Diffusion Simulation design document

# Overview

The simulation contains a lot of code – I think it’s fair to say that it is something like 25% of the total amount in Camino, but fear not! There’s an overarching design principal at work, and everything is organised straightforwardly. This chapter covers the following

1. Modular structure
2. Package structure
3. Top-level classes & interface with command line

Before we begin, though, it’s worth defining some terms. There are a few specific terms that I’ve coined because they made sense to me and I’ve been using them ever since. Plus there are a couple of terms that I use interchangeably and it’s worth making it explicit where this happens.

1. *Substrate* – the diffusion environment and everything that’s in it
2. *Geometry* – same as a substrate
3. *Walker* – point-like particle executing a random walk
4. *Spin* – same as walker
5. *Update* – an individual pass of *every* walker (or spin) in a simulation, moving each of them to the next position in their trajectory and updating measurements etc
6. *Timestep* – same as an update
7. *Scan* – a synthetic pulse sequence used to generate data, specified by a scheme file
8. *Structure* – anything in a substrate that interacts with spins
9. *Substrate Object* – a structural component of a substrate, such as a cylinder
10. *Mesh* – A collection of triangles
11. *Trajectory* – the path executed by a spin during the course of a simulation
12. *Voxel* – the space that contains a substrate
13. *Sub-voxel* – a subdivision of the substrate used for spatial optimization

## Modular structure

The simulation is designed to be as flexible as possible. The system is modular, with a simulation consisting of three elements, each of which is interchangeable with others of the same sort. Below is a quick description of each element of the simulation and examples of classes that fall into each category:

* *Dynamics* A module that describes and powers the fundamental dynamics of spins, generating their motion and their interactions with structures in the environment
  + *Step generators, Step amenders*
* *Environment* This module describes the diffusion environment: what is physically present in the space
  + *Substrates, Substrate Objects*
* *Measurement* This module extract information from the dynamics operating in the environment and produces the data desired.
  + *Scans, Stats Modules, Trajectories file writers and parsers*

The package structure of the simulation is designed to reflect these elements. The main simulation package is divided into three sub-packages: dynamics, geometry and measurement, and the geometry package is further divided into “substrates” and “elements” (see fig-1). As we will see later, “*substrates*” are classes that define a diffusion environment in its entirety, inheriting from the abstract class Substrate, whereas “elements” are individual geometric entities that exist within substrates, implementing the interface SubstrateObject. For example InflammationCylinderSubstrate is a *substrate* which contains many cylinders with radii draw from a gamma distribution. The cylinders themselves are *elements*, for example BasicCylinder implements SubstrateObject and forms part of the substrate.

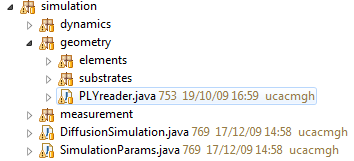


Fig-1: Package structure of the simulation

A typical simulation has one of each of these elements, specified at the command line. Each package and sub-package contains several classes which we will discuss in more detail later, but there are two high-level classes that provide core functionality, and we will cover those here.

### DiffusionSimulation

This is the core class of the simulation. It contains the central functionality that run a specified simulation including initialisation, update and readout. It also implements the VoxelOrderDataSource interface that allows the simulation to mesh with the rest of Camino’s data synthesis framework. This interface consists of the method nextVoxel() which is called by a SyntheticData object to return the readings from each synthetic voxel. This is the entry point for a simulation and defines the sequence of events run in each voxel. This involves calling three other methods in DiffusionSimulation:

1. *initialiseWalkers()* places the specified number of walkers on the substrate prior to the beginning of the simulation proper. Positions are generated and possibly filtered according to command line arguments
2. *runMainLoop()* executes the main update-and-readout loop of the simulation. For each update of the simulation:
   1. *Visit each spin*
   2. *Generate a step*
   3. *Have the spin query the substrate for interaction and amend accordingly*
   4. *Have the spin query the measurement module for interaction with the scan*
   5. *Move the spin*
3. *getSignals()* Generate the signals based on the phase shifts each has aggregated during the simulation

The specifics of each step in the main loop are handled by different modules via inheritance and polymorphism but the central loop is common to all simulations and so is handled here.

### SimulationParams

This class is the simulation’s equivalent of CL\_initialiser. For name-space reasons, parameters specified at the commandline that are UNIQUE to the simulation are defined here. There are many parameters that are shared with other parts of Camino, and those are in CL\_Initialiser with everything else, but the many, many simulation specific parameters are contained here. Commandline parsing is still performed in CL\_Initialiser. This is simply a convenient bag of variables. Specifying any of them sets a flag that tells SyntheticData to run a simulation.

The sequence of events that occurs when a simulation is run is contained in the final section of this document.

# Design Philosophy

As far as possible, I have tried to use design patterns to make the code robust and simple to understand. As a consequence there is one combination of patterns that is used again and again on different scales in the simulation. This chapter explains what it is and how it works. It then covers the important uses of this pattern in the simulation and how it can be used to add additional functionality easily and quickly without undue hacking.

1. Design pattern and how it works
2. Applications of the pattern, including a neat command line interface
3. Extending the simulation: adding a new substrate

## My favourite design pattern

The modular nature of the simulation lends itself to the Factory pattern. This is a simple design pattern that allows sets of similar objects to me maintained an extended without undue fuss. The idea here is that the common elements of a set of different objects are captured in an abstract class or interface which is used as the return type for a method that produces an object of specified type on demand.

The easiest way to illustrate this is with an example. Let’s have a look at StepGenerators. These are objects that generate steps in a spin’s trajectory. All the classes in this example are from the dynamics package.

Typically we use steps that are uniformly distributed on the sphere and of constant length via a FixedLengthStepGenerator, but the Factory pattern allows other types of step generators to be implemented and used. The key to this is that step generators all implement the interface StepGenerator. This defines three methods:

* *getStep()* which generates the steps
* *getType()* which allows the type of generator to be queried
* *getBorder()* which supports some legacy code and is best ignored

Step Generators are instantiated via the StepGeneratorFactory which defines the static method *getStepGenerator(type)* method and an enumerator which defines the types of step generators available. The type of step generator we want is defined by commandline options which lead to a particular member of the enum being specified and the values of the variables in SimulationParams.

The return type of getStepGenerator() is StepGenerator and the main body of the code is agnostic to which particular implementation is provided. I.e. the simulation will work with anything that implements this interface, including something non-stochastic or downright insane.

This pattern of Factories, enumerations and inheritance is repeated over and over again in the simulation, with variations in complexity and features. Examples include:

* Substrates
* Step Amenders (interactions between spins and structure)
* Synthetic Scans (implementations of different pulse sequences)
* Stats modules
* Walkers (spins)
* Cylinders

Some of these are more complex than others. The substrate inheritance hierarchy is a little more complicated, for example, and until recently there was only one type of walker but nevertheless still a factory.

The advantage of using factories is extensibility. To add a new type of substrate,

1. Implement the new substrate by extending the abstract Substrate class
2. Add a new member to the SubstrateType enumerator
3. Add an additional option to the Substrate Factory getSubstrate() method
4. Add an option to parse the commandline string for the new substrate type in CL\_Initialiser. This will be inside the “-substrate” option in the CL\_init() method. The easiest way to find it is to search for “-substrate” in CL\_Initialiser.

This procedure will work with any of the examples given above. New cylinder types will work with any cylinder-based substrate automatically, for example.

As an extension to this pattern, there is an additional pattern that I’ve used in Laura’s compartment model framework that isn’t here but makes things a lot simpler to tie to the commandline. I don’t know if this thing has a name or whether I invented it but for the sake of clarity let’s call it an Identifier pattern.

This just uses the toString() method of each member of the factory enumerator to compare to the string given on the commandline (case insensitive). It means that the final step of the above could be omitted. The Identifier would slot in nicely with the simulation code. Just thought I’d mention it.

## Some more about substrates

The first step in the instructions above is to implement a substrate by inheriting from the abstract class Substrate. Any entity in the simulation that makes use of a factory has either an abstract class or interface and hopefully the javadoc will be enough to see what each method is for. Substrates, however, have one or two other subtleties in their design that are worth delving into, and so in this section describes the ideas in more detail.

Substrates themselves describe a diffusion environment, this means defining a cuboidal region of a specified size and filling it with structures. The boundaries of this cuboidal region are periodic, so if a spin leaves one side of the region it will rejoin on the opposite side. There is a little piece of double-think involved here: on the one hand, spins are confined to a periodic region and their coordinates are mapped into this space prior to checking for intersections with structures whereas from the point of view of calculating phase shifts due to interactions with a scan, the spins inhabit an infinite 3d array of repeating substrates through which they move. This distinction is important. A spin needs to be mapped into a periodic region for intersection checking, but needs to maintain global, non-periodic coordinates to avoid teleporting around during a scan and introducing huge distortions into any accumulated phase shifts.

The structures in a substrate are defined as a *Collection* of *SubstratObject*s. *SubstrateObject* is an interface that is implemented by anything that is used as an elementary part of a substrate. So, for example, *Triangle* implements it, as does *BasicCylinder* although this in fact implements *Cylinder*, which is a sub-interface of *SubstrateObject*. The interface includes several methods, all used for intersection checking and spatial optimization. These are as follows:

* *crosses() –* checks intersection between the object and a spin making a step
* *getDiffusivityAt()* – returns the local diffusivity at given position. This is used to allow spatial variation of diffusivity, although this is not fully implemented
* *getBoundingBox()* – returns an oriented bounding box around the object. This is an alternate spatial optimization scheme using trees that is not fully implemented
* *boundingBoxIntersects()* – checks intersection between a bounding box a spin making a step. This is also not fully implemented
* *inside()* – checks if a given point is inside or outside the object
* *intersectsCubicRegion() –* checks if the object intersects a specified cuboidal region. This is used in conventional spatial optimization and IS fully implemented

The most important methods here are crosses() and intersectsCubicRegion(), used for intersection checking and spatial optimization respectively.

It’s worth pointing out that substrate objects can be quite complicated entities, possibly compound objects. An example of a more complex SubstrateObject is the *MyelinatedCylinder*. This is a thick-walled cylinder *a la* Sen and Basser. It’s made up of two *BasicCylinder*s along with some extra machinery to handle different diffusivities in different regions. There’s nothing to stop you implementing compound substrate objects although this is not always as useful as it appears. A notable omission in this context is that there is no mesh object. A mesh substrate is just a collection of triangles and requires no additional machinery so we just handle all the triangles separately.

The distinction between substrates and substrate objects is the reason for the two sub-packages: the geometry package is divided into *substrates* and *elements*, which contain substrate objects. There is no single substrate object factory as this doesn’t make a lot of sense, but there is a cylinder factory that handles the various different kinds of cylinder to avoid replication of substrate classes.

## Schemes and Scans

This real raison d’etre of the simulation is to synthesise synthetic diffusion-weighted measurements directly from diffusive dynamics. As such it needs to interface directly with the scheme framework in Camino in order to accurately obtain phase shifts for spins. For historical reasons, this part of the simulation also follows the factory pattern we’ve seen elsewhere, but in fact only one type of scan implemented. A scan factory manufactures a choice type of object that implements an interface called *SyntheticScan* which defines the following methods:

* *getSignals() –* returns the set of signals specified in a scheme file based on what is accumulated during the simulation
* *getCompartmentSignals()* – as above, but from walker only in a given compartment (intra or extracellular)
* *getPhaseShift() –* return the phase shift at a given location at a given time (obsolete)
* *getNumMeasurements()* – return the number of measurements in the scheme (obs)
* *update()* – so something in each timestep (unused so far)
* *getScanType()* – what type of scan is this?

One small difference between the simulation and an actual scan acquisition is that the simulation accumulates phase shifts for each line in the scheme file simultaneously from the same set of dynamics rather than acquiring them one-by-one.

In fact, there is only one implementation of this interface and only one type of scan made by the factory: the agnostic scan. An agnostic scan queries a scheme object for phase shifts at during a simulation. Scheme objects that are used by the simulation must implement the interface *SimulableScheme* and are discussed in the documentation for the Camino scheme framework.

Phase shift calculation is now made in the scheme objects themselves. We are now in a position whereby we have scheme objects for PGSE and TRSE sequences as well as a scheme that can take on a completely arbitrary three dimensional gradient waveform which covers just about anything you can think of from gen-waves to rotating chirps. It is unlikely you’ll need to add anything to this framework unless you need additional MR physics beyond gradient-based phase shifts.

In summary, *AgnosticScan* provides the bridge between the simulation and the Camino scheme framework, and we’re able to do just about anything you can think of in terms of gradients. Just construct the appropriate scheme file and the simulation will make use of it.

# Class Hierarchies

By this stage, the glorious simplicity of the simulation should be shining like a simple, shiny thing and this makes understanding the inheritance structure of the code very simple. Most of the inheritance is pretty self-explanitory, but as usual substrates are slightly more complicated. Since most of the inheritance hierarchies are more or less trivial, I’ll list most of them without much comment.

* Concrete classes are blue
* Abstract classes are green
* Interfaces are maroon

## Substrates

All substrates inherit from the abstract root class Substrate. There are then four main types:

* Cylinder substrates, which contain cylinders
* Mesh substrates, which contain triangles
* Empty substrates, which contain nothing
* Cellular lattices, which contain 3D arrays of cubes

Cylinder substrates and cellular lattices are also abstract classes which must be inherited from for a usable substrate. Cylinder substrates are as follows:

* Parallel cylinder substrate – regularly packed (square or hex) cylinders
* Crossing cylinders substrate – two crossing cylinder populations of same radius with given angle arranged in alternate sheets
* Distributed radius cylinder substrate – gamma distributed cylinders without inflammation
* Squashy inflammation substrate – gamma distributed cylinders with inflammation
* Sticky Cylinder Substrates – square packed regular cylinders which spins stick to rather than bounce off.

Cellular lattices are a set of cubes packed together. Striped lattices have “lanes” of space between them so that the empty space is directed. It’s obsolete but there for historical reasons. The perc lattice has cubes placed randomly with a fixed probability and is there because it might possibly be handy to have something disordered in a quantifiable way. So here’s 3D percolation if you need it.

This is by far the most complex hierarchy in the simulation. With a bit of luck you won’t need to add much to it, as most things can be done with a mesh or a straightforward new cylinder substrate.

## Substrate Objects

There’s also a certain amount of complexity in this hierarchy as well, but nothing like as much as in the substrates.

Substrate object and Cylinder are both interfaces. Triangle implements substrate object and the various cylinder classes implement Cylinder, which is a sub-interface with a couple of extra methods. There are a couple of other things in the elements package that aren’t used: bounding boxes, lines and quads. They’re implemented but not used.

From here on it’s pretty straightforward. Sets of objects inheriting from a root abstract class or interface. For completeness, here’s everything.

## Step generators

## Step amenders

## Scans

## Walkers

Here Walker is a concrete class and is used in most simulations. StickyWalker is a subclass that implements sticking to a cylinder rather than a conventional step amender.

## Stats modules

That about covers it.

# Other stuff

At this point we’ve covered almost everything, however there are a couple of bits of functionality and sequences of events which haven’t been covered elsewhere, so we’ll cover them here.

## Spatial optimisation

The mesh substrate and the more complex cylinder substrates make use of spatial optimisation. The method we use is regular grid spatial optimisation whereby the substrate is subdivided into lxmxn boxes called subvoxels. The values of l, m and n are set in an array of three ints defined locally and passed as an argument to the *initialiseSpatialOptimisation()* method of Substrate.

Spatial optimisation is initialised for each subvoxel by compiling a list of substrate objects that intersect with it. This is performed once at construction of the substrate. These lists are stored in Substrate. To check for intersections with a step, we first construct a candidate list of subvoxels using the *getCandidates()* method and then check intersection with each substrate object in the candidate subvoxels. There is no separate Spatial optimiser class, all this functionality is in Substrate but must be explicitly invoked from the substrate object that wants to make use of it.

PLY files

Meshes are stored and defined via a simple file format called PLY. This is a (technically) human-readable format that was chosen for its simplicity. The simulation does not implement the complete PLY format, but does understand a sensible and sizable subset of it. The PLY reader has its own class and is in the *geometry* package (the only class not to be a member a sub-package). The easiest way to explain the format is through an example.

Here are the contents of a simple PLY file of the type the simulation prefers:

Always need this line

ply

Defines number of vertices in the mesh

format ascii 1.0

element vertex 12

Defines data format. Simulation assumes this format is the same

property double x

property double y

property double z

Defines number of vertices in the mesh

element face 20

property list uchar int vertex\_index

end\_header

Always need this line

0 -0.525731 0.850651

0.850651 0 0.525731

0.850651 0 -0.525731

-0.850651 0 -0.525731

-0.850651 0 0.525731

Vertex coordinates (SI units)

-0.525731 0.850651 0

0.525731 0.850651 0

0.525731 -0.850651 0

-0.525731 -0.850651 0

0 -0.525731 -0.850651

0 0.525731 -0.850651

0 0.525731 0.850651

3 6 2 1

3 2 7 1

3 5 4 3

3 8 3 4

Connectivities defining object faces.

<number of corners> <vertex1>...

The code will cope with any number of corners on a face, but will assume a convex shape as it convert them into triangle fans.

Indexing of vertices starts at 0 and is in the same order that they are specified in the block above.

3 11 5 6

3 10 6 5

3 2 10 9

3 3 9 10

3 9 8 7

3 0 7 8

3 1 0 11

3 4 11 0

3 10 2 6

3 11 6 1

3 10 5 3

3 11 4 5

3 9 7 2

3 0 1 7

3 8 9 3

3 0 8 4

## What happens when you run a simulation

Here we’ll go through the specific steps that occur when a simulation is run. This is as inclusive as possible. Some steps will not be run in some simulations (e.g. not all simulations use spatial optimisation) but it seems better to be inclusive.

The entry-point is the main method of SyntheticData, so let’s start there:

1. Initialise data types
2. Parse commandline – setting any simulation variable will set a flag that activates a simulation.
3. Init scheme
4. Initialise data synthesiser –
   1. checks scheme for compatibility
   2. assembles simulation parameters
   3. get step generator
   4. call DiffusionSimulation constructor
      1. initialise simulation variables
      2. call substrate constructor
         1. initialise substrate objects
         2. initialise spatial optimisation
      3. report to commandline
5. init output manager
6. outer data synthesis loop: call DiffusionSimulation.nextVoxel()
   1. get step generator from simulation params
   2. initialise walkers – for each walker
      1. pick walker position
      2. instantiate walker
   3. run main simulation loop
      1. set geometry testing arrays
      2. initialise substrate
      3. for each time step
         1. for each walker
            1. pick step
            2. query substrate for amendment

get candidate subvoxels

check barriers and voxel boundaries

compile list of intersections

amend step

check amended step

repeat until no more intersections

* + - * 1. get phase shifts from scan and scheme
        2. add to accumulated phase shifts
        3. move walker
      1. update scan
    1. calculate signals from accumulate phase shifts & return
  1. output signals

1. close output manager
2. finish

## Some debugging strategies in case things go wrong

Most of what goes wrong with the simulation occurs during intersection checking. If you see messages about “walker such-and-such has crossed” that means a walker has stepped through an impermeable barrier, i.e. an intersection has been missed. Similarly “erroneously detected barrier crossing” means that an intersection has been found when no barrier is present. Both of these are tough to solve, and in the past I have found that I resort to graphical representation of the problem using excel helps a lot. I’ve also used the camino visualise to help me find out what’s gone wrong.

Test cases will help, but most bugs are due to floating point accuracy problems or bizarre unforeseen circumstances. Isolate, visualise, fix. Repeat ad infinitum.

## Random thoughts about next steps, big and small, and how to implement them

A few random things:

* Spatially varying steps are implemented except for adjusting steps as walkers step through barriers. Shouldn’t be too difficult to complete.
* Permeabilities are stored per substrate object. In theory different objects could have different permeabilities, but there is no mechanism to specify them. Add this mechanism and the code will work.
* Interacting walkers could be implemented via an appropriate step generator – there’s no reason why steps have to be stochastic
* Subvoxel grids for spatial optimisation could be defined on the commandline.
* T1 and T2 effects are implemented via the getMagneisation() method in substrate and on the commandline via -T2 but only on the sticky cylinder substrate.
* Meshes are assumed not to be concave (i.e. not to have a well-defined interior and exterior) but adding a line to the ply header that says “comment closed surface” will change this. If you switch this on on a non-closed substrate then I will not be held responsible for the consequences.

I think that about does it. Have fun with this mountain of beans...