of a substance, which serves as guidance cue and branching factor. **(left)** Because there growth direction is always influenced by the concentration gradient, the branches tend to stay in region with the highest concentration. **(right)** If we change the rules, so that above a certain concentration threshold the growing branches are no longer influenced by the gradient, we see that the arbor is more spread.

4.3 Intracellular Substances

`PhysicalObject` can contain a sub class of `Substance` : `IntracellularSubstance`, that is used for representing intracellular and membrane bound substances. This class has two additional fields:

1) `visibleFromOutside`: if `true`, the substance presence can be detected by neighboring objects, and represents thus a membrane substance; if `false`, it is a purely intracellular substance.

2) `volumeDependant`: when an intracellular substance is secreted at a certain rate (quantity per time), this has to be transformed into an intracellular concentration. If this field is `true`, the real volume of the sphere or cylinder is used, which can be useful to measure a total cell volume for instance. If `false`, we use the length of the physical object and not its volume. It avoids for instance that too much substance is trapped into the soma.

As for its supercalss, a `IntracellularSubstance` has to be declared to ECM, and then is produced and can

be probed from the following methods (note: there is no intracellular or membrane gradient!) :

```
PhysicalObject.getIntracellularConcentration(String substanceId)
PhysicalObject.modifyIntracellularQuantity(String id, double quantityPerTime)

PhysicalObject.getMembraneConcentration(String substanceId)
PhysicalObject.modifyMembraneQuantity(String id, double quantityPerTime)
```

Intracellular substances diffuse automatically inside the cells along the cylinders chain (unless the diffusion constant is set to 0.0)

As illustration, we present a model of production-consuption. Some necessary substance is secreted intracellularly by the soma, and diffuses along the chain of physical objects. The tip of each branch elongates (and consumes) at a speed proportional to the concentration of the substance in its terminal segment. In honor of Dr vanOoyen we call the substance T, for tubulin.

main method :

```
// defining the templates for the intracellular substance
double D = 1000;        // diffusion cst
double d = 0.01;        // degradation cst
IntracellularSubstance tubulin = new IntracellularSubstance("T",D,d);
tubulin.setVolumeDependant(false);
ecm.addNewIntracellularSubstanceTemplate(tubulin);
// getting a cell
Cell c = CellFactory.getCellInstance(new double[] {0,0,0});
c.setColorForAllPhysicalObjects(Param.RED);
// insert production module
SomaElement soma = c.getSomaElement();
soma.addLocalBiologyModule(new InternalSecretor("T", secretionRate));
//insert growth cone module
NeuriteElement ne = c.getSomaElement().extendNewNeurite(new double[] {0,0,1});
ne.getPhysical().setDiameter(1.0);
ne.addLocalBiologyModule(new GrowthCone());
// simulate it
Scheduler.simulate();
```

run method of the `InternalSecretor` class, the secretion module :

```
super.cellElement.getPhysical().modifyIntracellularQuantity(
                                        substanceId, secretionRate);
```

run method of the `GrowthCone` class, the growth module :

```
double speedFactor = 500;
double consumptionFactor = 100;
double bifurcationProba = 0.003;
double min = 100; // minimal concentration for movement

PhysicalCylinder cyl = (PhysicalCylinder)(super.cellElement.getPhysical());
double concentration = cyl.getIntracellularConcentration("T");
```

```
if(concentration<0.07)
    return;

double speed = concentration*speedFactor;
cyl.movePointMass(speed, add(cyl.getAxis(), randomNoise(0.5,3)));
cyl.modifyIntracellularQuantity("T", -concentration*consumptionFactor);

if(ECM.getRandomDouble()<bifurcationProba)
    ((NeuriteElement)(super.cellElement)).bifurcate();
```

Note :

There is a competition between several growth cones, and so the more branches there are, the slower they go. But the occurrence of bifurcations per time per growth cone is constant. The branches become then shorter.

Exercises :

1) Make the branching probability also depend on the concentration of T. Attribute different "cost" to branching, and look at the difference in arborization.

2) Make the branching dependent on a second substance secreted in the soma.

3) Design a module for active transport of intracellular substances. (Note: If you want to have a copy of that module in every `NeuriteElement`, you need both `isCopiedWhenNeuriteElongates()` and `isCopiedWhenNeuriteBranches()` to return `true`).

4.4 Membrane Substances

For a membrane-bound substances, we also use instances of the `IntracellularSubstance` class, but we have to make them detectable from other physical objects than the one they stays in. For this we use the the method `IntracellularSubstance.setVisibleFromOutside(.)`. These substances are then secreted and detected in physical objects with the same methods than the ones described above (for purely intracellular substances).

To illustrate the use membrane substances we write a module `ini.cx3d.simulations.tutorial.MembraneContact` that checks for the presence of a membrane marker called "A" on neighboring objects in close contact. If some "A" is detected, all modules are removed from the cell element containing this detector.

```
public void run() {
    PhysicalObject physical = super.cellElement.getPhysical();
```

```
    for (PhysicalObject o: physical.getPhysicalObjectsInContact()) {
        if(o.getMembraneConcentration("A")>1){
            physical.setColor(Param.YELLOW);
            super.cellElement.cleanAllLocalBiologyModules();
        }
    }
}
```

Note :

1) The method `iPhysicalObject.getPhysicalObjectsInContact()` returns all the instances of physical objects that are neighbors in the triangulation and that are close enough to be considered as in contact.

2) As you can see, `PhysicalObject.getMembraneConcentration(.)` returns the concentration of a membrane bound chemical, i.e. an intracellular substance that is visible from outside.

3) All modules can be removed at once with `CellElement.cleanAllLocalBiologyModules()`. In our example the cell element also contains a movement module, it is also removed, and the movement stops.

In the main method we declare the substance, set some physical boundaries, and create two types of cells: some immobile ones that express "A" at their surface, and some mobile with the contact detector:

```
public static void main(String[] args) {
    ECM ecm = ECM.getInstance();

    IntracellularSubstance adherence = new IntracellularSubstance("A",0,0);
    adherence.setVisibleFromOutside(true);
    adherence.setVolumeDependant(false);
    ecm.addNewIntracellularSubstanceTemplate(adherence);

    ecm.setArtificialWallsForSpheres(true);
    ecm.setBoundaries(-150, 150, -150, 150, -100, 100);

    for(int i = 0; i<10; i++){
        Cell c = CellFactory.getCellInstance(randomNoise(100, 3));
        c.setColorForAllPhysicalObjects(Param.RED);
        c.getSomaElement().getPhysical().modifyMembraneQuantity("A", 100000);
        c.getSomaElement().getPhysicalSphere().setInterObjectForceCoefficient(0.0);
    }
    for(int i = 0; i<10; i++){
        Cell c = CellFactory.getCellInstance(randomNoise(50, 3));
        c.getSomaElement().addLocalBiologyModule(new MembraneContact());
        c.getSomaElement().addLocalBiologyModule(new SomaRandomWalkModule());
        c.setColorForAllPhysicalObjects(Param.VIOLET);
    }
    Scheduler.simulate();
}
```

Note :

1) The method `IntracellularSubstance.setVisibleFromOutside(true)` is essential for making a membrane substance, but otherwise the procedure is the same than for purely intracellular substance.

2) `ecm.setArtificialWallsForSpheres(true)` puts some limitation on where a `PhysicalSphere can move`. It forces the wandering cells to stay within these limits. You set the size of the permitted volume with `ecm.setArtificialWallsBoundaries()`. that takes as argument the minimal and maximum value allowed along the x, y, and z axis.

3) In this example we are not interested in the dynamics of substance production, we just want the substance to be present on some spheres. Therefore we secrete once a (very) large amount with `PhysicalObject.modifyMembraneQuantity("A", 100000)`. After 1 time step, the quantity is 10^5 [quantity/time] * 10^{-2} [time] = 10^3 in each cell.

4.5 Gene Substances

`Cell` instances can also contain some specific substances. They are supposed to represent the proteins expressed from a gene, and therefore are termed `GeneSubstance`, a class designed by Sabina Pfister that also derives from `Substance`. I won't describe the specificities of these substances here.

5 Synapses

Biological synapses are composed of a pre-synaptic and a post-synaptic part. In cortical neurons these usually consist of a *bouton* (on an axon) and a *spine* (on a dendrite). In CX3D, synapses also consist of a spine and a bouton, each with a physical and a biological part (like for the PhysicalObject and the CellElement). All classes described in this section are in the package `ini.cx3d.synapse`.

Physical part

For the physical part of synapses we use the classes `PhysicalSpine` and `PhysicalBouton` (that both extend the abstract `Excrescence`). Note that these classes do not derive from `PhysicalObject`, and so they don't contain a node of the Delaunay triangulation, and they don't exert any force. Instead, they are attached to a `PhysicalSphere` or a `PhysicalCylinder`, to which they can be added with the method `PhysicalObject.addExcrescence(.)` One end of an excrescence is fixed on the physical object it belongs to, and follows it in case of displacement. The other end of an excrescence is either free, or linked to another excrescence, in which case they define a synapse.

The position on the physical object and the orientation of an excressence is described in polar coordinates, which we'll discuss in the next section.

Biological part

Each `PhysicalSpine` is linked to a `BiologicalSpine`, and each `PhysicalBouton` to a `BiologicalBouton`. Spines should be only on dendritic neurite elements, and boutons on axonal neurite element. To define if a neurite element is part of a dendrite or of an axon, we use the following method: `NeuriteElement.setIsAnAxon(.)` with a boolean argument.

We consider two types of cells in CX3D: excitatory and inhibitory. A synapse in which the presynaptic cell is excitatory (resp. inhibitory), is an excitatory (inhibitory) synapse. The type of the cell is set with the method `Cell.setNeuroMLType()` that takes as argument `Cell.ExcitatoryCell` or `Cell.InhibitoryCell`.

Network export

At any time during the simulation, it is possible to parse all cell elements, and to collect informations about all the synaptic connections. A file summarizing the connectivity of the grown network can be exported in an XML document, following the NeuroML specification. It can then be used to start a simulation in a point neuron simulator like PCSIM.

Single synapse Formation

To make a synapse between two neurites, one has to 1) instantiate a pair physical spine - biological spine and a pair physical bouton - biological bouton on the physical cylinders of two adjacent neurites, and 2) link them with : `Excrescence.synapseWith(Excrescence e, boolean createPhysicalBond)`. If `createPhysicalBond` is `true`, a spring is added to fix the distance between the two `PhysicalCylinders` supporting the bouton and the spine. Physical bonds are discussed in the next section.

The class `ini.cx3d.simulations.tutorial.SimpleSynapse` demonstrates the steps for the formation of a single synapse.

Automatized synapse formation

Coding explicitly each synapse formation is very tedious. Instead, you can use the following methods of `NeuriteElement`:

```
public void makeSpines(double interval)
//  Makes spines (the physical and the biological part) on this NeuriteElement,
//  with an average specified interval.

public void makeBoutons(double interval)
//  Make boutons (the physical and the biological part) on this NeuriteElement.

public int synapseBetweenExistingBS(double probabilityToSynapse){
//  Links the free boutons of this neurite element to adjacents free spines
```

Or you can make spines and connections in one command with this static method of the class `ini.cx3d.synapses.ConnectionsMaker`:

```
extendExcressencesAndSynapseOnEveryNeuriteElement().
```

<u>**Exercise :**</u>

Create 4 excitatory cells, 4 inhibitory cells, each one extending one axonal and one dendritic tree, and create synapse between them. (One example is in `tutorial.SmallNetwork`).

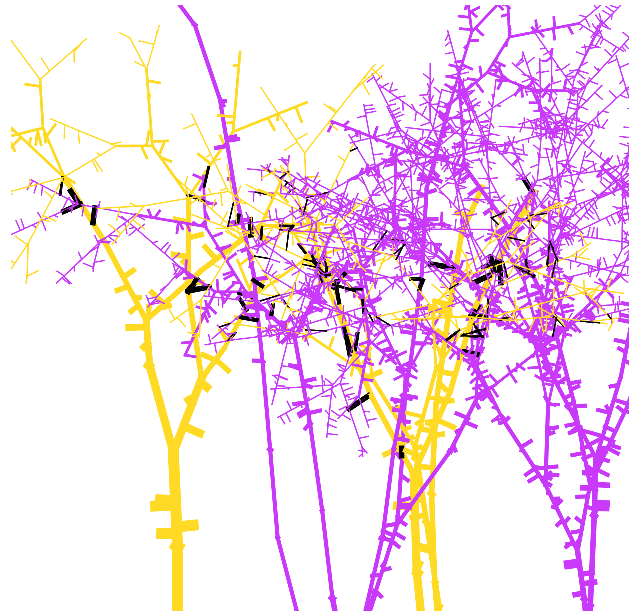

Figure 6: **Spines, boutons and synapses**. The figure shows the branches made by two "cortical" neurons in a 3D environment. Each neuron has an axon (violet) and an apical dendrite (yellow). The neurites extend some spines and boutons, in random direction. When a bouton and a spine have their free extremity within a small distance, a synapse is formed. To ensure that two branches sharing a synapse stay close one to another, a physical link is added between them (black), at the synapse location.

6 Programming interface

6.1 ini.cx3d

Param

Contant	Value	Description
`SIMULATION_TIME_STEP`	0.01 [h]	Δt
`SIMULATION_MAXIMAL_DISPLACEMENT`	3 [μm]	maximum possible physical displacement during one time step (needed for stability reasons, should not be changed).
`NEURITE_MAX_LENGTH`	15 [μm]	maximum length of a cylinder before it is automatically spit into two.
`NEURITE_MIN_LENGTH`	12 [μm]	shortest length for a cylinder before it is fused with its mother proximal cylinder.
`NEURITE_DEFAULT_LENGTH`	1 [μm]	
`NEURITE_DEFAULT_DIAMETER`	1 [μm]	
`NEURITE_DEFAULT_ADHERENCE`	0.1	static friction.
`NEURITE_DEFAULT_MASS`	1	kinetic friction.
`NEURITE_DEFAULT_SPRING_CONSTANT`	10.0	
`NEURITE_DEFAULT_TENSION`	0.0	
`SOMA_DEFAULT_DIAMETER`	1 [μm]	
`SOMA_DEFAULT_ADHERENCE`	0.4	static friction.
`SOMA_DEFAULT_MASS`	1	kinetic friction.

6.2 ini.cx3d.simulations

ECM

`getECMtime()`	returns the time of the simulation

`setRandomSeed(double seed)`
`getRandomDouble()`
`getGaussianDouble(double m, double sd)`
	initializes the random number generator
	returns a number between 0 and 1
	retruns nb from gaussian distribution

`setBoundaries(double x1, x2, y1, y2, z1, z2)`
 	
`setArtificialWallsForSpheres(boolean b)`
 	
`setArtificialWallsForCylinders(boolean b)`
	Sets the boundaries of a pseudo wall,
	that maintains the PhysicalObjects in a closed volume
	If set to true, the PhysicalSpheres tend to stay inside
	a box, who's boundaries are set with setBoundaries()
	If set to true, the PhysicalCylinders tend to stay inside
	a box, who's boundaries are set with setBoundaries()

`getPhysicalNodeInstance(double[] loc)`	Adds a "simple" PhysicalNode (with its SON) at a desired location

`addNewSubstanceTemplate(Substance s)`
`addNewIntracellularSubstanceTemplate(`
` IntracellularSubstance s)`
	Registers an extra-cellular substance
	Registers an intra-cellular substance

`addArtificialGaussianConcentrationZ(`
` String name, double maxConc, mean, sigma)`
`addArtificialLinearConcentrationZ(`
` String name, double maxConc, coordMax, coordMin)`
(*idem for X-axis*)
	Defines a bell-shaped artificial concentration in ECM,
	along the Z axis
	Defines a linear artificial concentration in ECM,
	between two points along the Z axis

6.3 ini.cx3d.physics

PhysicalNode

`isAPhysicalObject()`
`isAPhysicalSphere()`
`isAPhysicalCylinder()`
	returns `true` if it is a physical sphere or a physical cylinder.
	returns `true` if it is a physical sphere.
	returns `true` if it is a physical cylinder.

`getExtracellularConcentration(String id)`
`getConvolvedConcentration(String id)`
`getExtracellularConcentration(`
` String id, double[] location)`
`getExtracellularGradient(String id)`
	concentration at physical node's center
	average concentration over node and neighbors
	interpolated concentration
	
	gradient at node's center

`modifyExtracellularQuantity(`
` String id, double rate)`
`Substance getSubstanceInstance(Substance s)`
	adds `rate`$\cdot\Delta t$ to the quantity present in this node,
	and updates the concentration
	Returns the INSTANCE of Substance stored in this node,
	with the same id than the Substance given as argument.
	If there is no such Instance, a new one is created and returned
	(used for ECMChemicalReaction).

PhysicalObject

`getCellElement()`	

`set/getMass()`
 	
`set/getAdherence()`
	the mass, i.e. the kinetic friction
	(scales the movement amplitude,
	therefore is considered as the mass)
	the adherence to the extracellular matrix,
	i.e. the static friction (the minimum
	force amplitude needed for triggering a movement).

`set/getVolume()`
`set/getDiameter()`
`changeVolume(double speed)`
`changeDiameter(double speed)`
	
	modifies the volume by `speed` $\cdot\Delta t$.
	modifies the volume by `speed` $\cdot\Delta t$.

`set/getColor()`	

`getMassLocation()`
`moveMassLocation(double speed, double[] direction)`
`double speed, double[] direction)`
	
	active displacement of the point mass
	(direction is normalized)

`getIntracellularConcentration(String substanceId`
`modifyIntracellularQuantity(`
`String id, double quantityPerTime)`
`getMembraneConcentration(String substanceId`
`modifyMembraneQuantity(`
`String id, double quantityPerTime)`	

`isInContact(PhysicalObject o)`	retruns true is the two objects are close enough to exert a force on each other.

PhysicalSphere

`getNeuriteElement()`	

`getMother()`
`getDaughterLeft()`
 	
`getDaughterRight()`
	returns the proximal physical object
	returns the 1st distal cylinder
	(if null: we're a terminal cylinder)
	returns the proximal physical object
	(if null: not proximal to branch point)

`set/getRotationalInertia()`	inertia for rotation

`getActualLength()`
`getRestingLength()`
`get/setSpringConstant()`
`getTension()`
`setRestingLengthForDesiredTension(`
`double tensionWeWant)`	

`getSpringAxis()`	vector from proximal end to distal end

PhysicalCylinder

getNeuriteElement()	
getMother() getDaughterLeft() getDaughterRight()	returns the proximal physical object returns the 1st distal cylinder (if null: we're a terminal cylinder) returns the proximal physical object (if null: not proximal to branch point)
set/getRotationalInertia()	inertia for rotation
getActualLength() getRestingLength() get/setSpringConstant() getTension() setRestingLengthForDesiredTension(double tensionWeWant)	
getSpringAxis()	vector from proximal end to distal end

6.4 ini.cx3d.localBiology

CellElement

getCell()	
getPhysical()	returns the associated physical object.
isASomaElement() isANeuriteElement()	
addLocalBiologyModule(LocalBiologyModule m) removeLocalBiologyModule(LocalBiologyModule m) cleanAllLocalBiologyModules()	
getLocation() move(double speed, double[] direction	returns point mass location of physical obj. moves point mass of associated physical obj.

SomaElement

getPhysicalSphere()	
extendNewNeurite(double diameter, double phi, double theta) extendNewNeurite(double diameter, double[] direction) extendNewNeurite(double[] direction) extendNewNeurite(double diameter) extendNewNeurite()	returns a NeuriteElement. (spherical coordinates) (global cartesian coordinates) *idem* random direction *idem*

NeuriteElement

`getPhysicalCylinder()`	
`isAnAxon()` `setIsAnAxon(boolean a)`	return true if it is considered an axon defines if it is considered (only changes in this NeuriteElement.)

`elongateTerminalEnd(`
` double speed, double[] direction)`
`retractTerminalEnd(double speed)`
	if `direction` points backward,
	no movement.

`branch()` same method exists with direction and/or diameter	splits the compartment into two, and adds a side branch
`bifurcate()` same method exists with direction and/or diameter	adds two small distal cylinders (only for distal compartment).

`makeSingleSpine()`
`makeSingleSpine(double d)`
`makeSpines(double i)`
	
	distance from proximal end
	average interval between two spines

`makeSingleBouton()`
`makeSingleBouton(double d)`
`makeBoutons(double i)`
	
	distance from proximal end
	average interval between two spines

6.5 ini.cx3d.cells

Cell

`getSomaElement()`	
`set/getNeuroMLType()`	`Cell.InhibitoryCell` or `Cell.ExcitatoryCell`

`addCellModule(CellModule m)`
`removeCellModule(CellModule m)`
`cleanCellModules()`	

`divide(`
` double volumeRatio, double phi, double theta)`
`divide(`
` double volumeRatio, double[] direction)`
`divide(double[] direction)`
`divide(double volumeRatio)`
`divide()`
	returns a Cell
	(spherical coordinates)
	
	(global cartesian coordinates)
	idem
	random direction
	idem

`setColorForAllPhysicalObjects(Color color)`	