ARL KMCThinFilm Manual

Version 0.2.7

Contents

Chapter 1

Introduction to the ARL KMCThinFilm library

The ARL KMCThinFilm library is designed to assist researchers in creating kinetic Monte Carlo (kMC) simulations of thin film growth where the particles involved in the growth (which may be atoms, molecules, or some sort of coarse-grained “effective” particle) are assumed to be restricted to be on or near the sites of a lattice. What the library contains:

- A general lattice class that represents the lattice as a three-dimensional array of cells—but not necessarily in a cubic or orthogonal arrangement—where each cell contains an array of integers and/or floating-point numbers that may be used for a variety of purposes, such as whether a site within a cell is occupied, or the coordinates of an atom near a lattice site (for models where atoms may be on a distorted lattice).
- Solvers that select and execute events at each time step of a simulation.
- Hooks to add possible events, such as deposition, diffusion, or a reaction, in accordance with an implementation of a desired physical model.
- Hooks to add actions that run every *t* seconds of simulation time or every *N* time steps. These actions may simply dump the state of the lattice at various points during a simulation, or to alter it.
- A virtual class interface for random number generators, to allow generators not included with the ARL KMCThinFilm library to be used.
- Implementation of an approximate parallel kMC algorithm.

However, this library does *not* contain details about the physics of any particular kMC model. For example, there is no assumption that the model must be solid-on-solid, or that the lattice be simple cubic. The library also does not require that the rates of kMC events be determined via a particular theory, e.g. harmonic transition state theory. Aside from the assumptions that the model (1) involves sites on a lattice and (2) has periodic boundary conditions along the in-plane dimensions, the library's framework should be fairly flexible. The downside of this flexibility, of course, is the need to write additional code to create a complete kMC application. Use of this library, though, should still be far simpler than writing an entire kMC application from scratch.

Chapter 2

Compilation and installation

2.1 Compiling and installing the library itself

Compilation of the ARL KMCThinFilm library requires CMake < `http://www.cmake.org`> and Boost < `http://www.boost.org`>. Optional dependencies are the random number generators DCMT, at least version 0.6.2 (for KMCThinFilm::RandNumGenDCMT), and RngStreams, at least version 1.0.1 (for KMCThinFilm::RandNumGenRngStreams).

On a Unix-like system, one may write a shell script such as the following in order to run the command-line version of CMake:

```
#!/bin/sh

EXTRA_ARGS=$@

KMC_HOME="/path/to/KMCThinFilm/source-and-docs"

#CMAKE_BUILD_TYPE=DEBUG
CMAKE_BUILD_TYPE=RELEASE

cmake \
      -D CMAKE_BUILD_TYPE:STRING=$CMAKE_BUILD_TYPE \
      -D CMAKE_INSTALL_PREFIX:PATH="/path/to/desired/KMCThinFilm/install/location" \
      -D BOOST_ROOT:PATH="/path/to/Boost" \
      -D KMC_USE_DCMT:BOOL=TRUE \
      -D DCMT_ROOT:PATH="/path/to/dcmt0.6.2" \
      -D KMC_USE_RNGSTREAMS:BOOL=TRUE \
      -D RNGSTREAMS_ROOT:PATH="/path/to/RngStreams" \
      -D CMAKE_CXX_FLAGS:STRING="-DMPICH_IGNORE_CXX_SEEK" \
      -D CMAKE_CXX_STANDARD=11 \
      $EXTRA_ARGS \
      ${KMC_HOME}
```

Although the ARL KMCThinFilm code does not use C++11, recent enough versions of Boost *may*, so "`-D CMAKE_CXX_STANDARD=11`" is added to the above example script.

If either `KMC_USE_DCMT` or `KMC_USE_RNGSTREAMS` is set to `FALSE`, the corresponding path, i.e. `DCMT_ROOT` or `RNGSTREAMS_ROOT` does not need to be specified. The extra compilation flag `-DMPICH_IGNORE_CXX_SEEK` is only needed when compiling against certain MPI implementations, such as MPICH or IntelMPI. By default, an attempt will be made to compile both the serial and parallel versions of the library. If an MPI installation is not found, only the serial version will be built. To avoid attempting to compile the serial version, set the CMake variable `KMC_BUILD_SERIAL` to `FALSE`, and to avoid attempting to compile the parallel version, set the CMake variable `KMC_BUILD_PARALLEL` to `FALSE`. (One may set these variables false by adding `-D KMC_BUILD_SERIAL:BOOL=FALSE` or `-D KMC_BUILD_PARALLEL:BOOL=FALSE`, accordingly, to the command line executing CMake.) To avoid installing documentation, set the CMake variable `KMC_INSTALL_DOCS` to `FALSE`.

The above shell script should be run from a directory **different** from the directory containing the source and documentation of the ARL KMCThinFilm library. After the script is run, one may then type "make" and then "make install".

2.2 Compiling against the library

There are a couple things to note when compiling against the ARL KMCThinFilm library. First, there are two versions of the library, named “KMCThinFilmSerial” and “KMCThinFilmParallel.” One must compile a given binary against exactly one of these. Second, *two* paths to ARL KMCThinFilm header files must be specified. If the root directory of the ARL KMCThinFilm installation is `$KMC_INST`, then one of the paths is simply `$KMC_INST/include`. The other path depends upon whether one is compiling against the serial or parallel version of the library. For the serial case, the other path is `$KMC_INST/include/KMCThinFilm/serial`. For the parallel case, the other path is `$KMC_INST/include/KMCThinFilm/parallel`. To repeat in brief,

- To compile a serial application, compile against the KMCThinFilmSerial library binary and include `$KMC_INST/include` and `$KMC_INST/include/KMCThinFilm/serial` in header search paths.
- To compile a parallel application, compile against the KMCThinFilmParallel library binary and include `$KMC_INST/include` and `$KMC_INST/include/KMCThinFilm/parallel` in header search paths.

Chapter 3

Concepts and algorithms

3.1 Basics of kinetic Monte Carlo simulation as implemented in the ARL KMCThinFilm library

In the ARL KMCThinFilm library, a simulation is an object that executes the *n*-fold way algorithm [2], where at a given time step in the simulation, the set of all possible events in a system and their associated rate constants or propensities (i.e. probabilities per unit time) is determined, and one of these events is then randomly chosen to be executed, with the choice weighted according the propensities of the possible events. Following a later kMC work [4], the simulation time at each step is advanced by $-\ln r/p_{tot}$, where r is a random number uniformly chosen from the open interval (0,1), and p_{tot} is the sum of all the propensities of all possible events. To simplify the process of determining the set of possible events, each possible event is assumed to cause some sort of change to cells in a lattice. This lattice is itself an object that is owned by the simulation object, and it is initialized when the simulation object is initialized. Possible events are assumed to be of one of two kinds:

- *Cell-centered*. This is a type of event that originates in the neighborhood of some lattice cell, and a propensity of an instance of such an event at a particular cell is affected by the states of cells in the neighborhood of that particular cell. Adsorption, diffusion, or chemical reactions are examples of such events. Since the propensities of related cell-centered events can often be calculated using the same or almost the same series of steps, types of cell-centered events may be collected into one or more groups. A group of cell-centered event types consists of the following components:
  - An integer label identifying the group of event types.
  - A set of the relative locations of the cells in the aforementioned neighborhood, which are called "offsets."
  - A function or function object that calculates what amounts to an array (or more precisely, an `STL vector`) of propensities, one for each instance of a type of event in the group happening at a given cell. This function/function object is given very limited access to the lattice object owned by the simulation. In particular, it can only read the states of lattice cells within the neighborhood of cells defined by the offsets mentioned above.
  - A group of functions or function objects, one for each type of event in the group of cell-centered event types, that can execute an instance of that event type about a given cell. Each of these function/function objects can alter the lattice that is owned by the simulation object, and not just the cells of the lattice within a certain neighborhood of the cell about which an event is centered. In addition, it has read-only access to some aspects of the current simulation state, such as the elapsed simulation time. Optionally, for each function/function object in the group, there may be a set of offsets that indicate the relative locations of cells directly changed by the execution of the event, and this may be used to speed up the simulation.
- *Over-lattice*. This is a type of event that, from a physical perspective, is assumed to have actually originated at some point well above the lattice, out of the range of influence of things that happen at the surface or interior of

the lattice. Deposition would be the chief example of this type of event. An over-lattice event type consists of the following components:

- An integer label identifying the type of event.
- A propensity per unit area, i.e., the propensity of the event divided by the number of cells in a lattice monolayer. A deposition flux given in terms of the number of monolayers per unit time is already a propensity per unit area.
- A function or function object that can execute an instance of this event type originating from a randomly picked cell at the top of the lattice. This function/function object can alter the lattice that is owned by the simulation object, and if need be, it can alter cells that are far distant from the originating randomly picked cell. It also has read-only access to some aspects of the current simulation state, (e.g. elapsed simulation time).

After the simulation object is initialized, these event types are added to the object. Once these types are added, the simulation is then run for a certain amount of simulation time. A simulation can also be restarted from where it left off, and before restarting, event types can be added, modified, or removed from a simulation object. For example, one could add both deposition and diffusion event types to a simulation, run the simulation for t_{dep} units of time, then remove the deposition event type, change the objects that calculate the propensities of the diffusion event type to those for a higher temperature, and then restart the simulation from where it left off for t_{anneal} units of simulation time.

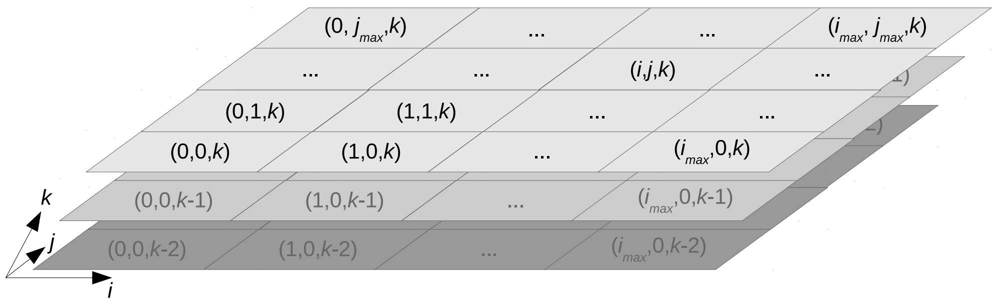
At the beginning of a simulation run, each cell of the lattice is scanned in order to determine the initial set of possible events and their propensities. When an event is executed at a time step, the locations of the lattice cells directly changed by this event, as well as certain cells within a neighborhood of these changed cells, are recorded. The list of recorded cell indices, then, is used to incrementally update the set of possible events and propensities, avoiding the computational cost of scanning each cell of the the lattice at each time step to refresh this set. This list is constructed from the offsets used to define cell-centered events. More precisely, it is constructed using a master list of offsets that is generated at the beginning of the simulation run, one that is the union of all the offsets of cell-centered events. If *auto-tracking* is used to track the locations of the directly affected cells, then when an event is executed, for each cell (a_n, b_n, c_n) directly changed by the event, cell indices (a_n, b_n, c_n), $(a_n - i_1, b_n - j_1, c_n - k_1)$, $(a_n - i_2, b_n - j_2, c_n - k_2)$, etc. will be added to the list of indices of cells affected by the event, where (i_1, j_1, k_1), (i_2, j_2, k_2), $\ldots$, is the master list of offsets. When this list of indices is constructed in this way, there may be redundancies in the list. These redundancies won't lead to spurious results, but they will lead to redundant propensity calculations, since a propensity calculation is done for every item in the list. To avoid this, *semi-manual* tracking may be used. To use this mode of tracking for a cell-centered event, sets of offsets $\{O_1, O_2, \ldots\}$, indicating the relative locations of cells changed by an event, must be specified for the type of this cell-centered event. From this information and the master list of offsets, new lists of offsets are constructed, one for each set O_m. For each offset (a_n^m, b_n^m, c_n^m) in set O_m, a list of offsets $l_n^m = (a_n^m, b_n^m, c_n^m), (a_n^m - i_1, b_n^m - j_1, c_n^m - k_1), (a_n^m - i_2, b_n^m - j_2, c_n^m - k_2)$, etc. is generated, where (i_1, j_1, k_1), (i_2, j_2, k_2), $\ldots$, again is the master list of offsets. The list L_m is then constructed as the union of all offsets in the lists l_1^m, l_2^m, $\ldots$. When an event is executed, for each set O_m, one of the cells affected by the event is designated as *center m*. For each center m, the indices of the rest of the cells changed by the event are formed adding the indices of this center to each of the offsets in O_m. The list of recorded cell indices is then composed of the indices of these centers, and the indices formed from adding the indices of center m to each of the offsets in L_m. If the centers are far enough apart or there is only one center, then the list of recorded cell indices should have no redundant values.

The process of choosing a random event may be done with one of two algorithms, which in the ARL KMCThinFilm library are called *solvers*. One algorithm [1] stores the N possible events and partial sums of their propensities in a binary tree and scales as $O(\log_2 N)$. Another algorithm stores possible events in a map where the key is a propensity of a possible event, and the value associated with that key is an array of possible events associated with that propensity. This algorithm scales with the number of unique propensity values in the system, which in many cases is independent of the number of possible events N. This is similar to another algorithm [8], except that algorithm used a two-dimensional array rather than a map.

In addition to possible events, *periodic actions* can be executed during a simulation as well. There are two types of periodic actions: *time-periodic* actions, which occur after every t units of simulation time, and *step-periodic* actions, which occur after every M time steps. These may perform various functions, from I/O operations, e.g. to record to a file the state of the lattice at various points during a simulation, or to even change the lattice or quantities associated with it. Like possible events, periodic actions can be added, changed, or removed from a simulation object.

3.2 Lattices in the ARL KMCThinFilm library

A lattice in the ARL KMCThinFilm library is modeled as a stack of two-dimensional arrays of cells, with each cell labeled with a triplet of integer indices, as illustrated in the figure below.



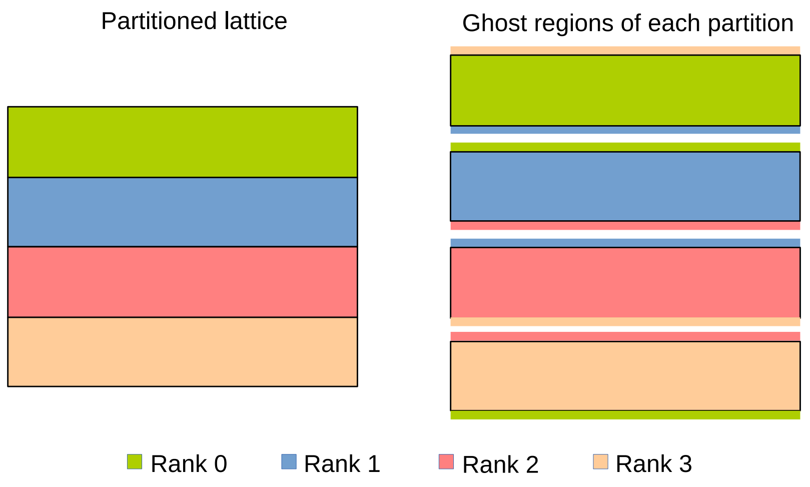
Cell (i,j,k) contains two arrays:

$$I_{ijk} = \left[i_{ijk}^{(0)}, i_{ijk}^{(1)}, i_{ijk}^{(2)}, \ldots, i_{ijk}^{(N_{\text{int}})} \right] \qquad F_{ijk} = \left[f_{ijk}^{(0)}, f_{ijk}^{(1)}, f_{ijk}^{(2)}, \ldots, f_{ijk}^{(N_{\text{float}})} \right]$$

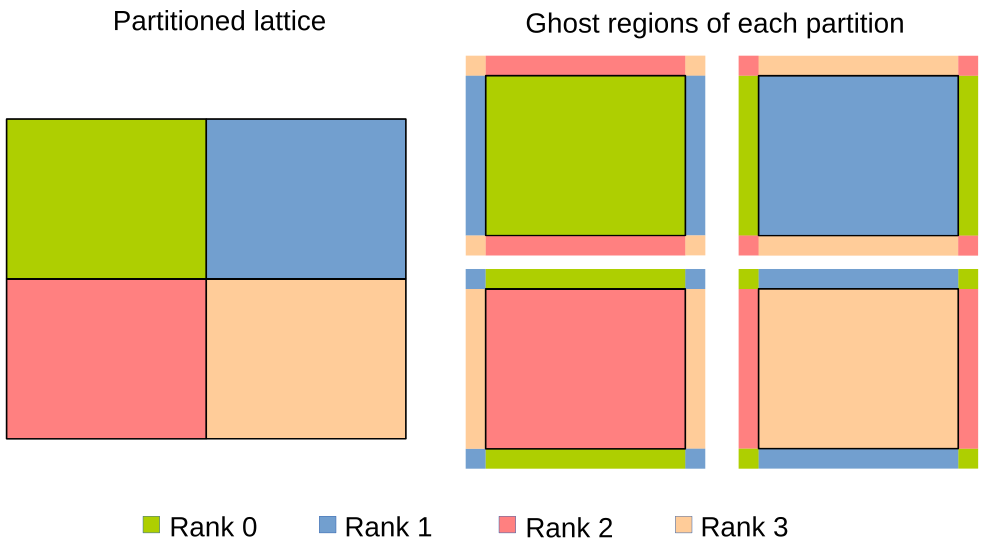
Periodic boundary conditions always apply to the first two indices of a cell, but not the third. In a manner similar to that of the software SPPARKS $<$ `http://spparks.sandia.gov`$>$, each cell also contains two one-dimensional arrays, one with integer values and one with double precision floating-point values. The size of these arrays is the same for all cells. The physical meaning of the contents of the arrays is left up to the client application that uses the ARL KMCThinFilm library; for a given application, they may be used to identify species of a basis of atoms within a cell, or the spatial coordinates of an atom in a cell in a possibly distorted lattice, or, for a simpler solid-on-solid application, the height of a column of atoms at $(i, j, 0)$. The lattice need not be cubic. If the cells are treated as those of a Bravais lattice with primitive lattice vectors $\mathbf{a}_i$, $\mathbf{a}_j$, and $\mathbf{a}_k$, then the physical location of cell (i, j, k) may be said to be $\mathbf{a}_i i + \mathbf{a}_j j + \mathbf{a}_k k$. The sizes of the arrays in each cell of the lattice, as well as the maximum values of the first two indices of a cell, are fixed when the lattice is initialized, but additional planes may be added to the lattice at any time step of the simulation.

3.3 Parallel approximate Kinetic Monte Carlo algorithm

The lattice in a parallel kMC simulation is partitioned among the processors in an MPI communicator. There are two methods of decomposition. The simplest is row-based decomposition, shown below.

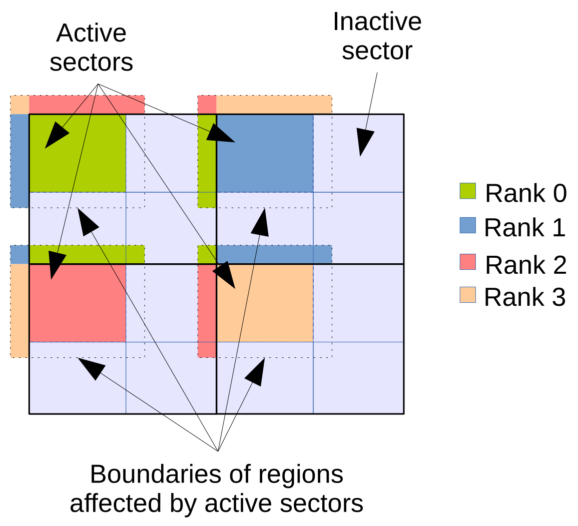


In the above diagram, a lattice is divided among four processors with ranks ranging from 0 to 3. Here, “ghost sites” appear along the top and bottom of each partition of the lattice. The ghost regions along the edges of each partition are copies of the lattice sites of a neighboring processor, and in the diagram, the processor from which they are copied is shown via their color. Periodic boundary conditions are employed here, so that the top of the rank 0 partition is connected to the bottom of the rank 3 partition. Ghost regions are not needed for the left and right sides. Alternatively, the lattice may be decomposed such that the perimeter of each partition is minimized. Here, this is called *compact* decomposition, and is shown below.



Now, ghost regions are along the whole boundary of each portion of the lattice. Again, the lattice is shown divided among four processors with ranks ranging from 0 to 3.

Attempting to do kMC simulations on each partition of the lattice would lead to problems at the partition boundaries, since the events done on each partition could lead to conflicting effects on the ghost sites. To avoid this problem, an approximate kMC algorithm was developed [10], where each partition is further subdivided into sectors, and at any given time in the simulation, events are executed only for sites within one of these sectors, as illustrated below for the case of compact decomposition.



Partition boundaries are indicated by thick solid lines, while the sector boundaries are indicated by thinner solid lines. Active sectors are shown in the color corresponding to the rank of the partition to which they belong. The dotted lines show the boundaries of the regions affected by events that occur within the active sectors. The parts of these regions that affect ghost sites are shown in the color corresponding to the ranks of the sites of which the ghost sites are copies. If row decomposition were used, there would be two sectors per partition instead of four. Here, in the above diagram, the active sectors happen to be the upper left quadrants of the lattice partitions. At any given time in the simulation, they could be the lower left, lower right, or upper right quadrants, so long as the relative locations of these active sectors are the same for all processors, that is, *all* upper left, *all* lower left, and so on. Because the active sectors all have the same relative location, the regions that are affected by events happening within them do not overlap, as illustrated by the diagram above.

With the sectors now defined, the approximate kMC algorithm can proceed on each processor as follows:

1. Initialize the global time t to zero.
2. Determine the initial value of the stop time t_{stop}.
3. Repeat the following until t exceed the desired time t_{max}:
   (a) Iterate over the sectors. For each sector visited,
      i. initialize the local time t_{local} to zero,
      ii. update the ghost sites and the set of possible events affected by changes to the ghost sites,
      iii. run a normal serial kMC algorithm on the sites within the sector, incrementing t_{local} as each event is executed until $t_{local} > t_{stop}$, but do not allow the event that would cause t_{local} to exceed t_{stop} to be executed, and
      iv. update the off-processor sites that correspond to the ghost sites that have changed due to the events that have occurred, and again update the set of possible events affected by the ghost site updates.
   (b) Increment the global time t by t_{stop}.
   (c) Update the value of t_{stop}.

The updates of the ghost sites that are performed when a sector is visited do not involve communicating the entirety of the ghost site regions bordering a sector, only the communication of *changes* to each region.

In principle, the sequence in which the sectors are visited may be random, so long as (1) each processor uses the same random sequences (in order that the active sectors on each processor are at the same relative location, as mentioned previously), and (2) that four sectors are visited before the global time is incremented. However, in the ARL KMCThinFilm library, the simpler approach seen in another kMC code, SPPARKS [7], is taken, where each sector is simply visited in turn in a deterministic loop. In the ARL KMCThinFilm library, this loop begins at the upper left and then continues to the lower left, followed by the lower right and upper right.

Compact decomposition, despite minimizing the perimeter of each partition, may be slower than row-based decomposition, because the former mode of decomposition requires twice as many sectors as the latter, and each visit of a sector requires the communication of ghosts. The volume of data communicated in row-based decomposition may indeed be higher than that in compact decomposition. However, since only changes to ghost regions are communicated, this volume is not that high to begin with, so the communication overhead is dominated more by the very acts of sending and receiving messages, rather the costs associated with the size of the data itself. Because of this, row-based decomposition is the default in the ARL KMCThinFilm library.

The value of t_{stop} may be determined by various time-stepping schemes. The simplest of these is to set t_{stop} to a fixed value. The other schemes are various kinds of adaptive algorithms, which attempt to determine a reasonable value of t_{stop} from the propensities of the possible events in the simulation. In all of these algorithms, the time step has the general form,

$$t_{stop} = \frac{n_{stop}}{F_{stop}}$$

where n_{stop} is an adjustable parameter, and F_{stop} is a function that determines the particular adaptive time step scheme. Here are the currently available choices for F_{stop}:

- The maximum propensity of all currently possible cell-centered events in the simulation. This is a simplified version of the adaptive method of determining F_{stop} recommended by [10]. When this method is used, a good conservative value of n_{stop} is 1.0, though in some applications, such as island coarsening, a value of up to 10.0 has been used without much loss of accuracy [9].
- The maximum of the average propensities per possible cell-centered event from each sector. $F_{stop} = \max(p_s^1, p_s^2, \ldots, p_s^{N_{proc}})$, where N_{proc} is the number of processors and for the case of compact decomposition, $p_s^n = \max(p_{s,UL}^n, p_{s,LL}^n, p_{s,LR}^n, p_{s,UR}^n)$, where $p_{s,UL}^n$ is the average propensity of the events in the upper-left sector of partition n, that is, the sum of the propensities of all possible events in that sector divided by the number of possible events, and similarly, $p_{s,LL}^n$, $p_{s,LR}^n$, and $p_{s,UR}^n$ are the mean propensities in the lower left, lower right, and upper right sectors of partition n, as in SPPARKS [7]. For this choice of F_{stop}, a reasonable starting value for n_{stop} is 1.0. Note that for a given value of n_{stop}, this approach may be less conservative than the previous one.

An additional optional parameter $t_{stop,max}$ may be used with the adaptive schemes. If this parameter is set, then if $n_{stop}/F_{stop} > t_{stop,max}$, t_{stop} will equal $t_{stop,max}$ instead of n_{stop}/F_{stop}. This may be useful in cases where the adaptive scheme temporarily overestimates t_{stop} during a simulation.

In a parallel simulation, the propensity of an over-lattice event is the propensity per unit area scaled by the in-plane area of a *sector*, rather than the size of a whole monolayer. Also, running a parallel simulation changes how periodic actions are run. In a serial simulation, if a periodic action executes, it executes shortly after an event has been executed. In a parallel simulation, if a periodic action executes, it executes shortly after t_{stop} has been incremented, that is, outside of the looping over sectors. This allows periodic actions to use MPI calls for parallel communication, since a periodic action will execute the same number of times on every processor.

Chapter 4

Illustrating the usage of the ARL KMCThinFilm library by example

Perhaps the best way to showcase the functionality of the ARL KMCThinFilm library is to provide examples of its use, so several examples will be shown here. The first example is a very basic solid-on-solid model and only requires a two-dimensional lattice. It is intended to give an overall sense of how the library is meant to be used. The second example shows what is needed to implement a parallel kinetic Monte Carlo simulation with this library. The third example is slightly less simple in that it shows how to implement a three-dimensional simulation, and it shows a few other features of the library. The fourth example involves a model of a patterned substrate that encourages two-dimensional island formation at regular places along a grid. It is meant to show how the ARL KMCThinFilm library can be used to speed the implementation of models with features that could require custom code to implement, and again shows some new features of the library not mentioned in previous examples. It is expected that these examples be read *in order*, since matters discussed in detail in a previous example may be reviewed only briefly, if at all, in later examples. It is also recommended to read all of the examples, in order to get a better overview of the library's features.

4.1 Example: Implementation of a “fractal” solid-on-solid model

This example implements a “fractal” kinetic Monte Carlo model (similar to that described in the work that developed the approximate semirigorous parallel kMC [10]). In this simple model, the lattice is simple cubic. The fractal model has two kinds of possible events: deposition and diffusion. The deposition flux, i.e. propensity per unit area, is F monolayers per unit time. Here, diffusion is a hop of a particle from one lattice site to a nearest neighboring site. Furthermore, in this model, a particle can only diffuse if it has no lateral nearest or next-nearest neighbors. This causes the particles to form a surface somewhat reminiscent of snowflakes. If a diffusion event can occur, it occurs with propensity D.

Since this is a solid-on-solid model, if a diffusing particle moves to a site that is not just above another particle, then the diffusing particle will fall until it lands on top of another particle. This guarantees no vacancies, and it also means that the three-dimensional *true* lattice, where each cell is either occupied or unoccupied, does not have to be explicitly modeled. Instead, a *computational* lattice is used that has the same lateral dimensions as the cubic true lattice but always contains only a single lattice plane, and the height of the column of particles at cell $(i, j, 0)$ of the true lattice is recorded at cell $(i, j, 0)$ of the lattice used for computation.

There will actually be *two* implementations shown in this example: one using auto-tracking to determine the cells affected by an event, and one using semi-manual tracking.

4.1.1 Using auto-tracking

The code for this example is in the directory `doc/example-code/testFractal` of the ARL KMCThinFilm installation. We will begin by viewing what is effectively the driver file for the simulation code, `testFractal.cpp`, which will show the general outline of the simulation, but leave in a few "blanks" to be filled in, so to speak. To fill in these blanks, we will then turn to the files `EventsAndActions.hpp` and `EventsAndActions.cpp`, where the implementation code lies.

Examining the driver code

The header files needed by `testFractal.cpp` are as follows:

```
#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>

#include "EventsAndActions.hpp"
```

The last of these headers, `EventsAndActions.hpp`, was mentioned above and will be discussed in more detail later. The remaining headers include the definitions for the KMCThinFilm::Simulation class and a wrapper class for a Mersenne Twister random number generator. Here, any available concrete implementation of the KMCThinFilm::RandNumGen class could have been used in place of KMCThinFilm::RandNumGenMT19937. In order to avoid having to type the prefix "`KMCThinFilm::`" repeatedly, the next line is

```
using namespace KMCThinFilm;
```

At the beginning of the `main()` function in `testFractal.cpp`, we have the following hard-coded parameters:

```
  double F = 1, DoverF = 1e5, maxCoverage = 4;
  int domainSize = 256;
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;
```

The first several parameters pertain to the fractal model itself. Parameter `F` is the aforementioned deposition flux F, while `DoverF` is D/F. The parameter `maxCoverage` indicates how many monolayers to deposit, and `domainSize` will be used to set the size of the lattice.

The last two parameters relate more to the computation of the kinetic Monte Carlo simulation. The parameter `seed` is the seed for the random number generator. Parameter `sId` indicates which solver will be used to randomly choose an event at a time step in the simulation.

> **Note**
>
> The parameters are only hard-coded in order to simplify the example. In a more realistic simulation code, one could use, for example, the Program Options library of Boost < `http://www.boost.org`> to input the parameters, especially since Boost is already a dependency of the ARL KMCThinFilm library.

To determine the simulation time needed to deposit the desired number of monolayers, `maxCoverage` is divided by the flux `F`, so the next line of `testFractal.cpp` reads

```
  double approxDepTime = maxCoverage/F;
```

At this point, the simulation itself is initialized. To do this, the parameters needed to initialize the lattice used in the simulation are passed to the constructor of a KMCThinFilm::Simulation object:

```
  LatticeParams latParams;
  latParams.numIntsPerCell = FIntVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;

  Simulation sim(latParams);
```

Here, the size of the array of integers in each lattice cell is set to `FIntVal::SIZE`, which will turn out to be an enumeration constant defined in `EventsAndActions.hpp`. It was mentioned before that `domainSize` will be used to set the size of the lattice. Now here, the lateral size of the lattice has been set to `domainSize` × `domainSize`.

The choice of solver and random number generator for the simulation are now set:

```
  sim.setSolver(sId);

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  sim.setRNG(rng);
```

The order of the above statements matters, since KMCThinFilm::Simulation::setRNG() must be called *after* KMCThinFilm::Simulation::
Here, a "smart" pointer to an instance of a KMCThinFilm::RandNumGen class is passed to our KMCThinFilm::Simulation

object, `sim`. Since this is a shared pointer, `sim` does not exclusively own this pointer. Instead, this pointer deletes itself when there are no more references to it. (Also, if need be, other objects besides `sim` are allowed to make use of this pointer to generate random numbers of their own, but that will not be an issue in this simple example.)

At this point, *possible events* and *periodic actions* may be added to the simulation. We begin by adding a possible deposition event:

```
sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);
sim.addOverLatticeEvent(FOverLatticeEvents::DEPOSITION,
                        F, DepositionExecute);
```

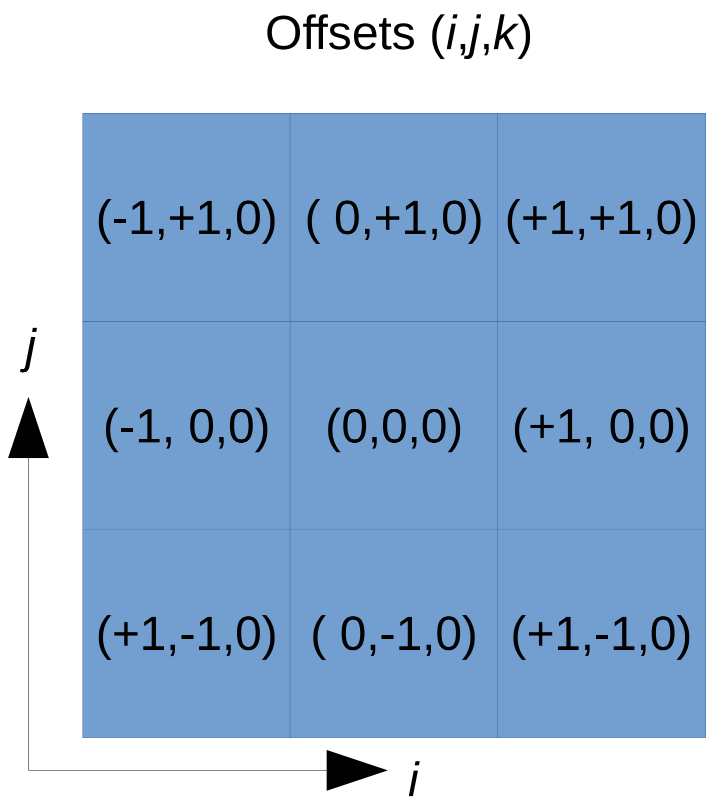
This code adds possible so-called *over-lattice* events, which are events that may originate from some randomly picked site at the top of the lattice, as discussed in the section entitled "Basics of kinetic Monte Carlo simulation as implemented in the ARL KM For example, deposition of a particle can be modeled as an event where a particle appears randomly at the top of the lattice and falls until it lands atop an occupied lattice site.

The member function KMCThinFilm::Simulation::reserveOverLatticeEvents() sets the number of possible over-lattice events that will be added to the simulation and reserves memory for them. Here, the number of such events is `FOver←LatticeEvents::SIZE`, another enumeration constant defined in `EventsAndActions.hpp`. The member function KMCThinFilm::Simulation::addOverLatticeEvent() actually adds the event to the simulation. The three arguments that it takes are an integer label identifying the event, which here is another enumeration constant `FOverLattice←Events::DEPOSITION`; a propensity per unit of in-plane area, that is, the flux *F*; and a function or function object responsible for actually executing the event, which here is `DepositionExecute`. (The actual propensity of the event is the propensity per area scaled by the size of a monolayer of lattice cells, i.e., `latParams.globalPlanarDims[0]` $\times$ `latParams.globalPlanarDims[1]`.)

We now begin to add *cell-centered* events to the simulation. Again, as discussed in the section entitled "Basics of kinetic Monte Carlo sim these events are called such because such events originate in the neighborhood of some lattice cell, and their propensities—unlike those of over-lattice events—*are* affected by the states of cells in that neighborhood. This neighborhood is defined via one or more *offsets* from a central cell, as follows:

```
CellNeighOffsets hopCNO(HopOffset::SIZE);

hopCNO.addOffset(HopOffset::UP,    CellIndsOffset(0,+1,0));
hopCNO.addOffset(HopOffset::DOWN,  CellIndsOffset(0,-1,0));
hopCNO.addOffset(HopOffset::LEFT,  CellIndsOffset(-1,0,0));
hopCNO.addOffset(HopOffset::RIGHT, CellIndsOffset(+1,0,0));

hopCNO.addOffset(HopOffset::RIGHT_UP,   CellIndsOffset(+1,+1,0));
hopCNO.addOffset(HopOffset::RIGHT_DOWN, CellIndsOffset(+1,-1,0));
hopCNO.addOffset(HopOffset::LEFT_UP,    CellIndsOffset(-1,+1,0));
hopCNO.addOffset(HopOffset::LEFT_DOWN,  CellIndsOffset(-1,-1,0));
```

The object `hopCNO` is a container of these offsets. In its constructor, it is passed the number of offsets that it will contain, which here is expressed as another enumeration constant, `HopOffset::SIZE`. The actual offsets are added with the KMCThinFilm::CellNeighOffsets::addOffset(), which takes an integer label greater than zero and less than `HopOffset::SIZE`, and the offset itself. The integer label 0 (which is also `HopOffset::SELF`, a constant that will be seen later) is not used as an argument to KMCThinFilm::CellNeighOffsets::addOffset() because it is reserved for the zero offset, i.e., `CellIndsOffset(0,0,0)`, which is automatically included in a KMCThinFilm::CellNeighOffsets object. The offsets added in the snippet of code above correspond to the positions shown in the figure below:



Since the simulation is on a square lattice, *i* and *j* happen to be orthogonal, and the axis along which index *k* would point is perpendicularly out of the plane. Also, because this is a two-dimensional simulation, the third index of the offset, *k*, is always zero here.

Once the offsets are defined, cell-centered events can then be added:

```
sim.reserveCellCenteredEventGroups(1,FCellCenteredEvents::SIZE);

EventExecutorGroup hopExecs(FCellCenteredEvents::SIZE);
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_LEFT,
                          HoppingExecute(FCellCenteredEvents::HOP_LEFT));
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_RIGHT,
                          HoppingExecute(FCellCenteredEvents::HOP_RIGHT));
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_UP,
                          HoppingExecute(FCellCenteredEvents::HOP_UP));
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_DOWN,
                          HoppingExecute(FCellCenteredEvents::HOP_DOWN));

sim.addCellCenteredEventGroup(1, hopCNO,
                              HoppingPropensity(DoverF*F),
                              hopExecs);
```

The member function KMCThinFilm::Simulation::reserveCellCenteredEventGroups() is somewhat analogous to the member function KMCThinFilm::Simulation::reserveOverLatticeEvents() mentioned before. Unlike over-lattice events, though, similar types of cell-centered events are grouped together. The first argument to this function indicates the number of groups—which is just 1 in this case—while the second argument indicates the total number of individual cell-centered event types, regardless of grouping. Here, this number happens to be the enumeration constant `FCell↩ CenteredEvents::SIZE`.

Before actually adding a group of cell-centered event types, an KMCThinFilm::EventExecutorGroup object must be constructed. This object encapsulates a set of functions or function objects used to execute one of the events in the group of event types, and its constructor takes as an argument the number of function/function objects to be added to the group. Functions and function objects are added to the group in a manner similar to the way offsets are added to a

KMCThinFilm::CellNeighOffsets object. Here, the KMCThinFilm::EventExecutorGroup::addEventExecutor() takes two arguments: an integer label that is greater than or equal to zero and less than `FCellCenteredEvents::SIZE`, and a function/function object satisfying the KMCThinFilm::EventExecutorAutoTrack signature.

A group of cell-centered event types is added by means of the KMCThinFilm::Simulation::addCellCenteredEventGroup() member function. This function takes four arguments: an integer label identifying the event, which is arbitrarily set to one, a KMCThinFilm::CellNeighOffsets object, which here is `hopCNO`, a function or function object used to calculate the propensity of the event, which here is the object `HoppingPropensity(DoverF*F)`, and the aforementioned KMCThinFilm::EventExecutorGroup object. Here, the constructor for the `HoppingPropensity` function object class happens to take an argument equal to the hopping propensity D, but that will not necessarily be the case in general.

Finally, we define a time-periodic action, which here dumps the state of the lattice to a file at regular intervals of simulation time.

```
  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintASCII("snapshot"),
                            0.05*approxDepTime, true);
```

Again, we have a member function for reserving a type of object, followed by a member function that adds the type of object. The argument to KMCThinFilm::Simulation::reserveTimePeriodicActions() is the number of time-periodic actions to be added, and as before, this number is represented via an enumeration constant, `PAction::SIZE`. The member function KMCThinFilm::Simulation::addTimePeriodicAction() has four arguments: an integer label, which here is another enumeration constant `PAction::PRINT`; a function object to execute the periodic action; the period for the action, which here is taken to be $0.05 \times$ `approxDepTime`; and a Boolean flag to indicate if the action should be performed at the end of a simulation run.

After all this setup, the simulation is then actually set to run as follows:

```
  sim.run(approxDepTime);
  sim.removeOverLatticeEvent(FOverLatticeEvents::DEPOSITION);
  sim.run(0.1*approxDepTime);
```

Note that the simulation is run in two stages. In the first stage, there is both diffusion and deposition for a certain amount of time. Then deposition is stopped by removing the deposition event from the simulation. In the second stage, the simulation is run with only diffusion.

(By the way, in this particular fractal model, since particles cannot diffuse if they have any nearest or next nearest neighbors, the simulation soon runs out of particles that can diffuse because the available particles soon acquire neighbors and thus become fixed in place. This causes the simulation to run out of possible events and end before the alloted simulation time is up.)

The general outline of the simulation, as indicated from the driver code, is as follows. First, a simulation object is initialized. Then, possible events and periodic actions are added to the simulation. Finally, the simulation is then run. There are still several "blanks" left. Some of these are the enumeration constants. Others are the details of the functions or function objects used to implement the possible events and periodic actions. For these, we turn to an examination of the implementation code in EventsAndActions.cpp and its associated header file.

Examining the implementation code

We begin with the header file EventsAndActions.hpp. Here we see that the enumerations, rather than being defined directly, are generated from convenience macros defined in MakeEnum.hpp from the ARL KMCThinFilm library. These macros define enumerations associated with adding possible events and actions:

```
KMC_MAKE_ID_ENUM(FOverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(FCellCenteredEvents,
                 HOP_UP,
                 HOP_DOWN,
                 HOP_LEFT,
                 HOP_RIGHT);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);
```

The enumerations defined through these macros all include a constant containing the name "SIZE", which indicates the number of constants in the enumeration (not including SIZE itself): FOverLatticeEvents::SIZE, FCell↩ CenteredEvents::SIZE, and PAction::SIZE.

The following convenience macro operates similarly:

```
KMC_MAKE_LATTICE_INTVAL_ENUM(F, HEIGHT);
```

However, the prefix of the enumerations from this macro is not just the first argument of the macro, but the first argument plus "IntVal, hence why the constant FIntVal::SIZE was seen in the driver code. This macro also defines the constant FIntVal::HEIGHT, which will be seen later in the implementation code.

The final macro invocation is

```
KMC_MAKE_OFFSET_ENUM(HopOffset,
                     UP, DOWN, LEFT, RIGHT,
                     RIGHT_UP, RIGHT_DOWN, LEFT_UP, LEFT_DOWN);
```

This defines the enumeration used to label the offsets between the cell about which an event is centered and the neighboring cells that affect its propensity. Unlike the previously defined enumerations, this one contains *two* special constants. One of them is HopOffset::SIZE, which as before is the number of constants in the enumeration (not including SIZE itself). We have seen this constant used in the driver code as the argument to a constructor of a KMCThinFilm::CellNeighOffsets object. The other is HopOffset::SELF, which identifies CellIndsOffset(0,0,0), the offset corresponding to the cell about which an event is centered. This constant will be seen later in the implementation code.

The rest of the lines of the file EventsAndActions.hpp contain the declarations and definitions for the functions and function objects that define possible events and a periodic action. We now look at these and at their implementations in the file EventsAndActions.cpp.

Here is the declararation for the function performing deposition:

```
void DepositionExecute(const KMCThinFilm::CellInds & ci,
                       const KMCThinFilm::SimulationState & simState,
                       KMCThinFilm::Lattice & lattice);
```

This function declaration satisfies the KMCThinFilm::EventExecutorAutoTrack signature, because it accepts arguments that are references to a KMCThinFilm::CellInds object, a KMCThinFilm::SimulationState object, and a KMCThinFilm::Lattice object, in that order. Furthermore, since this function is used to alter the lattice, the reference to the KMCThinFilm::Lattice object is *not* constant. The implementation of this function is shown below. (It lacks the "KMCThinFilm::" prefix because EventsAndActions.cpp begins with the statement "using namespace KMCThinFilm;".)

```
void DepositionExecute(const CellInds & ci,
                       const SimulationState & simState,
                       Lattice & lattice) {

  int currVal = lattice.getInt(ci, FIntVal::HEIGHT);
  lattice.setInt(ci, FIntVal::HEIGHT, currVal + 1);

}
```

The argument ci represents the indices of a lattice cell. Since the function executes an *over-lattice* event, ci.i and ci.j are random values, and ci.k is one less than the current number of lattice planes—which for this two-dimensional simulation is simply zero. Essentially what is happening is that the column height at random location (ci.i, ci.j, 0) in the lattice is being incremented by one.

In principle, this could have been implemented as class for a function *object*, that is, a class containing an overload of operator(), but here it is an ordinary function for the sake of simplicity. In some later examples, deposition *is* implemented by a function object class.

Note

The simState argument is not used in this case because the execution of this deposition event does not depend upon time. On the other hand, if, for example, deposition were on a rotating substrate, then it might be possible that deposition could occur along a trajectory whose azimuth was $2\pi\omega \times$ simState.elapsedTime(), where ω is the rate of rotation [3].

The function object class `HoppingPropensity` is defined as follows:

```
class HoppingPropensity {
public:
  HoppingPropensity(double D) : D_(D) {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double D_;
};
```

In this class, the private data member `D_` stores the hopping propensity D. Also, this class satisfies the KMCThinFilm::CellCenteredGroup signature, since it contains an overload of `operator()` that takes as arguments a constant reference to a KMCThinFilm::CellNeighProbe object and *non*-constant reference to a `std::vector` of double precision values. This second argument is an output argument that will be the propensities of hopping in various directions. Before an instance of this function object class even has a chance to execute, the output argument will have already been resized so that it has the same number of elements as there are cell-centered events in the group associated with this function object class, and these elements will have been initialized to zero.

Here is the part of the implementation of the function object class that actually calculates the propensity for hopping:

```
void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  int currHeight = cnp.getInt(cnp.getCellToProbe(HopOffset::SELF), FIntVal::HEIGHT);

  if (currHeight > 0) {
    if ((currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT_UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT_DOWN), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT_UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT_DOWN), FIntVal::HEIGHT))) {

      propensityVec[FCellCenteredEvents::HOP_LEFT] =
        propensityVec[FCellCenteredEvents::HOP_RIGHT] =
        propensityVec[FCellCenteredEvents::HOP_UP] =
        propensityVec[FCellCenteredEvents::HOP_DOWN] = D_;
    }
  }

}
```

In propensity calculations, access to the lattice is *indirect* and occurs via the KMCThinFilm::CellNeighProbe object. This object can only access cells that can be returned by KMCThinFilm::CellNeighProbe::getCellToProbe, which takes the integer label of an offset and returns an object that refers to the cell whose indices are the indices of the cell about which the event is centered plus that offset. If the cell indices about which an object is centered are (i, j, k), then `getCellToProbe(HopOffset::UP)` returns an object referring to the cell with indices $(i, j+1, k)$, `getCell↩ToProbe(HopOffset::DOWN)` returns an object referring to the cell with indices $(i, j-1, k)$, and so on. Also, `get↩CellToProbe(HopOffset::SELF)` returns an object referring to the cell about which the event is centered, that is, (i, j, k).

Note

The association of offsets with a KMCThinFilm::CellCenteredGroupPropensities object is done in driver code via calls to KMCThinFilm::Simulation::addCellCenteredEventGroup().

The reason for using such indirect lattice access is that it forces the implementation of cell-centered propensity calculations to use well-defined local neighborhoods about the cell where an event is centered, which makes the incremental update of the set of possible events at each time step possible.

The actual propensity calculation itself is simple. The variable `currHeight` is the height of the column of particles at cell $(\texttt{ci.i}, \texttt{ci.j}, 0)$ in the true lattice. The propensities stored in `propensityVec` are already initially zero, so they only becomes equal to `D_` if

- the cell $(\texttt{ci.i}, \texttt{ci.j}, \texttt{currHeight} - 1)$ in the true lattice is occupied (i.e. the column height stored at cell $(\texttt{ci.i}, \texttt{ci.j}, 0)$ of the computational lattice is nonzero), and

- the lateral neighboring sites of cell $(\mathtt{ci.i}, \mathtt{ci.j}, \mathtt{currHeight} - 1)$ in the true lattice are empty (i.e. the column height stored at cell $(\mathtt{ci.i}, \mathtt{ci.j}, 0)$ in the computational lattice is greater than the column heights stored in the nearest and next-nearest neighbors of that cell).

For convenience and ease of reading, the indices used with `propensityVec` are symbolic enumeration constants that correspond to an event type, i.e. `propensityVec[FCellCenteredEvents::HOP_LEFT]` is the propensity for hopping to the left, `propensityVec[FCellCenteredEvents::HOP_RIGHT]` is the propensity for hopping to the right, and so on.

Note

Lattice indices start from zero, which is why the indices of the topmost particle of a column would be $(\mathtt{ci.i}, \mathtt{ci.j}, \mathtt{currHeight} - 1)$, where `currHeight` is the height of the column of particles.

As a general rule, the propensity of an event centered at a cell should be zero if it cannot happen at that particular cell.

The function object class `HoppingExecute` is defined as follows:

```
class HoppingExecute {
public:
  HoppingExecute(FCellCenteredEvents::Type hopDir);
  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice) const;
private:
  int jump_i_, jump_j_;
};
```

This class satisfies the KMCThinFilm::EventExecutorAutoTrack signature, since it contains an overload of `operator()` that accepts arguments that are references to a KMCThinFilm::CellInds object, a KMCThinFilm::SimulationState object, and a KMCThinFilm::Lattice object, in that order. Again, since this operator is used to alter the lattice, the reference to the KMCThinFilm::Lattice object is *not* constant. The constructor for this class is as follows:

```
HoppingExecute::HoppingExecute(FCellCenteredEvents::Type hopDir) {

  switch (hopDir) {
  case FCellCenteredEvents::HOP_UP:
    jump_i_ = 0;
    jump_j_ = 1;
    break;
  case FCellCenteredEvents::HOP_DOWN:
    jump_i_ = 0;
    jump_j_ = -1;
    break;
  case FCellCenteredEvents::HOP_LEFT:
    jump_i_ = -1;
    jump_j_ = 0;
    break;
  case FCellCenteredEvents::HOP_RIGHT:
    jump_i_ = 1;
    jump_j_ = 0;
    break;
  default:
    exitWithMsg("Bad hop direction!");
  }
}
```

One can see from this constructor that the function object it instantiates can do one of four possible events, namely a hop to one of four nearest neighboring cells.

Here is the part of the implementation of the function object class that actually executes the event:

```
void HoppingExecute::operator()(const CellInds & ci,
                                const SimulationState & simState,
                                Lattice & lattice) const {

  CellInds ciTo(ci.i + jump_i_, ci.j + jump_j_, ci.k);

  int currFrom = lattice.getInt(ci, FIntVal::HEIGHT);
  int currTo = lattice.getInt(ciTo, FIntVal::HEIGHT);

  lattice.setInt(ci, FIntVal::HEIGHT, currFrom - 1);
  lattice.setInt(ciTo, FIntVal::HEIGHT, currTo + 1);
}
```

Here, `ci` is the set of indices of a cell where the cell-centered event can happen, rather than just some random location at the top of the lattice as it is for an over-lattice event. Since this particular model is two-dimensional, `ci.k` is always zero. The hopping of a particle from cell $(\texttt{ci.i}, \texttt{ci.j}, \texttt{currFrom} - 1)$ in the true lattice to $(\texttt{ciTo.i}, \texttt{ciTo.j}, \texttt{currTo})$ is represented in the computational lattice as a decrement of the column height at `ci` and an increment of the column height at `ciTo`.

Finally, the function object class `PrintASCII`, which prints the state of the lattice to a text file, is defined as follows:

```
class PrintASCII {
public:
  PrintASCII(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};
```

This code satisfies the function signature KMCThinFilm::PeriodicAction, since it takes as arguments references to a KMCThinFilm::SimulationState object and a KMCThinFilm::Lattice object. While this particular periodic action only dumps the state of the lattice, it is possible for a periodic action to change the lattice, which is why the reference to the KMCThinFilm::Lattice object is *not* constant.

Here is the implementation of the function object class:

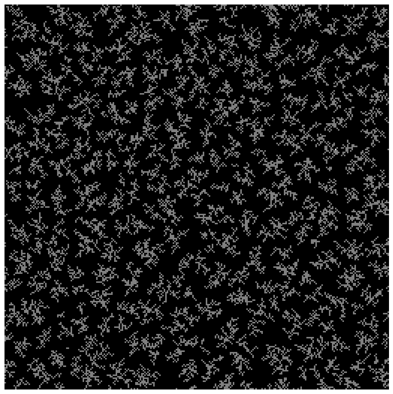
```
void PrintASCII::operator()(const SimulationState & simState, Lattice & lattice) {

  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".dat";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int iminGlobal = localPlanarBBox.imin;
  int jminGlobal = localPlanarBBox.jmin;

  // "P1" here is short for "Plus 1".
  int imaxP1Global = localPlanarBBox.imaxP1;
  int jmaxP1Global = localPlanarBBox.jmaxP1;

  outFile << "# " << iminGlobal << " " << imaxP1Global << " " << jminGlobal << " " << jmaxP1Global
          << " time:" << simState.elapsedTime() << "\n";

  CellInds ci; ci.k = 0;
  for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
    for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {
      outFile << ci.i << " " << ci.j << " "
              << lattice.getInt(ci, FIntVal::HEIGHT) << "\n";
    }
  }

  outFile.close();
}
```

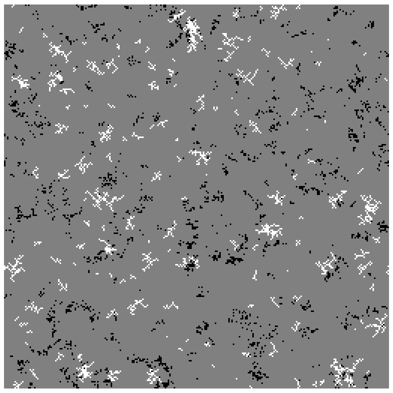
This code prints a header that indicates the minimum and maximum values of in-plane lattice cell indices *i* and *j*, and the elapsed time. It then prints the in-plane indices of each cell and the column height stored at that cell.

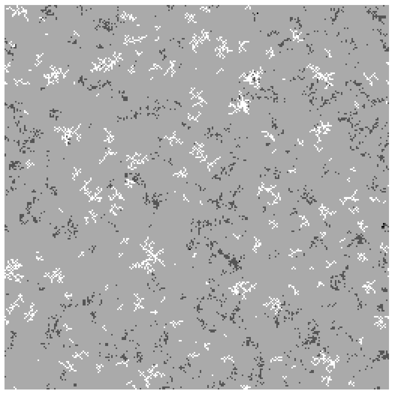
Also, this code is written to in such a way as to facilitate parallelization. Since this is a serial code, KMCThinFilm::Lattice::getGlobalPlan could have been used in place of KMCThinFilm::Lattice::getLocalPlanarBBox() to obtain the minimum and maximum values of the in-plane indices. However, the code as written above will also work in parallel to produce a dump to a file of the part of the lattice that a given processor owns, which would *not* be true if KMCThinFilm::Lattice::getGlobalPlanarBBox() had been used.

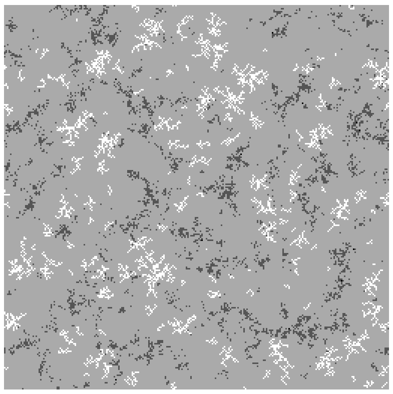
Results

An overhead view of the surface of the "fractal" thin film at various simulation times *t*, from 0.20 units to its maximum value of 4.0 units, is shown below. Within a given overhead view, black represents the smallest possible height of a column of particles, and white represents the largest.


t = 0.20


t = 1.0


t = 2.0


t = 4.0

4.1.2 Using semi-manual tracking

The code for this example, which is in the directory `doc/example-code/testFractal_semi_manual_track` of the ARL KMCThinFilm installation, is not much different from that used for previous section "Using auto-tracking."

The source code for the driver file, `testFractal.cpp`, mainly differs in how event types are added to the system. The following code is used to add the over-lattice event type of deposition to the simulation:

```
std::vector<CellNeighOffsets> tmpExecCNO;
tmpExecCNO.reserve(1);

sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);

tmpExecCNO.push_back(CellNeighOffsets(1));
sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);
sim.addOverLatticeEvent(FOverLatticeEvents::DEPOSITION,
                        F, DepositionExecute, tmpExecCNO);
```

The member function KMCThinFilm::Simulation::reserveOverLatticeEvents() works the same as before. What differs is the member function KMCThinFilm::Simulation::addOverLatticeEvent(), which now has an additional argument, `tmpExecCNO`, a `std::vector` of KMCThinFilm::CellNeighOffsets objects. For a given deposition event, only one lattice cell is directly changed, so `tmpExecCNO` has only one element, which in turn contains only one

KMCThinFilm::CellNeighOffsets instance, which contains only the offset that is present in every such instance, $(0,0,0)$. The third argument of KMCThinFilm::Simulation::addOverLatticeEvent() is also different, since it satisfies the KMCThinFilm::EventExecutorSemiManualTrack signature rather than the KMCThinFilm::EventExecutorAutoTrack signature. The following code is used to add a group of cell-centered event types:

```
sim.reserveCellCenteredEventGroups(1,FCellCenteredEvents::SIZE);

EventExecutorGroup hopExecs(FCellCenteredEvents::SIZE);
tmpExecCNO.clear();
tmpExecCNO.push_back(CellNeighOffsets(2));
tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::LEFT));

hopExecs.addEventExecutor(FCellCenteredEvents::HOP_LEFT,
                          HoppingExecute(), tmpExecCNO);

tmpExecCNO.back().resetOffsets(2);
tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::RIGHT));
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_RIGHT,
                          HoppingExecute(), tmpExecCNO);

tmpExecCNO.back().resetOffsets(2);
tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::UP));
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_UP,
                          HoppingExecute(), tmpExecCNO);

tmpExecCNO.back().resetOffsets(2);
tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::DOWN));
hopExecs.addEventExecutor(FCellCenteredEvents::HOP_DOWN,
                          HoppingExecute(), tmpExecCNO);

sim.addCellCenteredEventGroup(1, hopCNO,
                              HoppingPropensity(DoverF*F),
                              hopExecs);
```

Here, the member functions KMCThinFilm::Simulation::reserveCellCenteredEventGroups() and KMCThinFilm::Simulation::addCellCe
are the same as before. What differs is the way entries are added to the KMCThinFilm::EventExecutorGroup instance. Here, KMCThinFilm::EventExecutorGroup::addEventExecutor() takes an additional argument, the vector `tmpExec↩ CNO`. This vector still contains a single element each time it is used, but since a hopping event changes two lattice cells, this single element contains a KMCThinFilm::CellNeighOffsets instance with two offsets: the ever-present offset $(0,0,0)$, corresponding to the cell from which a particle hops, and an offset corresponding to the cell to which a particle hops, which depends on the hopping direction. The second argument to KMCThinFilm::EventExecutorGroup::addEventExecutor() is also different, since it satisfies the KMCThinFilm::EventExecutorSemiManualTrack signature rather than the KMCThinFilm::EventEx
signature.

In `EventsAndActions.hpp`, here is the declararation for the function performing deposition:

```
void DepositionExecute(const KMCThinFilm::CellInds & ci,
                       const KMCThinFilm::SimulationState & simState,
                       const KMCThinFilm::Lattice & lattice,
                       std::vector<KMCThinFilm::CellsToChange> & ctcVec);
```

This function declaration satisfies the KMCThinFilm::EventExecutorSemiManualTrack signature, because it accepts arguments that are references to a KMCThinFilm::CellInds object, a KMCThinFilm::SimulationState object, a KMCThinFilm::Lattice object, and a `std::vector` of KMCThinFilm::CellsToChange objects, in that order. Here, the reference to the KMCThinFilm::Lattice object *is* constant, since changes to the lattice are forced to occur via KMCThinFilm::CellsToChange objects. The implementation of this function is shown below.

```
void DepositionExecute(const CellInds & ci,
                       const SimulationState & simState,
                       const KMCThinFilm::Lattice & lattice,
                       std::vector<KMCThinFilm::CellsToChange> & ctcVec) {

  CellsToChange & ctc = ctcVec[0];
  ctc.setCenter(ci);

  int currVal = lattice.getInt(ci, FIntVal::HEIGHT);
  ctc.setInt(0, FIntVal::HEIGHT, currVal + 1);
}
```

In general, a KMCThinFilm::CellsToChange object contains a set of offsets that specifies the relative positions of changed cells, and the KMCThinFilm::CellsToChange::setCenter() member function essentially provides an “anchor” used to translate these relative positions to absolute ones. Given a center (i_0, j_0, k_0) and offsets $(0,0,0)$, (o_i^1, o_j^1, o_k^1), etc., the absolute coordinates of the cells to change become (i_0, j_0, k_0), $(i_0 + o^1_i, j_0 + o^1_j, k_0 + o^1_k)$, etc. Because of how this over-lattice event was set up in the driver code, the KMCThinFilm::CellsToChange object here happens to contain only one offset, $(0,0,0)$, the indices of the cell to be changed are just (`ci.i`, `ci.j`, `ci.k`). The member function KMCThinFilm::CellsToChange::setInt() performs the actual change to the cell. This member function is analogous to KMCThinFilm::Lattice::setInt(), except that its first argument is an integer ID corresponding to one of the offsets stored in a KMCThinFilm::CellsToChange object, rather than a set of cell indices. An integer ID of zero always corresponds to the offset $(0,0,0)$, and accordingly to the indices of the center, which here is (`ci.i`, `ci.j`, `ci.k`).

The function object class `HoppingExecute` is now defined as follows:

```
class HoppingExecute {
public:
  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec) const;
};
```

Since semi-manual tracking is used, this class satisfies the KMCThinFilm::EventExecutorSemiManualTrack signature, rather than the KMCThinFilm::EventExecutorAutoTrack signature. Unlike the previous implementation of the `HoppingExecute` class, this one has no private members to indicate the hopping direction, because the hopping direction is determined from the `ctcVec` argument. Here is the part of the implementation of the function object class that actually executes the event:

```
void HoppingExecute::operator()(const CellInds & ci,
                                const SimulationState & simState,
                                const KMCThinFilm::Lattice & lattice,
                                std::vector<KMCThinFilm::CellsToChange> & ctcVec) const {

  CellsToChange & ctc = ctcVec[0];
  ctc.setCenter(ci);

  int currFrom = lattice.getInt(ci, FIntVal::HEIGHT);
  int currTo = ctc.getInt(1, FIntVal::HEIGHT);

  ctc.setInt(0, FIntVal::HEIGHT, currFrom - 1);
  ctc.setInt(1, FIntVal::HEIGHT, currTo + 1);
}
```

Here, `ctcVec` contains a KMCThinFilm::CellsToChange object with *two* offsets: $(0,0,0)$, which corresponds to the cell from which a particle hops, and an offset that corresponds to the cell to which a particle hops, which depends on the hopping direction. Here we see KMCThinFilm::CellsToChange::getInt() as well as KMCThinFilm::Lattice::getInt(). The former function is analogous to the latter, with the former function using an integer ID—corresponding to one of the offsets in a KMCThinFilm::CellsToChange object—instead of cell indices. The integer ID of offset $(0,0,0)$ is, as always,

1. The integer ID of the other offset can be inferred from the portion of the driver code where the offset was set. For example, for rightward hopping, the code to set the offset is

```
  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::RIGHT));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_RIGHT,
                            HoppingExecute(), tmpExecCNO);
```

The first argument of KMCThinFilm::CellNeighOffsets::addOffset(), which here happens to be 1, is the integer ID for the offset corresponding to the cell to which a particle hops.

The rest of the code of this implementation of a fractal solid-on-solid model is the same as it is in the implementation using auto-tracking.

4.2 Example: Parallelizing an implementation of a “fractal” solid-on-solid model

The code for this example is in the directory `doc/example-code/testFractal_parallel` of the ARL KMCThinFilm installation, and it is almost the same as that in `doc/example-code/testFractal_semi_manual_track`. Some of the most pertinent differences between the two codes will be discussed here. The parallelized code is actually written so that it can be compiled to either a serial application or an MPI application.

The actual process of parallelizing the code described in the previous section is actually straightforward. Only the driver code, in `testFractal.cpp` differs from its serial counterpart, though if the code for dumping the state of the lattice to a file had been different, this might not have been the case. The most obvious changes are near the beginning and the end of the function `main()`, where `MPI_Init()` and `MPI_Finalize() are called`:

```
#if KMC_PARALLEL
  MPI_Init(&argc, &argv);
#endif

#if KMC_PARALLEL
  MPI_Finalize();
#endif
```

In order for the driver code to be compilable as serial code, these MPI calls are wrapped by the `#if KMC_PARALLEL` and `#endif` preprocessor directives. Note that `#if` and **not** `#ifdef` is used here.

While most of the parameters near the beginning of `main()` are the same as in the serial version of the code, there is one minor change and one addition, which can be seen below.

```
  double F = 1, DoverF = 1e5, maxCoverage = 4;
  int domainSize = 256;
  unsigned int seedGlobal = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  TimeIncr::SchemeVars schemeVars;
  schemeVars.setSchemeName(TimeIncr::SchemeName::MAX_AVG_PROPENSITY_PER_POSS_EVENT);

  schemeVars.setSchemeParam(TimeIncr::SchemeParam::NSTOP,
#ifndef BAD_NSTOP
                            1
#else
                            100
#endif
                            );
```

The minor change is just a renaming of the variable `seed` to `seedGlobal`, which reflects a change in how the random number generator is initialized in parallel. The addition is the presence of an instance of KMCThinFilm::TimeIncr::SchemeVars, which is used to set the choice of parallel time stepping scheme and any parameter of it. A choice of such a scheme is required to run a parallel Kinetic Monte Carlo simulation. (Parallel time stepping is discussed in the section entitled "Parallel approximate Kinetic Monte Carlo algorithm.") The preprocessor variable `BAD_NSTOP` is used in this example to determine whether the parameter n_{stop} for the parallel time stepping is set to a reasonable value for this simulation, or to one that will produce artifacts.) Unlike the calls to MPI functions, this addition does not have to be placed between `#if KMC_PARALLEL` and `#endif` directives. It simply has no effect in a serial simulation. This is a common practice in the ARL KMCThinFilm library. Unless a part of its programming interface explicitly involves an MPI data structure, such as `MPI_Comm`, it will be both legal and harmless in serial code.

Most of the initialization of the simulation, which is shown here,

```
  LatticeParams latParams;
  latParams.numIntsPerCell = FIntVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;
  latParams.ghostExtent[0] = latParams.ghostExtent[1] = 1;

#ifdef USE_COMPACT_DECOMP
  latParams.parallelDecomp = LatticeParams::COMPACT;
#endif

  Simulation sim(latParams);
```

is the same as in the serial simulation, except for a crucial addition: specifying the values of the `ghostExtent` member of the KMCThinFilm::LatticeParams object, which indicate the extent of the ghost region at the edge of each lattice plane. As with the reference to KMCThinFilm::TimeIncr::SchemeVars, there is no need to wrap the statement `lat↩ Params.ghostExtent[0] = latParams.ghostExtent[1] = 1` within `#if KMC_PARALLEL` and `#endif`, since it simply has no effect in a serial simulation. The preprocessor variable `USE_COMPACT_DECOMP` is used in this example to determine whether compact parallel decomposition is used instead of the default row-based decomposition. (See the section entitled "Parallel approximate Kinetic Monte Carlo algorithm""" for a description of methods of decomposition.)

Also, in a parallel simulation, the KMCThinFilm::LatticeParams object has an additional member of type `MPI_Comm`, `latticeCommInitial`. By default, this is simply set to `MPI_COMM_WORLD`, so there is typically no need to explicitly specify it. However, since it is not defined in the serial version of the ARL KMCThinFilm library, if there had been any explicit reference to it in the driver code, it would have needed to have been surrounded by `#if KMC_PARALLEL` and `#endif`.

Setting the random number generator becomes slightly more complicated in a parallel simulation, as shown below.

```
#if KMC_PARALLEL
  RandNumGenSharedPtr rng(new RandNumGenDCMT(sim.procID(),
                                             seedGlobal,
                                             123*sim.procID() + 456,
                                             RandNumGenDCMT::P521));
#else
  RandNumGenSharedPtr rng(new RandNumGenMT19937(seedGlobal));
#endif

  sim.setRNG(rng);
```

Here, in the serial version of the code, the KMCThinFilm::RandNumGenMT19937 is used, while in parallel, KMCThinFilm::RandNumC
is used. In principle, KMCThinFilm::RandNumGenDCMT could be have been used for both parallel and serial, but KMCThinFilm::RandNumGenMT19937 is restricted to serial use. One should also note that the constructor to parallel random number generators such as KMCThinFilm::RandNumGenDCMT takes the MPI rank or processor ID as an argument. Here, this is given by KMCThinFilm::Simulation::procID().

Once KMCThinFilm::Simulation::setSolver() has been called, the parallel time step scheme can now be set as follows.

```
  sim.setTimeIncrScheme(schemeVars);
```

Again, there is no need to surround this by `#if KMC_PARALLEL` and `#endif`, since it is just a no-op in serial code.

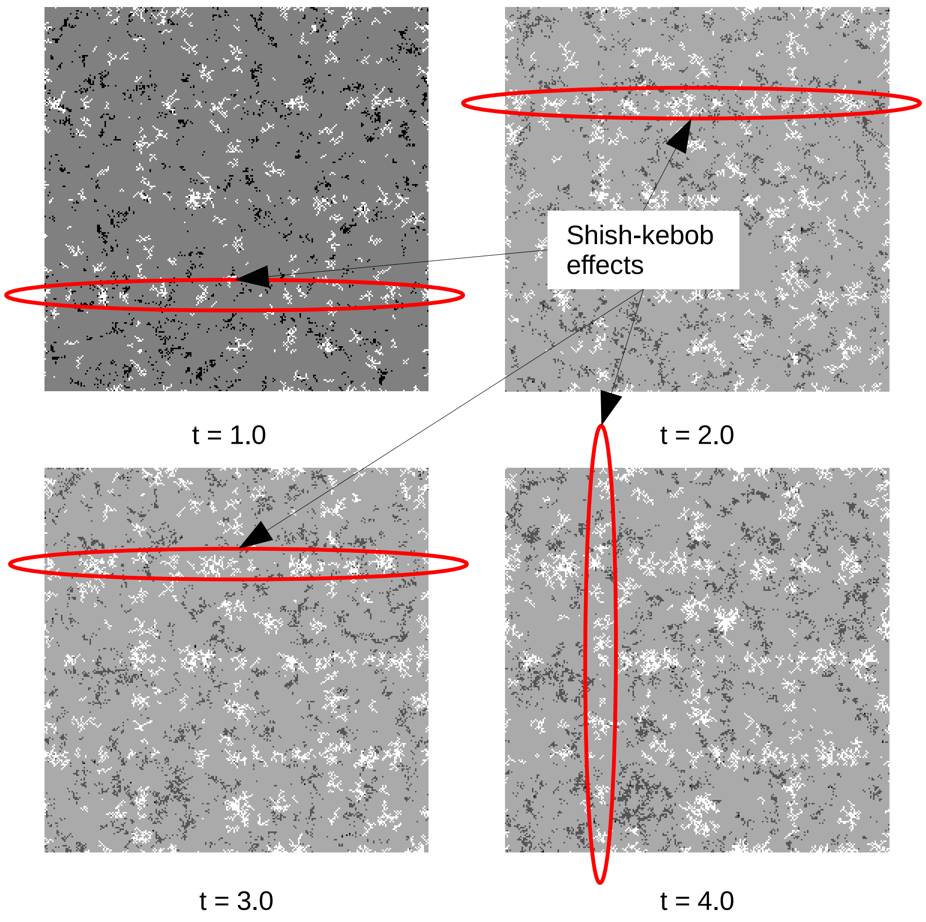
Finally, there is a minor but needed change in the setup for dumping the state of the lattice to files:

```
  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintASCII("outFile_ProcCoords" +
                                       boost::lexical_cast<std::string>(sim.commCoord(0)) + "_" +
                                       boost::lexical_cast<std::string>(sim.commCoord(1)) + "_snapshot"),
                            0.05*approxDepTime, true);
```

Whereas before in the purely serial code, the root of the names of the dump files was just "`snapshot`", now it is a string that is unique to each MPI process. Here, each process writes to its own file, which is not always optimal but will suffice here. In serial code, `sim.commCoord(0)` and `sim.commCoord(1)` both return an integer value of zero.

When running the parallelized code, two things may be noticed. First, for such a small simulation, it is likely to run *slower* than the corresponding serial code. Second, whereas the serial code will simply quit when it runs out of possible events to execute, the parallel code will simply continue until the specified simulation time is reached.

The results of the parallel simulations should be about the same as those in the serial simulation, provided that the parameters for the parallel time stepping algorithm are set appropriately. The following results show what may happen if they are not set correctly:



These snapshots of the simulated surface, at simulations times t 1.0 through 4.0, contain artifacts that have been described as a “shish-kebob” effect [7], where parts of the growing simulated film concentrate on sector boundaries. Some instances of these artifacts have been circled in red. The presence of these artifacts indicates that the size of the parallel time steps is too large.

4.3 Example: Implementations of a ballistic deposition model

While the examples above showcased several features of the ARL KMCThinFilm library, they were still two-dimensional. This particular example is meant to show what is needed for a simulation that requires a three-dimensional lattice. It will also show a few other features of the ARL KMCThinFilm library.

What follows is a pair of example implementations that involve ballistic deposition on a cubic lattice and a contrived cell-centered event to change the “color” of a lattice cell, which here is a floating-point number between zero and one. In ballistic deposition, a depositing particle travels in a straight line until it is close enough to another particle to “stick” to it. The result is a deposit that can have several vacancies and overhangs. For simplicity, the depositing particles travel straight downward, which allows the use of simplified algorithms that obviate the need to explicitly model the actual travel of the particle. The propensity of the cell-centered event is proportional to the number of occupied nearest neighbors of a lattice cell, including those above and below the cell, and the execution of the event changes the color to an average of the the current cell color and the colors of its occupied nearest neighbors.

In the simplified deposition algorithm [6], there is both a cubic lattice, whose cells may either be occupied or unoccupied, and an auxiliary two-dimensional array of active zone height coordinates, which has the same lateral dimensions as the cubic lattice and may be denoted as $h_a(i,j)$. Before any deposition events begin, the elements of

$h_a(i,j)$ are initialized to zero. When a deposition event is chosen, in-plane lattice coordinates (i,j) are randomly chosen. Then, a particle is placed at lattice site $(i,j,h_a(i,j))$, and for each in-plane coordinate $(i_{neigh}, j_{neigh}) \in \{(i+1,j),(i-1,j),(i,j+1),(i,j-1)\}$, the value of $h_a(i_{neigh}, j_{neigh})$ is checked to see if is less than $h_a(i,j)$. If it is, then it is increased it to the value of $h_a(i,j)$. After this, $h_a(i,j)$ is incremented by one.

4.3.1 First implementation

The code for this implementation is in `doc/example-code/testBallisticDep1` of the installation directory of the ARL KMCThinFilm library. In this implementation, auto-tracking has been used. Also, the values of the auxiliary array $h_a(i,j)$ are stored within the first plane of the lattice. Accordingly, in `EventsAndActions.hpp`, the following enumeration,

```
KMC_MAKE_LATTICE_INTVAL_ENUM(BD, IS_OCCUPIED, ACTIVE_ZONE_HEIGHT);

KMC_MAKE_LATTICE_FLOATVAL_ENUM(BD, COLOR);
```

has been defined. The definition for the function object class defining deposition is as follows:

```
class DepositionExecute {
public:
  DepositionExecute();

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);
private:
  std::vector<KMCThinFilm::CellIndsOffset> neighOffsets_;
};
```

The vector `neighOffsets_` will be used to find the in-plane neighboring lattice cells of the randomly chosen deposition site. This vector is now initialized in the constructor for the deposition function object:

```
DepositionExecute::DepositionExecute() {

  neighOffsets_.reserve(4);
  neighOffsets_.push_back(CellIndsOffset( 0,-1));
  neighOffsets_.push_back(CellIndsOffset( 0,+1));
  neighOffsets_.push_back(CellIndsOffset(-1, 0));
  neighOffsets_.push_back(CellIndsOffset(+1, 0));

}
```

The implementation of the operator that performs the actual execution of the deposition event is shown below.

```
void DepositionExecute::operator()(const CellInds & ci,
                                   const SimulationState & simState,
                                   Lattice & lattice) {

  /* Ballistic deposition algorithm for cubic lattices from Meakin and
     Krug, Physical Review A, vol. 46, num. 6, pp. 3390-3399 (1992).*/

  CellInds ciInPlane(ci.i, ci.j, 0);

  int kDepAtom = lattice.getInt(ciInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT);

  CellInds ciTo(ci.i, ci.j, kDepAtom);

  lattice.addPlanes(ciTo.k - ci.k);
  lattice.setInt(ciTo, BDIntVal::IS_OCCUPIED, 1);

  for (std::vector<CellIndsOffset>::const_iterator offsetItr = neighOffsets_.begin(),
         offsetItrEnd = neighOffsets_.end(); offsetItr != offsetItrEnd; ++offsetItr) {

    CellInds neighInPlane = ciInPlane + *offsetItr;

    if (lattice.getInt(neighInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT) < kDepAtom) {
      lattice.setInt(neighInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT, kDepAtom);
    }
  }

  lattice.setInt(ciInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT, kDepAtom + 1);

}
```

Again, the argument `ci` represents the indices of a lattice cell. The operator executes an *over-lattice* event, so `ci.i` and `ci.j` are already random values. This takes care of the part of Meakin's algorithm where a random pair of in-plane indices is chosen. The variable kDepAtom is used to store the current active zone height for the in-plane indices (ci.i, ci.j). Since the active zone heights are stored only within the *first* plane of the lattice, ciInPlane.k is zero.

Before placing a particle in the lattice cell at ciTo, or (ci.i, ci.j, kDepAtom), it must be ensured that this cell is actually available in the computational lattice. This is what the call to KMCThinFilm::Lattice::addPlanes() is for. The argument to this member function, if positive, is the number of lattice planes to add. Now ci.k is initially one less than the maximum number of lattice planes, that is, the maximum possible value of the third lattice coordinate, k. Accordingly, if ciTo.k − ci.k is positive, then it is the number of planes that would need to be added to ensure that ciTo is a valid set of indices. If it is zero or negative, then no planes need to be added, and then the call to KMCThinFilm::Lattice::addPlanes() will simply do nothing. Once the call has been made, KMCThinFilm::Lattice::setInt() can be safely called.

The remainder of the implementation of the operator performing deposition, from the for loop onward, simply updates the values of $h_a(i, j)$. The main feature of interest in this remaining part is the "+" operator applied to ciInPlane and ∗offsetItr. The meaning of this operator is such that the statement

```
CellInds neighInPlane = ciInPlane + *offsetItr;
```

is equivalent to

```
CellInds neighInPlane(ciInPlane.i + offsetItr->i,
                      ciInPlane.j + offsetItr->j,
                      ciInPlane.k + offsetItr->k);
```

This operator is not commutative; the offset must be its second operand.

Now normally, when KMCThinFilm::Lattice::addPlanes() is called, the cells that it creates have the quantities associated with them (e.g. those labeled as BDIntVal::IS_OCCUPIED, BDIntVal::ACTIVE_ZONE_HEIGHT, and BDFloat↩ Val::COLOR) all set to zero. However, this can be changed, and has been changed for this example. In the initialization of the simulation done via these lines in testBallisticDep.cpp,

```
  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  LatticeParams latParams;
  latParams.numIntsPerCell = BDIntVal::SIZE;
  latParams.numFloatsPerCell = BDFloatVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;
  latParams.setEmptyCellVals = SetEmptyCellWithRandColor(rng);
  latParams.numPlanesToReserve = 100;

  Simulation sim(latParams);
```

a special function object, an instance of SetEmptyCellWithRandColor, is used to initialize an empty cell. This function object is defined and implemented in the function files InitLattice.hpp and InitLattice.cpp, respectively, which are shown below:

```
#ifndef INIT_LATTICE_HPP
#define INIT_LATTICE_HPP

#include <KMCThinFilm/Lattice.hpp>
#include <KMCThinFilm/RandNumGen.hpp>

class SetEmptyCellWithRandColor {
public:
  SetEmptyCellWithRandColor(KMCThinFilm::RandNumGenSharedPtr rng)
    : rng_(rng)
  {}

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<int> & emptyIntVals,
                  std::vector<double> & emptyFloatVals);
private:
  KMCThinFilm::RandNumGenSharedPtr rng_;
};

#endif /* INIT_LATTICE_HPP */
```

```
#include "InitLattice.hpp"
#include "EventsAndActions.hpp"

using namespace KMCThinFilm;

void SetEmptyCellWithRandColor::operator()(const CellInds & ci,
                                           const Lattice & lattice,
                                           std::vector<int> & emptyIntVals,
                                           std::vector<double> & emptyFloatVals) {

  emptyFloatVals[BDFloatVal::COLOR] = rng_->getNumInOpenIntervalFrom0To1();

}
```

Here, even an "empty" lattice cell has a color associated with it, which is randomly determined. Note that there is no need to resize the vectors used as output arguments of `SetEmptyCellWithRandColor::operator()`, since the sizes of these vectors have already been set to the number of integer and floating-point quantities, respectively, at each lattice cell.

Since multiple lattice planes will be added during the course of the simulation, `latParams.numPlanesToReserve` is set to roughly the number of planes that might be added. This reserves space in memory for those planes but does not add them to the lattice outright. Actually adding the planes is done by KMCThinFilm::Lattice::addPlanes(). `lat↩ Params.numPlanesToReserve` is *not* a hard limit on the number of lattice planes that may be added; it merely affects performance.

Note

While this example of a means of initializing an empty lattice is contrived, a not-so contrived example would be a case where the lattice is distorted and the current coordinates of a particle, which may be near rather than exactly at a lattice site, are stored at a lattice cell. In a new empty cell, it would make sense for these coordinates to be initialized not to zero, but rather to some function of the lattice indices (i, j, k), e.g. $\mathbf{a}_i i + \mathbf{a}_j j + \mathbf{a}_k k + \mathbf{b}$, where $\mathbf{a}_i$, $\mathbf{a}_j$, and $\mathbf{a}_k$ are primitive lattice vectors and $\mathbf{b}$ is a basis vector.

The enumeration associated with the offsets used in the function object classes involved in the color change in a lattice cell is as follows:

```
KMC_MAKE_OFFSET_ENUM(MIX_OFFSET,
                     NORTH, SOUTH, WEST, EAST /* First four neighbors are lateral */,
                     UP, DOWN);
```

The definition for the function object class defining the propensity for the color change in a lattice cell is

```
class ColorMixPropensity {
public:
  ColorMixPropensity(double mixPropPerNeighbor)
    : mixPropPerNeighbor_(mixPropPerNeighbor)
  {}

  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double mixPropPerNeighbor_;
};
```

and its implementation is

```
void ColorMixPropensity::operator()(const KMCThinFilm::CellNeighProbe & cnp,
                                    std::vector<double> & propensityVec) const {

  if (cnp.getInt(cnp.getCellToProbe(MIX_OFFSET::SELF), BDIntVal::IS_OCCUPIED)) {

    int numNeighs = 0;

    // Visiting four lateral neighbors
    for (int whichOffset = 1; whichOffset <= 4; ++whichOffset) {

      if (cnp.getInt(cnp.getCellToProbe(whichOffset), BDIntVal::IS_OCCUPIED)) {
        ++numNeighs;
      }

    }

    CellToProbe downCell = cnp.getCellToProbe(MIX_OFFSET::DOWN);
    if (cnp.belowLatticeBottom(downCell) || cnp.getInt(downCell, BDIntVal::IS_OCCUPIED)) {
      ++numNeighs;
    }

    CellToProbe upCell = cnp.getCellToProbe(MIX_OFFSET::UP);
    if ((!cnp.exceedsLatticeHeight(upCell)) && cnp.getInt(upCell, BDIntVal::IS_OCCUPIED)) {
      ++numNeighs;
    }

    propensityVec[CellCenteredEvents::COLOR_MIXING] = mixPropPerNeighbor_*numNeighs;

  }
}
```

There are a couple new things to note.

First, the contributions of the lateral nearest neighboring cells are determined by looping over the *numeric* labels for the offsets, 1 through 4, rather than the symbolic constants `MIX_OFFSET::NORTH`, `MIX_OFFSET::SOUTH`, `MIX_OFFSET↩ ::WEST`, and `MIX_OFFSET::EAST`. This is perfectly legal; the N enumeration constants listed in the arguments of KMC_MAKE_OFFSET_ENUM will correspond respectively to the numbers 1 through N. (Note that the number zero corresponds to the offset `MIX_OFFSET::SELF`.) There is a tradeoff here. Iterating over *numeric* labels for the offsets may be less self-documenting, but using the symbolic enumeration constants may involve more "cut-and-paste" code.

Second, when probing the non-lateral neighbors, the member functions KMCThinFilm::CellNeighProbe::belowLatticeBottom() and KMCThinFilm::CellNeighProbe::exceedsLatticeHeight() are used to ensure that calls to KMCThinFilm::CellNeighProbe::getInt() are only applied to KMCThinFilm::CellToProbe objects with valid indices (due to the short-circuit evaluation of operators || and &&). In two-dimensional serial simulations, in-plane indices i and j are always valid, because they are wrapped due to periodic boundary conditions, and in two-dimensional parallel simulations, in-plane indices i and j should always be valid if the size of the ghost regions of the lattice has been properly set. In three-dimensional simulations, index k could easily be set to an invalid value, either a negative value—referring to a cell effectively below the computational lattice—or a value that is greater or equal to the number of lattice planes—referring to a cell effectively above the computational lattice. Note that cells below the lattice are treated differently than those above it. Here, if cell indices point to a cell that would be below the computational lattice, `numNeighs` is incremented because such a cell is understood as belonging to the substrate onto which a film is being deposited. If cell indices point to a cell that would be above the computational lattice, `numNeighs` is not incremented, since such a cell is understood as belonging to the empty space above the film.

The definition for the function object class that would execute color mixing is as follows:

```
class ColorMixExecute {
public:
  ColorMixExecute(KMCThinFilm::RandNumGenSharedPtr rng_,
                  const KMCThinFilm::CellNeighOffsets * mixCNO,
                  int * numMixes);

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);
private:
  KMCThinFilm::RandNumGenSharedPtr rng_;
  const KMCThinFilm::CellNeighOffsets * mixCNO_;
  int * numMixes_;
};
```

Private member `mixCNO_` is simply a pointer to the same set of offsets used to calculate the propensity for mixing, as seen in the part of the driver code, `testBallisticDep.cpp`, where cell-centered event types are added to the simulation:

```
  CellNeighOffsets mixCNO(MIX_OFFSET::SIZE);
  mixCNO.addOffset(MIX_OFFSET::NORTH, CellIndsOffset(+1, 0, 0));
  mixCNO.addOffset(MIX_OFFSET::SOUTH, CellIndsOffset(-1, 0, 0));
  mixCNO.addOffset(MIX_OFFSET::WEST,  CellIndsOffset( 0,-1, 0));
  mixCNO.addOffset(MIX_OFFSET::EAST,  CellIndsOffset( 0,+1, 0));
  mixCNO.addOffset(MIX_OFFSET::UP,    CellIndsOffset( 0, 0,+1));
  mixCNO.addOffset(MIX_OFFSET::DOWN,  CellIndsOffset( 0, 0,-1));

  int numMixes;

  EventExecutorGroup mixExec(CellCenteredEvents::SIZE);

  mixExec.addEventExecutor(CellCenteredEvents::COLOR_MIXING,
                           ColorMixExecute(rng, &mixCNO, &numMixes));

  sim.reserveCellCenteredEventGroups(1, CellCenteredEvents::SIZE);
  sim.addCellCenteredEventGroup(1, mixCNO,
                                ColorMixPropensity(10*F),
                                mixExec);
```

Private member numMixes_ will be used to keep track of how many times a color mixing event is executed. One can see from the above code and from the code of the constructor of ColorMixExecute below,

```
ColorMixExecute::ColorMixExecute(RandNumGenSharedPtr rng,
                                 const CellNeighOffsets * mixCNO,
                                 int * numMixes)
  : rng_(rng),
    mixCNO_(mixCNO),
    numMixes_(numMixes) {

  *numMixes_ = 0;

}
```

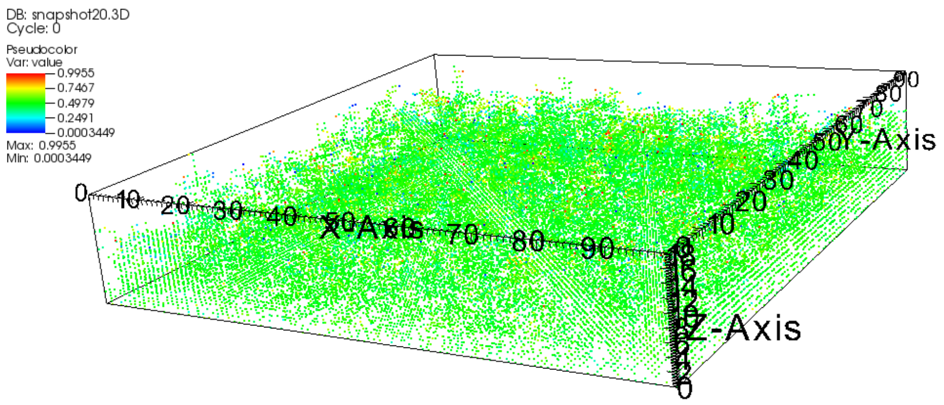
that this private member points to the integer variable numMixes in the driver code. After the simulation has finished running, the value of numMixes will be printed.

The implementation of the operator in the ColorMixExecute class that executes color mixing is shown below:

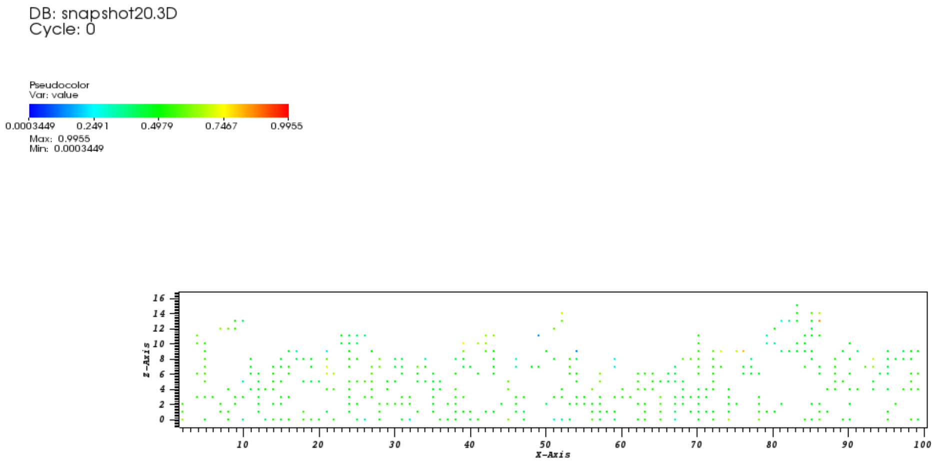
```
void ColorMixExecute::operator()(const KMCThinFilm::CellInds & ci,
                                 const KMCThinFilm::SimulationState & simState,
                                 KMCThinFilm::Lattice & lattice) {

  double color = lattice.getFloat(ci, BDFloatVal::COLOR);
  int numColors = 1;

  // Visiting four lateral neighbors
  for (int whichOffset = 1; whichOffset <= 4; ++whichOffset) {

    CellInds ciNeigh = ci + mixCNO_->getOffset(whichOffset);

    if (lattice.getInt(ciNeigh, BDIntVal::IS_OCCUPIED)) {
      color += lattice.getFloat(ciNeigh, BDFloatVal::COLOR);
      ++numColors;
    }

  }

  CellInds ciDown = ci +  mixCNO_->getOffset(MIX_OFFSET::DOWN);

  bool belowLatticeBottom = (ciDown.k < 0);
  bool ciDownIsOccupied = (belowLatticeBottom || lattice.getInt(ciDown, BDIntVal::IS_OCCUPIED));

  if (ciDownIsOccupied) {
    color += (belowLatticeBottom ?
              (rng_->getNumInOpenIntervalFrom0To1()) :
              lattice.getFloat(ciDown, BDFloatVal::COLOR));
    ++numColors;
  }

  CellInds ciUp = ci +  mixCNO_->getOffset(MIX_OFFSET::UP);

  if ((ciUp.k < lattice.currHeight()) && lattice.getInt(ciUp, BDIntVal::IS_OCCUPIED)) {

    color += lattice.getFloat(ciUp, BDFloatVal::COLOR);
    ++numColors;

  }

  lattice.setFloat(ci, BDFloatVal::COLOR, color/numColors);

  ++(*numMixes_);

}
```

Whereas in the function object to calculate propensity, there were special member functions to check whether index *k* of some set of cell indices was less than zero or greater than or equal to the lattice height, here such checks are performed "manually," so to speak. Also, again cells below the computational lattice are here treated differently than those above it. Cells below the lattice are again assumed to belong to the substrate, and for this contrived example, each particle in the substrate is assumed to have a random color. Cells above the computational lattice are again assumed to belong to the empty space above the deposited thin film, and thus do not contribute to the new value of the color of the cell.

The periodic action used to dump the state of the lattice to a file uses a file format called " Point3D,", which can viewed with the software VisIt < http://visit.llnl.gov>. It is used to display arrangements of particles colored according to some quantity, which in this case is the so-callled color of each lattice cell. The last snapshot, when viewed in VisIt, should look something like this:



A slice of this snapshot in VisIt, showing particles with cell index $j = 50$, should look something like this:



4.3.2 Second implementation

One may have noticed a problem with the implementation of ballistic deposition described above. While only the values of the quantity labeled `BDIntVal::ACTIVE_ZONE_HEIGHT` in the first lattice plane are used, this quantity is defined for all cells of the lattice, which is somewhat wasteful. In this alternate implementation, then, the values of the auxiliary array $h_a(i, j)$ will be stored not within the first plane of the lattice, but rather in a separate auxiliary array. Also, semi-manual tracking will be used for this example. The code for this new implementation is in `doc/example-code/testBallisticDep2` of the installation directory of the ARL KMCThinFilm library.

We begin by defining `IntArray2D` in `EventsAndActions.hpp`, a type for a two-dimensional integer array, using the multi-array implementation from Boost < `http://www.boost.org`>, and a shared pointer for that type, also using a “smart” pointer implementation from the Boost library:

```
#include <boost/multi_array.hpp>
#include <boost/shared_ptr.hpp>

typedef boost::multi_array<int, 2> IntArray2D;
typedef boost::shared_ptr<IntArray2D> IntArray2DSharedPtr;
```

In the definition of the function object class performing ballistic deposition,

```
class DepositionExecute {
public:
  DepositionExecute(const KMCThinFilm::LatticePlanarBBox & planarBBox);

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec);
private:
  IntArray2DSharedPtr activeZoneHeights_;

  std::vector<KMCThinFilm::CellIndsOffset> neighOffsets_;
};
```

we have a shared pointer to an `IntArray2D` object, `activeZoneHeights_`. The constructor takes a KMCThinFilm::LatticePlanarBBox parameter object, which is used to size the array to which `activeZoneHeights_` points:

```
DepositionExecute::DepositionExecute(const LatticePlanarBBox & planarBBox)
: activeZoneHeights_(new IntArray2D) {

  // The resize() member function will initialize the elements of this
  // array to zero.
  activeZoneHeights_->resize(boost::extents[planarBBox.imaxP1 - planarBBox.imin]
                             [planarBBox.jmaxP1 - planarBBox.jmin]);

  neighOffsets_.reserve(4);
  neighOffsets_.push_back(CellIndsOffset( 0,-1));
  neighOffsets_.push_back(CellIndsOffset( 0,+1));
  neighOffsets_.push_back(CellIndsOffset(-1, 0));
  neighOffsets_.push_back(CellIndsOffset(+1, 0));

}
```

In the driver code `testBallisticDep.cpp`, the KMCThinFilm::LatticePlanarBBox parameter object passed to this constructor contains the lateral bounds of the global lattice.

Finally, the implementation of ballistic deposition is as follows:

```
void DepositionExecute::operator()(const KMCThinFilm::CellInds & ci,
                                   const KMCThinFilm::SimulationState & simState,
                                   const KMCThinFilm::Lattice & lattice,
                                   std::vector<KMCThinFilm::CellsToChange> & ctcVec) {

  /* Ballistic deposition algorithm for cubic lattices from Meakin and
     Krug, Physical Review A, vol. 46, num. 6, pp. 3390-3399 (1992).*/

  int kDepAtom = (*activeZoneHeights_)[ci.i][ci.j];

  CellInds ciTo(ci.i, ci.j, kDepAtom);

  CellsToChange & ctc = ctcVec[0];

  ctc.setCenter(ciTo);

  ctc.addLatticePlanes(ciTo.k - ci.k);
  ctc.setInt(0, BDIntVal::IS_OCCUPIED, 1);

  for (std::vector<CellIndsOffset>::const_iterator offsetItr = neighOffsets_.begin(),
         offsetItrEnd = neighOffsets_.end(); offsetItr != offsetItrEnd; ++offsetItr) {

    CellInds neigh = ciTo + *offsetItr;

    int i = lattice.wrapI(neigh);
    int j = lattice.wrapJ(neigh);

    if ((*activeZoneHeights_)[i][j] < kDepAtom) {
      (*activeZoneHeights_)[i][j] = kDepAtom;
    }
  }

  ++((*activeZoneHeights_)[ci.i][ci.j]);
}
```

Note that since `activeZoneHeights_` is an external array, we use KMCThinFilm::Lattice::wrapI() and KMCThinFilm::Lattice::wrapJ() to ensure that the array indices used with `activeZoneHeights_` are correctly wrapped to account for periodic boundary conditions. This was not necessary when $h_a(i,j)$ was stored in the first plane of the lattice and KMCThinFilm::Lattice::getInt() was used to access its contents, since getInt() automatically accounts for any periodic boundary conditions. Also, since semi-manual tracking is used, `lattice` is a *constant* reference, so planes must be added to the lattice via the member function KMCThinFilm::CellsToChange::addLatticePlanes().

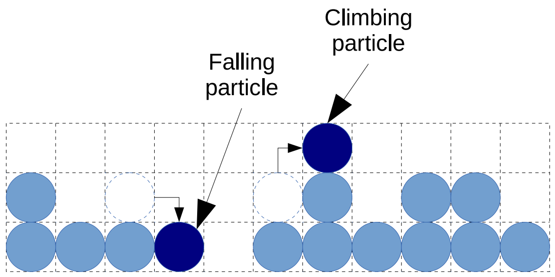
One may ask why `activeZoneHeights_` is a pointer to an array instead of just an array. The reason for this is that when the DepositionExecute object is passed to KMCThinFilm::Simulation::addOverLatticeEvent(), it is passed by value. This would mean that there would be two copies of the activeZoneHeights_ array, one created when the DepositionExecute object is constructed, and another when a copy of this object is made to pass it by value. By using a pointer, only a pointer to the array is copied. A "smart" shared pointer is used so that it will automatically be deleted when there are no more references to it.

One downside of this implementation is that it is more difficult to parallelize than the previous one. With *active↩ ZoneHeights_ as a separate array, the process of splitting up the array, taking care of updates of ghost entries, etc. has to be handled explicitly. The handling of MPI communication could be done via a periodic action executed at each step, though one would need to refactor the code so that a DepositionExecute object and the function object performing the periodic action could both access the array containing $h_a(i,j)$.

The resulting simulated thin film produced by this implementation is similar to that produced in the previous simulation but not the same, even if the same seed to the random number generator has been used. This is because the previous implementation contains calls to lattice.setInt(..., BDIntVal::ACTIVE_ZONE_HEIGHT, ...) in the function object that executes a deposition event. This triggers the solvers in the simulation to examine the possible events in the neighborhood of where the value of the quantity labeled BDIntVal::ACTIVE_ZONE_HEIGHT has been changed, which, due to implementation details in the ARL KMCThinFilm library, ends up changing the order that possible events with the same propensity are stored, which in turn affects which event is randomly chosen at a time step.

4.4 Example: Implementations of a patterned substrate model

This example is meant to show how the ARL KMCThinFilm library may be used to implement models that are unusual in the sense that they have features that would almost inevitably require custom code to implement. An example of such a model is that of a patterned substrate [5]. Like the fractal model, they use solid-on-solid modeling of a cubic lattice, which again means that the arrangement of atoms in the actual true lattice can be stored in a two-dimensional computational lattice, such that $(i,j,0)$ in the computational lattice stores the height of the column of particles at $(i,j,0)$ in the true lattice. As before, if a diffusing particle in the true lattice moves to a nearest-neighboring site that is not just above another particle, then the diffusing particle will fall until it lands on top of another particle. Also, if the site to which a particle attempts to move is occupied, then the particle will climb to the top of the column of particles that contains that occupied site. Falling and climbing are illustrated below, where an empty circle indicates the old position of a particle and a dark filled circle indicates the new position of a particle:



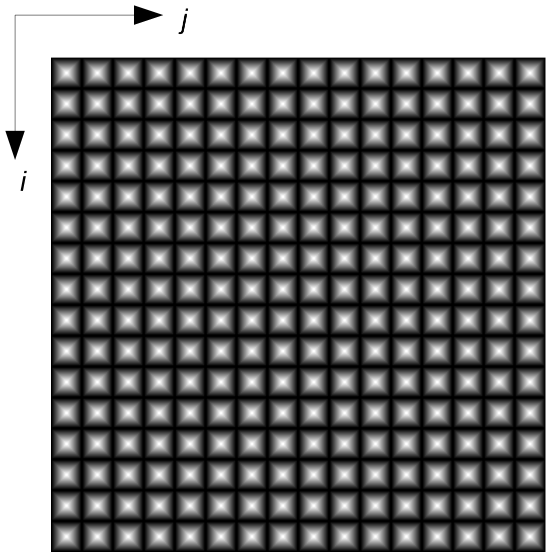
In one of the models of Kuronen et al., the propensity for the hopping of a particle at a lattice cell is

$$p = k\exp\left(-\frac{E}{k_B T}\right)$$

where k_B is the Boltzmann constant, T is temperature, $k = kT/h$ with h being Planck's constant, and

$$E = E_s(i,j) + nE_n$$

Here, n is the number of occupied lateral neighbors of the cell, E_n $(= 0.18\text{ eV})$ is a measure of the bond strength between a particle and a lateral nearest neighboring particle, and E_s is a position-dependent diffusion barrier dependent on the patterning of the substrate, which varies with i and j as illustrated below, with brighter colors indicating higher values.



The above pattern consists of a 16 × 16 array of square domains, and each domain is an array of 22 × 22 elements. This implies that the lattice has lateral dimensions (16·22) × (16·22). At the edge of a domain, E_s is at its minimum value, 0.65 eV, and at the center of a domain, it is at its maximum, 0.85 eV. E_s varies linearly between its minimum and maximum values.

The results of this patterned substrate model should be an arrangement of islands centered about the parts of the substrate where E_s is maximum.

4.4.1 First implementation

The code for this implementation is in `doc/example-code/testPatternedSurface1` of the installation directory of the ARL KMCThinFilm library. In it, the value $E_s(i, j)$ is stored at cell $(i, j, 0)$ of the computational lattice. This requires using a special method to initialize the lattice. The values of $E_s(i, j)$ are stored in a file (entitled `tiled16x16↩ Domain.dat`) where the first line contains the lateral dimensions of the lattice and subsequent lines are of the form

```
i j E_s(i,j)
```

where the first two numbers in the line are the lateral lattice cell indices and the third number is the value of E_s for those indices.

We begin with the files `InitLattice.hpp` and `InitLattice.cpp`, which contain the definition and implementation of the function object than initializes the lattice. The contents of the first file are

```
#ifndef INIT_LATTICE_HPP
#define INIT_LATTICE_HPP

#include <string>

#include <KMCThinFilm/Lattice.hpp>
#include <KMCThinFilm/ErrorHandling.hpp>

class InitLatticeFromFile {
public:
  InitLatticeFromFile(const std::string & inpFName)
    : inpFName_(inpFName)
  {}

  void operator()(KMCThinFilm::Lattice & lattice) const;
private:
  std::string inpFName_;
};

#endif /* INIT_LATTICE_HPP */
```

The constructor of the `InitLatticeFromFile` class here takes as an argument the name of the input file containing the values of E_s. The contents of `InitLattice.cpp` are as follows:

```
#include "InitLattice.hpp"
#include "EventsAndActions.hpp"

#include <fstream>

#include <boost/array.hpp>
#include <boost/lexical_cast.hpp>

using namespace KMCThinFilm;

void InitLatticeFromFile::operator()(Lattice & lattice) const {

  std::ifstream inpFile(inpFName_.c_str());

  boost::array<int,2> globalPlanarDims;

  inpFile >> globalPlanarDims[0] >> globalPlanarDims[1];

  LatticePlanarBBox globalPlanarBBox;
  lattice.getGlobalPlanarBBox(globalPlanarBBox);

  exitOnCondition((globalPlanarDims[0] != (globalPlanarBBox.imaxP1 - globalPlanarBBox.imin)) ||
                  (globalPlanarDims[1] != (globalPlanarBBox.jmaxP1 - globalPlanarBBox.jmin)),
                  "Mismatch in lattice dimensions and dimensions of strain eng. den. array.");

  lattice.addPlanes(1);

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int maxLineNumP1 = (globalPlanarDims[0])*(globalPlanarDims[1]);

  CellInds ci; ci.k = 0;
  for (int lineNum = 0; lineNum < maxLineNumP1; ++lineNum) {

    int i, j;
    double E_s;

    inpFile >> i >> j >> E_s;

    if ((i >= localPlanarBBox.imin) && (i < localPlanarBBox.imaxP1) &&
        (j >= localPlanarBBox.jmin) && (j < localPlanarBBox.jmaxP1)) {

      ci.i = i;
      ci.j = j;

      lattice.setFloat(ci, PSFloatVal::E_s, E_s);
    }

  }

  inpFile.close();

}
```

This operator reads in the file containing the values of $E_s(i,j)$, line by line, and stores these values in the floating-point array element labeled `PSFloatVal::E_s` in lattice cell $(i,j,0)$. The (global) lateral bounds of the lattice have already been initialized before this operator has even been invoked, so here there is a check to ensure that the lateral bounds of the lattice indicated in the first line of the file match the actual lateral bounds of the lattice. After this check, the actual initialization begins. First, a lattice plane is added with the KMCThinFilm::Lattice::addPlanes() member function. Shortly afterward comes the reading in of the rest of the file containing the values of $E_s(i,j)$. There is some allowance for parallel implementation. The local bounds of indices i and j are determined using KMCThinFilm::Lattice::getLocalPlanarBBox. While each MPI process would still read in the whole file, which is not that efficient, the process would also only store the values of $E_s(i,j)$ for the lattice cell indices i and j whose values are within the previously determined local bounds.

In the driver file `testPatternedSurface.cpp`, the simulation is initialized as follows:

```
  LatticeParams latParams;
  latParams.numIntsPerCell = PSIntVal::SIZE;
  latParams.numFloatsPerCell = PSFloatVal::SIZE;

  std::ifstream patternFile(patternFName.c_str());

  if (patternFile) {
    patternFile >> latParams.globalPlanarDims[0] >> latParams.globalPlanarDims[1];
  }
  else {
    exitWithMsg("Cannot access " + patternFName +
                ". Check if your are executing this program from the correct directory.");
  }

  patternFile.close();

  latParams.latInit = InitLatticeFromFile(patternFName);

  Simulation sim(latParams);
```

Here, `patternFName` is the name of the file containing the values of $E_s(i,j)$. The first line of this file is read in to determine the lateral dimensions of the lattice, and then the file is closed. Afterwards, the *latInit* member of the KMCThinFilm::LatticeParams parameter object is set to an instance of the `InitLatticeFromFile` class, which is of course initialized to read from the file with the name `patternFName`. (This also means that this implementation is a bit inelegant, since part of the file `patternFName` will be read twice.)

This takes care of the initialization of the lattice.

Deposition here is done by the same means as in the example of the fractal solid-on-solid model using auto-tracking, while the execution of a hopping event is done by the same means as in the example of the fractal solid-on-solid model using semi-manual tracking. It is mainly the determination of a hopping event's propensity that is different. The definition of the function object class for determining propensity of hopping is shown below:

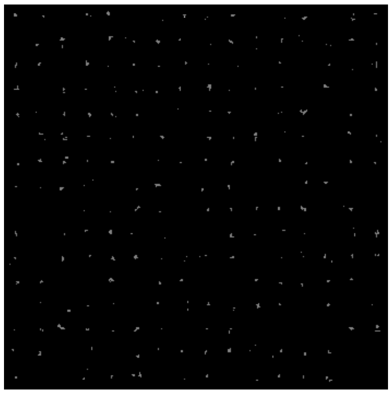
```
class HoppingPropensity {
public:
  HoppingPropensity(double E_n, double T)
    : E_n_(E_n), kBT_(PhysConst::kB*T), k_(kBT_/PhysConst::h)
  {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double E_n_, kBT_, k_;
};
```

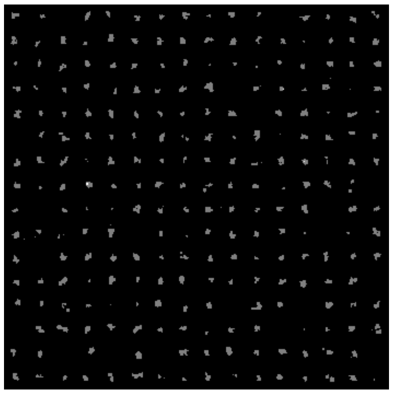
The constants `PhysConst::kB` and `PhysConst::h` are merely the Boltzmann constant and Planck's constant, and they are defined in the file `PhysicalConstants.hpp` of this example code. The private variables `E_n_`, `kBT_`, and `k_` correspond to the aforementioned expressions E_n, k_BT, and k. The implementation of the function object class for determining hopping propensity is shown below:

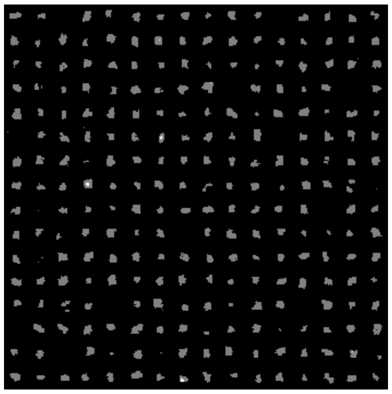
```
void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  KMCThinFilm::CellToProbe ctpSelf = cnp.getCellToProbe(HopOffset::SELF);

  int currHeight = cnp.getInt(ctpSelf, PSIntVal::HEIGHT);

  if (currHeight > 0) {

    int n = 0;

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::UP), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    double E = cnp.getFloat(ctpSelf, PSFloatVal::E_s) + n*E_n_;

    double p = k_*std::exp(-E/kBT_);

    for (int i = 0; i < PSCellCenteredEvents::SIZE; ++i) {
      propensityVec[i] = p;
    }

  }
}
```

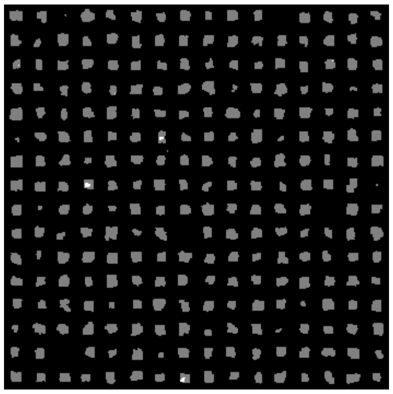
Here, `currHeight` is the height of the column of particles at $(\texttt{ci.i}, \texttt{ci.j}, 0)$ in the true lattice. Let $(i_{\text{off}}, j_{\text{off}})$ be an offset of cell indices, which for `HopOffset::UP` is $(0, +1)$, for `HopOffset::DOWN` is $(0, -1)$, for `HopOffset::`↩
`LEFT` is $(-1, 0)$, and for `HopOffset::RIGHT` is $(+1, 0)$. If a column of particles at $(\texttt{ci.i} + i_{\text{off}}, \texttt{ci.j} + j_{\text{off}}, 0)$ is greater than or equal to `currHeight`, that means that the site $(\texttt{ci.i}, \texttt{ci.j}, \texttt{currHeight} - 1)$ in the true lattice has an occupied nearest neighbor at $(\texttt{ci.i} + i_{\text{off}}, \texttt{ci.j} + j_{\text{off}}, \texttt{currHeight} - 1)$, and the variable `n` is incremented accordingly. The calculation of the propensity, then, follows straightforwardly from the aforementioned formulas for the patterned substrate [5].

An overhead view of the surface of the islands on the patterned substrate at various simulation times t, from 2.27 units to its maximum value of 50.0 units, is shown below.


t = 2.27


t = 11.36


t = 22.7


t = 50.0

4.4.2 Second implementation

The previous implementation of the patterned substrate model [5] does not take advantage of the periodicity of E_s. This new implementation will. Instead of storing values of $E_s(i, j)$ in the computational lattice itself, they will be stored in a small array used by the function object used to calculate propensities. This means that no special means to initialize the lattice is required. The code for this implementation is in `doc/example-code/testPatternedSurface2` of the installation directory of the ARL KMCThinFilm library.

The file containing the values of $E_s(i, j)$, `singleDomain.dat` only contains the values of E_s within a single domain of (22 × 22 array) elements. The first line of this file contains the dimensions of this domain, and the rest of the lines in this file have the following format,

```
i j E_s(i,j)
```

where the first two numbers in the line are the indices of the domain array and the third number is the value of E_s for those indices. In the driver code `testPatternedSurface.cpp`, this file is read into an array as follows,

```
  DblArray2D E_s;

  std::ifstream patternFile("../singleDomain.dat");
  boost::array<int,2> E_s_extents;

  patternFile >> E_s_extents[0] >> E_s_extents[1];
  E_s.resize(E_s_extents);

  int E_s_i, E_s_j;
  double E_s_val;

  while (patternFile >> E_s_i >> E_s_j >> E_s_val) {
    E_s[E_s_i][E_s_j] = E_s_val;
  }
```

where `DblArray2D` is defined in `EventsAndActions.hpp` as

```
#include <boost/multi_array.hpp>
typedef boost::multi_array<double,2> DblArray2D;
```

Again, we use the multi-array implementation from Boost < `http://www.boost.org`>. The function object for determining hopping propensity is defined as follows:

```
class HoppingPropensity {
public:
  HoppingPropensity(const DblArray2D * E_s, double E_n, double T)
    : E_s_(E_s), E_n_(E_n), kBT_(PhysConst::kB*T), k_(kBT_/PhysConst::h)
  {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  const DblArray2D * E_s_;
  double E_n_, kBT_, k_;
};
```

This function object is similar to the corresponding one in the previous implementation, except that now there is a new private variable, `E_s_`, which points to an array containing the values of $E_s(i, j)$ in a domain (and, of course, a constructor with an argument used to set the value of this private variable). The reason for `E_s_` being a pointer is similar to the one for having `activeZoneHeights_` be a pointer in the second implementation of the ballistic deposition model. When an instance of the `HoppingPropensity` object is passed to KMCThinFilm::Simulation::addCellCenteredEventGroup(), it is passed by value, and having `E_s_` be a pointer to an array rather than the array itself means that only a pointer to the array is copied rather than the whole array.

The function object for determining hopping propensity is implemented as follows:

```
void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  KMCThinFilm::CellToProbe ctpSelf = cnp.getCellToProbe(HopOffset::SELF);

  int currHeight = cnp.getInt(ctpSelf, PSIntVal::HEIGHT);

  if (currHeight > 0) {

    int n = 0;

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::UP), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    const CellInds & ciSelf = ctpSelf.inds();

    int coord_i_in_domain = ciSelf.i % (E_s_->shape()[0]);
    int coord_j_in_domain = ciSelf.j % (E_s_->shape()[1]);

    double E = (*E_s_)[coord_i_in_domain][coord_j_in_domain] + n*E_n_;

    double p = k_*std::exp(-E/kBT_);

    for (int i = 0; i < PSCellCenteredEvents::SIZE; ++i) {
      propensityVec[i] = p;
    }

  }
}
```

This function object is also similar to the corresponding one in the previous implementation, except that

1. it actually accesses lattice cell indices, via the member function KMCThinFilm::CellToProbe::inds(), and
2. it converts the in-plane cell indices, `ciSelf.i` and `ciSelf.j`, into indices `coord_i_in_domain` and `coord_j_in_domain` of the array to which `E_s_` points.

The latter is more a matter for the particular kinetic Monte Carlo model being implemented, but the former is more generally useful functionality.

The rest of this implementation is the same as the previous one. Provided that the same parameters, lattice dimensions, domain dimensions, and seeding for the random number generator are used in both this implementation and the previous one, the two implementations should yield exactly the same results.

Chapter 5

Namespace Documentation

5.1 KMCThinFilm Namespace Reference

Namespaces

- namespace SolverId
- namespace TimeIncr

Classes

- class AddEmptyPlanes
- class CellInds
- class CellIndsOffset
- class CellNeighOffsets
- class CellNeighProbe
- class CellsToChange
- class CellToProbe
- class EventExecutorGroup
- class Lattice
- struct LatticeParams
- struct LatticePlanarBBox
- class RandNumGen
- class RandNumGenDCMT
- class RandNumGenMT19937
- class RandNumGenRngStreams
- class Simulation
- class SimulationState

Typedefs

- typedef boost::function< void(const CellNeighProbe &, std::vector< double > &)> CellCenteredGroupPropensities
- typedef boost::function< void(const CellInds &, const SimulationState &, Lattice &)> EventExecutorAutoTrack
- typedef boost::function< void(const CellInds &, const SimulationState &, const Lattice &, std::vector< CellsToChange > &)> EventExecutorSemiManualTrack

- typedef boost::function< void(Lattice &lattice)> LatticeInitializer
- typedef boost::function< void(const SimulationState &, Lattice &)> PeriodicAction
- typedef boost::shared_ptr< RandNumGen > RandNumGenSharedPtr
- typedef boost::function< void(const CellInds &, const Lattice &, std::vector< int > &emptyIntVals, std↩ ::vector< double > &emptyFloatVals)> SetEmptyCellVals

Functions

- void abortOnCondition (bool condition, const std::string &msg)
- void abortWithMsg (const std::string &msg)
- void exitOnCondition (bool condition, const std::string &msg)
- void exitWithMsg (const std::string &msg)
- CellInds operator+ (const CellInds &ci, const CellIndsOffset &offset)
- CellIndsOffset operator+ (const CellIndsOffset &offset1, const CellIndsOffset &offset2)

5.1.1 Detailed Description

This is the namespace that contains all the class definitions and functions of the ARL KMCThinFilm library.

5.1.2 Typedef Documentation

CellCenteredGroupPropensities

```
typedef boost::function<void (const CellNeighProbe &, std::vector<double> &)> KMCThinFilm::CellCenteredGroupPropensities
```

Signature of the function object used to determine the propensities of a group of related events.

EventExecutorAutoTrack

```
typedef boost::function<void (const CellInds &, const SimulationState &, Lattice &)> KMCThinFilm::EventExecutorAutoTrack
```

Signature for a function object to execute an event where "auto-tracking" is used.

In auto-tracking, cells affected by the event are determined automatically from the offsets used to calculate propensities and the positions of cells directly changed by the executed event, which are logged automatically when Lattice::setInt() and Lattice::setFloat() are called. This mode of tracking may lead to redundant propensity calculations.

EventExecutorSemiManualTrack

```
typedef boost::function<void (const CellInds &, const SimulationState &, const Lattice &, std::vector<CellsToChange>
&)> KMCThinFilm::EventExecutorSemiManualTrack
```

Signature for a function object to execute an event where "semi-manual" tracking is used.

In semi-manual tracking, the user specifies the relative positions of the cells changed by an event. Using this information along with the offsets used to calculate propensities can reduce or eliminate redundant propensity calculations.

LatticeInitializer

```
typedef boost::function<void (Lattice & lattice)> KMCThinFilm::LatticeInitializer
```

Signature of a function or function object to be used to initialize a lattice.

PeriodicAction

```
typedef boost::function<void (const SimulationState &, Lattice &)> KMCThinFilm::PeriodicAction
```

Signature for the function object called periodically during a simulation.

RandNumGenSharedPtr

```
typedef boost::shared_ptr<RandNumGen> KMCThinFilm::RandNumGenSharedPtr
```

"Smart" pointer used to point to an implementation of the RandNumGen class.

Examples

testBallisticDep1/EventsAndActions.hpp, testBallisticDep1/InitLattice.hpp, and testBallisticDep2/EventsAndActions.hpp.

SetEmptyCellVals

```
typedef boost::function<void (const CellInds &, const Lattice &, std::vector<int> & emptyIntVals, std::vector<double>
& emptyFloatVals)> KMCThinFilm::SetEmptyCellVals
```

Signature of a function or function object to be used to set the integers and floating-point numbers in an empty lattice cell to their proper values.

Note that when this function is used, the vectors *emptyIntVals* and *emptyFloatVals* have already had their sizes set to the number of integer values and number of floating-point values per lattice cell, respectively.

5.1.3 Function Documentation

abortOnCondition()

```
void KMCThinFilm::abortOnCondition (
              bool condition,
              const std::string & msg)
```

Aborts the program and prints the message string *msg* on all processors where *condition* is true.

This function terminates less gracefully than exitOnCondition(), but may be used for cases where *condition* may not necessarily evaluate to the same value on all processors.

See also

exitOnCondition()

abortWithMsg()

```
void KMCThinFilm::abortWithMsg (
              const std::string & msg)
```

Aborts the program and prints the message string *msg* on all processors.

exitOnCondition()

```
void KMCThinFilm::exitOnCondition (
              bool condition,
              const std::string & msg)
```

Exits the program and prints the message string *msg* on rank 0 of MPI_COMM_WORLD if *condition* is true.

This function allows for limited error handling, but only for the cases where *condition* is tested on all processors and *will evaluate to the same value on all processors*. If these conditions will not always be satisfied, then another method of error handling, e.g. abortOnCondition(), should be used instead.

See also

abortOnCondition()

exitWithMsg()

```
void KMCThinFilm::exitWithMsg (
                const std::string & msg)
```

Exits the program and prints the message string *msg* on rank 0 of MPI_COMM_WORLD.

operator+() [1/2]

```
CellInds KMCThinFilm::operator+ (
                const CellInds & ci,
                const CellIndsOffset & offset)
```

Returns a CellInds instance with the indices ci.i + offset.i, ci.j + offset.j, and ci.k + offset.k.

operator+() [2/2]

```
CellIndsOffset KMCThinFilm::operator+ (
                const CellIndsOffset & offset1,
                const CellIndsOffset & offset2)
```

Returns a CellIndsOffset instance with the indices offset1.i + offset2.i, offset1.j + offset2.j, and offset1.k + offset2.k.

5.2 KMCThinFilm::SolverId Namespace Reference

Enumerations

- enum Type { DYNAMIC_SCHULZE , BINARY_TREE }

5.2.1 Detailed Description

Namespace to enclose the SolverId::Type enumeration

5.2.2 Enumeration Type Documentation

Type

```
enum KMCThinFilm::SolverId::Type
```

Types of solvers used in a KMC simulation

Enumerator

DYNAMIC_SCHULZE	A solver implementing an algorithm similar to that described by Schulze in *Physical Review E*, vol. 65, 036704 (2002), but allowing for the number of event propensities to be determined at runtime. This should allow the choosing of an event to scale with the number of unique propensities, rather than the number of events.
BINARY_TREE	A solver where the possible events are stored as leaves in a binary tree, allowing the choosing of an event to scale as $O(log_2 N)$, where N is the number of possible events. May be faster than DYNAMIC_SCHULZE in practice.

5.3 KMCThinFilm::TimeIncr Namespace Reference

Namespaces

- namespace SchemeName
- namespace SchemeParam

Classes

- class SchemeVars

5.3.1 Detailed Description

This namespace wraps a class and enumerations relating to specifying parallel time-stepping schemes.

5.4 KMCThinFilm::TimeIncr::SchemeName Namespace Reference

Enumerations

- enum Type { MAX_AVG_PROPENSITY_PER_POSS_EVENT , MAX_SINGLE_PROPENSITY , FIXED_VALUE }

5.4.1 Detailed Description

Namespace to enclose the SchemeName::Type enumeration

5.4.2 Enumeration Type Documentation

Type

`enum KMCThinFilm::TimeIncr::SchemeName::Type`

Types of parallel time-incrementing schemes

Enumerator

MAX_AVG_PROPENSITY_PER_POSS_EVENT	Scheme where the time step is inversely proportional to the maximum of the average propensities per possible event from all sectors, where the average propensity per possible event is the sum of all propensities in a sector divided by the number of possible events in that sector. This is similar to the adaptive time step scheme used in SPPARKS (`http://spparks.sandia.gov/`).
MAX_SINGLE_PROPENSITY	Scheme where the time step is inversely proportional to largest event propensity available in the simulation. This is similar to the adaptive time step scheme suggested by Shim and Amar in *Physical Review B*, vol. 71, 125432 (2005).
FIXED_VALUE	Scheme where the time step is a fixed value.

5.5 KMCThinFilm::TimeIncr::SchemeParam Namespace Reference

Enumerations

- enum Type { NSTOP , TSTOP_MAX , TSTOP }

5.5.1 Detailed Description

Namespace to enclose the SchemeParam::Type enumeration

5.5.2 Enumeration Type Documentation

Type

`enum KMCThinFilm::TimeIncr::SchemeParam::Type`

Parameters used in the parallel time-incrementing schemes

Enumerator

NSTOP	This is a premultiplier used in adaptive time schemes where the time step takes the form $t_{stop} = NSTOP/P$, where P is some function of the available event propensities.
TSTOP_MAX	Maximum time step for the adaptive schemes. The time step is set to this value if the step determined by the adaptive scheme would otherwise exceed it.
TSTOP	Time step used for scheme SchemeName::FIXED_VALUE.

Chapter 6

Class Documentation

6.1 KMCThinFilm::AddEmptyPlanes Class Reference

`#include <Lattice.hpp>`

Public Member Functions

- AddEmptyPlanes (int numPlanesToAdd)
- void operator() (Lattice &lattice) const

6.1.1 Detailed Description

Simple function object used to add empty planes to the lattice, especially during initialization of the lattice.

6.1.2 Constructor & Destructor Documentation

AddEmptyPlanes()

```
KMCThinFilm::AddEmptyPlanes::AddEmptyPlanes (
                int numPlanesToAdd)
```

Constructor, defines number of lattice planes to add.

6.1.3 Member Function Documentation

operator()()

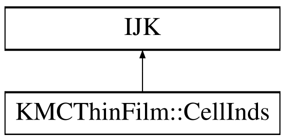
```
void KMCThinFilm::AddEmptyPlanes::operator() (
                Lattice & lattice) const
```

Overload of operator() that performs the actual adding of planes.

6.2 KMCThinFilm::CellInds Class Reference

`#include <CellInds.hpp>`

Inheritance diagram for KMCThinFilm::CellInds:



Public Member Functions

- CellInds ()
- CellInds (int ii, int jj, int kk=0)

6.2.1 Detailed Description

Indices of a lattice cell.

This class has three *public* data members, i, j, and k, which represent the first, second, and third indices. If the lattice has primitive lattice vectors ai, aj, and ak, then the location of the lattice cell may be taken as i∗ai + j∗aj + k∗ak.

See also

CellIndsOffset KMCThinFilm::operator+()

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep1/EventsAndActions.hpp, testBallisticDep1/InitLattice.cpp, testBallisticDep1/InitLattice.hpp, testBallisticDep2/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.hpp, testFractal/EventsAndActions.cpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/Even
testPatternedSurface1/InitLattice.cpp, testPatternedSurface2/EventsAndActions.cpp, and testPatternedSurface2/EventsActions

6.2.2 Constructor & Destructor Documentation

CellInds() `[1/2]`

```
KMCThinFilm::CellInds::CellInds ()  [inline]
```

Constructs an empty CellInds.

CellInds() `[2/2]`

```
KMCThinFilm::CellInds::CellInds (
                int ii,
                int jj,
                int kk = 0)  [inline]
```

Constructs a CellInds instance from a doublet or triplet of integers.

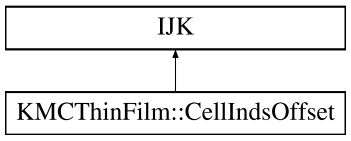
Parameters

ii	First index of a lattice cell
jj	Second index of lattice cell
kk	Third index of lattice cell

6.3 KMCThinFilm::CellIndsOffset Class Reference

`#include <CellInds.hpp>`

Inheritance diagram for KMCThinFilm::CellIndsOffset:



Public Member Functions

- CellIndsOffset ()
- CellIndsOffset (int ii, int jj, int kk=0)
- CellIndsOffset operator- () const

6.3.1 Detailed Description

An offset to indices of some lattice cell.

Like the CellInds class, it has three *public* data members, i, j, and k, which represent the offsets to be added to a given set of lattice cell indices. For example, one may write

```
CellInds ci(1,2,3);
CellInds offset(1,1,-1);

CellInds ci_w_offset = ci + offset;
```

where the indices of ci_w_offset now equal 2, 3, and 2.

See also

CellInds KMCThinFilm::operator+()

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/EventsAndActions.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testFractal.cpp, testFractal_semi_manual_tr
testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatternedSurface.cpp.

6.3.2 Constructor & Destructor Documentation

CellIndsOffset() `[1/2]`

```
KMCThinFilm::CellIndsOffset::CellIndsOffset ()   [inline]
```

Constructs an empty CellIndsOffset.

CellIndsOffset() `[2/2]`

```
KMCThinFilm::CellIndsOffset::CellIndsOffset (
                int ii,
                int jj,
                int kk = 0)   [inline]
```

Constructs a CellIndsOffset instance from a doublet or triplet of integers.

Parameters

ii	Offset to the first index of a lattice cell

Parameters

jj	Offset to the second index of lattice cell
kk	Offset to the third index of lattice cell

6.3.3 Member Function Documentation

operator-()

```
CellIndsOffset KMCThinFilm::CellIndsOffset::operator- () const
```

Unary minus operator. Returns a sign-reversed copy of the offset.

6.4 KMCThinFilm::CellNeighOffsets Class Reference

```
#include <CellNeighOffsets.hpp>
```

Public Member Functions

- CellNeighOffsets (int numberOfOffsets)
- void addOffset (int whichOffset, const CellIndsOffset &offset)
- void clearOffsets ()
- const CellIndsOffset & getOffset (int whichOffset) const
- int numOffsets () const
- void resetOffsets (int numberOfNewOffsets)

6.4.1 Detailed Description

Collection of CellIndsOffset objects that are used to determine an event's propensity.

Each offset in the collection is associated with an integer ID. No matter what, this collection always contains a CellIndsOffset object with the indices (0,0,0), and this offset has the integer ID of zero.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/EventsAndActions.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testFractal.cpp, testFractal_semi_manual_tr
testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatternedSurface.cpp.

6.4.2 Constructor & Destructor Documentation

CellNeighOffsets()

```
KMCThinFilm::CellNeighOffsets::CellNeighOffsets (
                int numberOfOffsets)  [explicit]
```

Constructs a CellNeighOffsets object.

Parameters

numberOfOffsets	Number of offsets to be added.

6.4.3 Member Function Documentation

addOffset()

```
void KMCThinFilm::CellNeighOffsets::addOffset (
              int whichOffset,
              const CellIndsOffset & offset)
```

Adds an additional offset.

The argument *whichOffset* is not allowed to be zero, since that integer ID is already associated with the offset (0,0,0).

Parameters

whichOffset	Integer ID of offset. Must be >= 1 and less than numOffsets()
offset	The offset to be added.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

clearOffsets()

```
void KMCThinFilm::CellNeighOffsets::clearOffsets ()
```

Removes all previously added offsets.

Note that no new offsets may be added until resetOffsets() is called.

getOffset()

```
const CellIndsOffset & KMCThinFilm::CellNeighOffsets::getOffset (
              int whichOffset) const
```

Returns the offset with the integer ID *whichOffset*. Note that, here, *whichOffset is* allowed to be zero.

Parameters

whichOffset	Integer ID of offset.

Examples

testFractal_parallel/testFractal.cpp, testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cp
and testPatternedSurface2/testPatternedSurface.cpp.

numOffsets()

```
int KMCThinFilm::CellNeighOffsets::numOffsets () const
```

Number of offsets stored.

resetOffsets()

```
void KMCThinFilm::CellNeighOffsets::resetOffsets (
              int numberOfNewOffsets)
```

Removes all previous offsets and allocates memory for *numberOfNewOffsets* CellIndsOffset objects.

6.5 KMCThinFilm::CellNeighProbe Class Reference

`#include <CellNeighProbe.hpp>`

Public Member Functions

- CellNeighProbe (const Lattice ∗lattice=NULL)
- void attachCellInds (const CellInds ∗ci, const std::vector< CellIndsOffset > ∗cioVecPtr)
- void attachLattice (const Lattice ∗lattice)
- bool belowLatticeBottom (const CellToProbe &ctp) const
- bool exceedsLatticeHeight (const CellToProbe &ctp) const
- CellToProbe getCellToProbe (int probedCellInd) const
- double getFloat (const CellToProbe &ctp, int whichFloat) const
- int getInt (const CellToProbe &ctp, int whichInt) const

6.5.1 Detailed Description

A class used by a CellCenteredPropensity function object to probe the states of neighboring lattice cells in order to determine a cell-centered event's propensity.

Each neighboring lattice cell is identified by one of the integer IDs used to label the offsets inside a CellNeighOffsets object, and the indices of this neighboring lattice cell are that of the current cell (about which an event is centered) plus the offset corresponding to its integer ID.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.cpp, testFractal_semi_manual_track/Events
testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.hpp, testPatternedSurface2/EventsAndAc
and testPatternedSurface2/EventsAndActions.hpp.

6.5.2 Constructor & Destructor Documentation

CellNeighProbe()

```
KMCThinFilm::CellNeighProbe::CellNeighProbe (
                const Lattice ∗ lattice = NULL)   [explicit]
```

[**ADVANCED**] Constructs a CellNeighProbe and attaches it to a Lattice object. Mainly for use in testing and debugging.

6.5.3 Member Function Documentation

attachCellInds()

```
void KMCThinFilm::CellNeighProbe::attachCellInds (
                const CellInds ∗ ci,
                const std::vector< CellIndsOffset > ∗ cioVecPtr)
```

[**ADVANCED**] Attaches a vector of CellIndsOffset objects (generated from a CellNeighOffsets instance). Mainly for use in testing and debugging.

Parameters

ci	Pointer to the indices of the cell about which an event is centered.

cioVecPtr	Pointer to the vector of CellIndsOffset objects that define the neighboring lattice cells used to calculate the propensity of the event.

attachLattice()

```
void KMCThinFilm::CellNeighProbe::attachLattice (
              const Lattice * lattice)
```

[**ADVANCED**] Attaches a Lattice object, replacing whatever lattice object that was attached before. Mainly for use in testing and debugging.

belowLatticeBottom()

```
bool KMCThinFilm::CellNeighProbe::belowLatticeBottom (
              const CellToProbe & ctp) const
```

Indicates whether the height coordinate of the (possibly non-existent) lattice cell pointed to by *ctp* is less than zero. If true, it implies that this lattice cell does not and will not actually exist in the simulation.

Examples

testBallisticDep1/EventsAndActions.cpp, and testBallisticDep2/EventsAndActions.cpp.

exceedsLatticeHeight()

```
bool KMCThinFilm::CellNeighProbe::exceedsLatticeHeight (
              const CellToProbe & ctp) const
```

Indicates whether the height coordinate of the (possibly non-existent) lattice cell pointed to by *ctp* exceeds the current height of the lattice. If true, it implies that this lattice cell does not actually exist yet in the simulation.

Examples

testBallisticDep1/EventsAndActions.cpp, and testBallisticDep2/EventsAndActions.cpp.

getCellToProbe()

```
CellToProbe KMCThinFilm::CellNeighProbe::getCellToProbe (
              int probedCellInd) const
```

Retrieves the lattice cell to be probed from its integer ID.

Parameters

probedCellInd	Integer ID of the lattice cell (and its corresponding offset).

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, and testPatternedSurface2/I

getFloat()

```
double KMCThinFilm::CellNeighProbe::getFloat (
              const CellToProbe & ctp,
              int whichFloat) const
```

Retrieves one of the floating-point values (if present) from the lattice cell pointed to by *ctp*. The parameter *whichFloat* indicates which of the floating-point values to return.

Examples

testPatternedSurface1/EventsAndActions.cpp.

getInt()

```
int KMCThinFilm::CellNeighProbe::getInt (
              const CellToProbe & ctp,
              int whichInt) const
```

Retrieves one of the integer values (if present) from the lattice cell pointed to by *ctp*. The parameter *whichInt* indicates which of the integer values to return.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, and testPatternedSurface2/I

6.6 KMCThinFilm::CellsToChange Class Reference

```
#include <CellsToChange.hpp>
```

Public Member Functions

- void addLatticePlanes (int numPlanesToAdd)
- const CellInds & getCellInds (int whichOffset) const
- double getFloat (int whichOffset, int whichFloat) const
- int getInt (int whichOffset, int whichInt) const
- void setCenter (const CellInds &ci)
- void setFloat (int whichOffset, int whichFloat, double val)
- void setInt (int whichOffset, int whichInt, int val)

Friends

- class **Simulation**

6.6.1 Detailed Description

Class used to change the lattice when using a function object satisfying the EventExecutorSemiManualTrack signature to execute an event.

Instances of this class contain offsets that define the relative positions of cells that would be directly changed by the event. These relative positions can be translated to actual positions by setting the *center* of a CellsToChange instance, that is, the absolute coordinates of the cell associated with the offset $(0,0,0)$ stored within the instance. For example, if the offsets are $(0,0,0)$, $(0,\pm1,0)$, $(\pm1,0,0)$, and $(0,0,\pm1)$, and the center is (a,b,c), then the absolute coordinates of the cells changed by the event would be (a,b,c), $(a,b\pm1,c)$, $(a\pm1,b,c)$, and $(a,b,c\pm1)$. Once the absolute coordinates of the cells are defined, the values in the lattice associated with these cells can be changed via setInt() and setFloat().

The offsets are specified through the *cnoVec* argument in the following member functions:

- EventExecutorGroup::addEventExecutor(int whichEvent, EventExecutorSemiManualTrack evExec, const std::vector<CellNeigh
- Simulation::addOverLatticeEvent(int, double, EventExecutorSemiManualTrack evExec, const std::vector<CellNeighOffsets> &

The offsets specified in the first element of *cnoVec* correspond to the offsets stored in the first element of the argument of type std::vector<CellsToChange>& in a EventExecutorSemiManualTrack function object, the offsets specified in the second element of *cnoVec* correspond to the offsets stored in the second element of the argument of type std↩
::vector<CellsToChange>& in that same function object, etc.

Examples

testBallisticDep2/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAı
and testPatternedSurface2/EventsAndActions.cpp.

6.6.2 Member Function Documentation

addLatticePlanes()

```
void KMCThinFilm::CellsToChange::addLatticePlanes (
              int numPlanesToAdd)
```

Adds zero or more layers to a given lattice, increasing the maximum value of the third lattice cell coordinate.

Note that if *numPlanesToAdd* is non-positive, this member function does nothing.

See also

Lattice::addPlanes()

Examples

testBallisticDep2/EventsAndActions.cpp.

getCellInds()

```
const CellInds & KMCThinFilm::CellsToChange::getCellInds (
              int whichOffset) const
```

Returns the absolute coordinates that become associated with the offset ID *whichOffset* once setCenter() has been called.

Obviously, setCenter() **MUST** be called before this function.

getFloat()

```
double KMCThinFilm::CellsToChange::getFloat (
              int whichOffset,
              int whichFloat) const
```

Returns the value of double-precision array element *whichFloat* at the lattice cell associated with offset ID *whichOffset*.

Given a lattice object instance `lattice` and a CellsToChange instance `ctc`, this function is essentially equivalent to `lattice.getFloat(ctc.getCellInds(whichOffset), whichFloat)`.

setCenter() **MUST** be called before this function.

See also

Lattice::getFloat()

getInt()

```
int KMCThinFilm::CellsToChange::getInt (
              int whichOffset,
              int whichInt) const
```

Returns the value of integer array element *whichInt* at the lattice cell associated with offset ID *whichOffset*.

Given a lattice object instance `lattice` and a CellsToChange instance `ctc`, this function is essentially equivalent to `lattice.getInt(ctc.getCellInds(whichOffset), whichInt)`

setCenter() **MUST** be called before this function.

See also

Lattice::getInt()

Examples

testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, and testPatternedSurface2/I

setCenter()

```
void KMCThinFilm::CellsToChange::setCenter (
              const CellInds & ci)
```

Sets to *ci* the absolute coordinates of the cell associated with the offset $(0,0,0)$ stored within a CellsToChange object.

Examples

testBallisticDep2/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAı
and testPatternedSurface2/EventsAndActions.cpp.

setFloat()

```
void KMCThinFilm::CellsToChange::setFloat (
              int whichOffset,
              int whichFloat,
              double val)
```

Sets the value of double-precision array element *whichFloat* at lattice cell associated with offset ID *whichOffset* to *val*.

setCenter() **MUST** be called before this function.

See also

Lattice::setFloat()

Examples

testBallisticDep2/EventsAndActions.cpp.

setInt()

```
void KMCThinFilm::CellsToChange::setInt (
              int whichOffset,
              int whichInt,
              int val)
```

Sets the value of integer array element *whichInt* at lattice cell associated with offset ID *whichOffset* to *val*.

setCenter() **MUST** be called before this function.

See also

Lattice::setInt()

Examples

testBallisticDep2/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAı
and testPatternedSurface2/EventsAndActions.cpp.

6.7 KMCThinFilm::CellToProbe Class Reference

`#include <CellNeighProbe.hpp>`

Public Member Functions

- const CellInds & inds ()

Friends

- class **CellNeighProbe**

6.7.1 Detailed Description

A largely opaque representation of a lattice cell for use with the CellNeighProbe class.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp and testPatternedSurface2/EventsAndActions.cpp.

6.7.2 Member Function Documentation

inds()

```
const CellInds & KMCThinFilm::CellToProbe::inds ()  [inline]
```

Returns the indices of the lattice cell pointed to by an instance of CellToProbe.

Examples

testPatternedSurface2/EventsAndActions.cpp.

6.8 KMCThinFilm::EventExecutorGroup Class Reference

`#include <EventExecutorGroup.hpp>`

Public Member Functions

- EventExecutorGroup (int numEventsInGroup)
- void addEventExecutor (int whichEvent, EventExecutorAutoTrack evExec)
- void addEventExecutor (int whichEvent, EventExecutorSemiManualTrack evExec, const std::vector< CellNeighOffsets > &cnoVec)
- void clearGroup ()
- int numEventExecutors () const
- void resetGroup (int numEventsInGroup)

Friends

- class **Simulation**

6.8.1 Detailed Description

Collection of function objects satisfying the EventExecutorAutoTrack and EventExecutorSemiManualTrack signatures, used to specify a group of possible cell-centered events.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

6.8.2 Constructor & Destructor Documentation

EventExecutorGroup()

```
KMCThinFilm::EventExecutorGroup::EventExecutorGroup (
              int numEventsInGroup)  [explicit]
```

Constructs an EventExecutorGroup object.

Parameters

numEventsInGroup	Number of function objects in group that execute events

6.8.3 Member Function Documentation

addEventExecutor() [1/2]

```
void KMCThinFilm::EventExecutorGroup::addEventExecutor (
              int whichEvent,
              EventExecutorAutoTrack evExec)
```

Adds an EventExecutorAutoTrack object to the group

Parameters

whichEvent	Integer ID of event in group. Must be >= 0 and less than numEventExecutors()
evExec	Function object to execute the event

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

addEventExecutor() [2/2]

```
void KMCThinFilm::EventExecutorGroup::addEventExecutor (
              int whichEvent,
              EventExecutorSemiManualTrack evExec,
              const std::vector< CellNeighOffsets > & cnoVec)
```

Adds an EventExecutorSemiManualTrack object and its associated offsets to the group.

The argument *cnoVec* indicates the relative positions of cells that would be directly changed by executing *evExec*. If, for example, the event changed cells (i_1, j_1, k_1) and $(i_1 + a, j_1 + b, k_1 + c)$, and also (i_2, j_2, k_2), $(i_2 + d_1, j_2 + e_1, k_2 + f_1)$, and $(i_2 + d_2, j_2 + e_2, k_2 + f_2)$, where the relative positions of (i_1, j_1, k_1) and (i_2, j_2, k_2) may vary with respect to each other, then the first entry of *cnoVec* would be the offsets associated with (i_1, j_1, k_1), that is, $(0,0,0)$ and (a,b,c), and the second entry of *cnoVec* would be the offsets associated with (i_2, j_2, k_2), that is, $(0,0,0)$, (d_1, e_1, f_1) and (d_2, e_2, f_2).

Parameters

whichEvent	Integer ID of event in group. Must be >= 0 and less than numEventExecutors()

Parameters

evExec	Function object to execute the event
cnoVec	Relative positions of cells directly changed by event.

clearGroup()

```
void KMCThinFilm::EventExecutorGroup::clearGroup ()
```

Removes all previous function objects from the group.

numEventExecutors()

```
int KMCThinFilm::EventExecutorGroup::numEventExecutors () const
```

Returns the number of function objects stored in the group.

resetGroup()

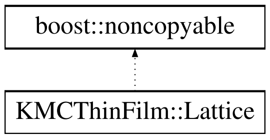
```
void KMCThinFilm::EventExecutorGroup::resetGroup (
              int numEventsInGroup)
```

Removes all previous function objects from the group and allocates memory for *numEventsInGroup* function objects.

6.9 KMCThinFilm::Lattice Class Reference

`#include <Lattice.hpp>`

Inheritance diagram for KMCThinFilm::Lattice:



Public Member Functions

- Lattice (const LatticeParams &paramsForLattice)
- void addPlanes (int numPlanesToAdd)
- bool addToExportBufferIfNeeded (const CellInds &ci)
- void clearGhostsToSend ()
- const MPI_Comm & comm () const
- int commCoord (int dim) const
- int currHeight () const
- double getFloat (const CellInds &ci, int whichFloat) const
- void getGlobalPlanarBBox (LatticePlanarBBox &bbox) const
- int getInt (const CellInds &ci, int whichInt) const
- void getLocalPlanarBBox (bool wGhost, LatticePlanarBBox &bbox) const
- const std::vector< std::vector< IJK > > & getReceivedGhostInds () const
- const std::vector< std::vector< IJK > > & getReceivedLocalInds () const

- void getSectorPlanarBBox (int sectNum, LatticePlanarBBox &bbox) const
- int ghostExtent (int dim) const
- int nFloatsPerCell () const
- int nIntsPerCell () const
- int nProcs () const
- int numSectors () const
- int planesReserved () const
- int procID () const
- int procPerDim (int dim) const
- void recvGhosts (int sectNum)
- void recvGhostsUpdate (int sectNum)
- void reservePlanes (int numTotalPlanesToReserve)
- int sectorOfIndices (const CellInds &ci) const
- void sendGhosts (int sectNum)
- void sendGhostsUpdate (int sectNum)
- void setFloat (const CellInds &ci, int whichInt, double val)
- void setInt (const CellInds &ci, int whichInt, int val)
- int wrapI (const CellInds &ci) const
- int wrapJ (const CellInds &ci) const

Friends

- class **Simulation**

6.9.1 Detailed Description

A lattice that may be distributed over several processors.

This lattice is designed to be simple but general. It is represented as a three-dimensional array, which can not only be used for square or simple cubic lattices, but also lattices of other physical shapes, such as the part of layer *k* of a simple hexagonal lattice shown below:

```
    (0,1,k)-----(1,1,k)----(2,1,k)----(3,1,k)---(4,1,k)
      / \        / \        / \        / \        /
     /   \      /   \      /   \      /   \      /
    /     \    /     \    /     \    /     \    /
   /       \  /       \  /       \  /       \  /
  /_________\/_________\/_________\/_________\/
(0,0,k)   (1,0,k)   (2,0,k)     (3,0,k)   (4,0,k)
```

For further flexibility, each site may effectively contain an array of integers and/or an array of double-precision floating point numbers. The lengths of these arrays is the same for all sites on the lattice.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep1/EventsAndActions.hpp, testBallisticDep1/InitLattice.cpp, testBallisticDep1/InitLattice.hpp, testBallisticDep2/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.hpp, testFractal/EventsAndActions.cpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/Even
testPatternedSurface1/InitLattice.cpp, testPatternedSurface1/InitLattice.hpp, testPatternedSurface2/EventsAndActions.cpp, and testPatternedSurface2/EventsAndActions.hpp.

6.9.2 Constructor & Destructor Documentation

Lattice()

```
KMCThinFilm::Lattice::Lattice (
                const LatticeParams & paramsForLattice)
```

[**ADVANCED**] Constructor of a Lattice object

Unless one is debugging or testing an application that uses ARL KMCThinFilm, there is little point in calling this directly, since when a Simulation object is constructed, it builds a Lattice object internally for its own use.

6.9.3 Member Function Documentation

addPlanes()

```
void KMCThinFilm::Lattice::addPlanes (
                int numPlanesToAdd)
```

Adds zero or more layers to a given lattice, increasing the maximum value of the third lattice cell coordinate.

Note that if *numPlanesToAdd* is non-positive, this member function does nothing. This behavior may be used to conditionally add lattice planes. For example,

```
// Lattice object named "lattice" defined above

CellInds ci;

// Misc. calcs ...

ci.k = someFunction(...);
kMax = lattice.currHeight() - 1;

lattice.addPlanes(ci.k - kMax); // Only adds a lattice plane if ci.k exceeds kMax

lattice.setInt(ci, whichInt, anotherFunction(...));
```

Examples

testBallisticDep1/EventsAndActions.cpp, and testPatternedSurface1/InitLattice.cpp.

addToExportBufferIfNeeded()

```
bool KMCThinFilm::Lattice::addToExportBufferIfNeeded (
                const CellInds & ci)
```

[**ADVANCED**] Adds *ci* to the lists of CellInds objects to be sent to other processors, provided that *ci* is either in a ghost region or in the part of the boundary of the lattice that is sent to other processors. Returns true if *ci* is in a ghost region.

This should typically be used between a pair of calls to recvGhostsUpdate(i) and sendGhostsUpdate(i), where the value of *i* is the same for both calls. If ghosts are never exported, then clearGhostsToSend() should be used in place of sendGhostsUpdate(), and calls to addToExportBufferIfNeeded() may be called before any calls to recvGhostsUpdate().

clearGhostsToSend()

```
void KMCThinFilm::Lattice::clearGhostsToSend ()
```

[**ADVANCED**] Clears any ghosts that would be sent by a call to sendGhostsUpdate()

comm()

```
const MPI_Comm & KMCThinFilm::Lattice::comm () const
```

The MPI communicator used by the lattice for interchange of ghosts, if the preprocessor variable KMC_PARALLEL is non-zero. **Not available in the serial version of the ARL KMCThinFilm library.**

This is *not* the same as the *latticeCommInitial* member of the LatticeParams object used to construct the lattice. While it contains the same number of processors as *latticeCommInitial*, the lines of code

```
int rank;
LatticeParams latParams;
MPI_Comm_rank(latParams.latticeCommInitial, &rank);
```

and

```
// Lattice object named "lattice" defined above
int rank;
MPI_Comm_rank(lattice.comm(), &rank);
```

may not return the same value for *rank*.

commCoord()

```
int KMCThinFilm::Lattice::commCoord (
              int dim) const
```

Coordinates for a processor in an MPI Cartesian topology along in-plane dimension *dim* for a given processor.

For example, if the lattice is partitioned as follows,

```
*-----*-----*-----*
|  3  |  4  |  5  |
*-----*-----*-----*
|  0  |  1  |  2  |
*-----*-----*-----*
```

where the numbers 0, 1, 2, etc., denote processor ranks, then if the MPI rank of a processor is 4, then commCoord(0) returns a value of 1, and commCoord(0) returns a value of 2. The coordinates for all processor ranks in the above partitioned lattice are shown below:

```
*-----*-----*-----*
|(1,0)|(1,1)|(1,2)|
*-----*-----*-----*
|(0,0)|(0,1)|(0,2)|
*-----*-----*-----*
```

Note that the coordinates described here are *not* the same as the coordinates of a lattice cell.

See also

procPerDim()

currHeight()

```
int KMCThinFilm::Lattice::currHeight () const
```

The current number of lattice planes.

This is also one larger than the maximum value of the third lattice coordinate.

See also

getLocalPlanarBBox() getSectorPlanarBBox()

Examples

testBallisticDep1/EventsAndActions.cpp, and testBallisticDep2/EventsAndActions.cpp.

getFloat()

```
double KMCThinFilm::Lattice::getFloat (
              const CellInds & ci,
              int whichFloat) const
```

Returns the value of double-precision array element *whichFloat* at lattice cell indices *ci*.

The value of *whichFloat* ranges from 0 to nFloatsPerCell() - 1.

See also

getInt() setFloat() setInt()

Examples

testBallisticDep1/EventsAndActions.cpp, and testBallisticDep2/EventsAndActions.cpp.

getGlobalPlanarBBox()

```
void KMCThinFilm::Lattice::getGlobalPlanarBBox (
              LatticePlanarBBox & bbox) const
```

Obtain limiting values of the global lattice coordinates.

"BBox" here is short for "bounding box."

Note that this function in general *cannot* be used to determine lattice coordinate values for use with getInt(), setInt(), getFloat(), and setFloat(), since those functions require local lattice coordinates. To obtain limiting values for local lattice coordinates, use getLocalPlanarBBox() or getSectorPlanarBBox() instead.

See also

getLocalPlanarBBox() getSectorPlanarBBox()

Parameters

bbox	Bounding box of the global lattice coordinates.

Examples

testPatternedSurface1/InitLattice.cpp.

getInt()

```
int KMCThinFilm::Lattice::getInt (
              const CellInds & ci,
              int whichInt) const
```

Returns the value of integer array element *whichInt* at lattice cell indices *ci*.

The value of *whichInt* ranges from 0 to nIntsPerCell() - 1.

See also

getFloat() setFloat() setInt()

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, and testPatternedSurface2/I

getLocalPlanarBBox()

```
void KMCThinFilm::Lattice::getLocalPlanarBBox (
                bool wGhost,
                LatticePlanarBBox & bbox) const
```

Obtain limiting values of the local lattice coordinates.

"BBox" here is short for "bounding box."

A typical use for this member function would be something like this:

```
// Lattice object named "lattice" defined above

bool includeGhosts = myBooleanFunc(...);

LatticePlanarBBox localPlanarBBox;
lattice.getLocalPlanarBBox(includeGhosts, localPlanarBBox);
kMaxP1 = lattice.currHeight();

CellInds ci;
for (ci.k = 0; ci.k < kMaxP1; ++(ci.k)) {
   for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
      for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {

         // Some calcs ...

         lattice.setInt(ci, WHICH_INT, someFunc(...));

         // Other calcs ...

     }
   }
}
```

See also

getSectorPlanarBBox() getGlobalPlanarBBox()

Parameters

wGhost	Indicates whether minimum and maximum values for lattice cell coordinates include the coordinates of ghost lattice cells.
bbox	Bounding box of the local lattice coordinates.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/InitL
and testPatternedSurface2/EventsAndActions.cpp.

getReceivedGhostInds()

```
const std::vector< std::vector< IJK > > & KMCThinFilm::Lattice::getReceivedGhostInds () const
```

[**ADVANCED**] Returns the lists of the indices of cells that are changed by the latest call to recvGhostsUpdate().

getReceivedLocalInds()

```
const std::vector< std::vector< IJK > > & KMCThinFilm::Lattice::getReceivedLocalInds () const
```

[**ADVANCED**] Returns the lists of the indices of cells that are changed by the latest call to sendGhostsUpdate().

getSectorPlanarBBox()

```
void KMCThinFilm::Lattice::getSectorPlanarBBox (
               int sectNum,
               LatticePlanarBBox & bbox) const
```

Obtain limiting values of the local lattice coordinates for a particular sector or sublattice.

"BBox" here is short for "bounding box."

A typical use for this member function would be something like this:

```
// Lattice object named "lattice" defined above

for (int sectNum = 0; i < lattice.numSectors(); ++i) {

   lattice.recvGhosts(sectNum);

   LatticePlanarBBox sectorPlanarBBox;
   lattice.getSectorPlanarBBox(sectNum, sectorPlanarBBox);
   kMaxP1 = lattice.currHeight();

   CellInds ci;
   for (ci.k = 0; ci.k < kMaxP1; ++(ci.k)) {
      for (ci.i = sectorPlanarBBox.imin; ci.i < sectorPlanarBBox.imaxP1; ++(ci.i)) {
         for (ci.j = sectorPlanarBBox.jmin; ci.j < sectorPlanarBBox.jmaxP1; ++(ci.j)) {
            // Some calcs ...

            lattice.setFloat(ci, FLT_VAL1, someFunc(...));

            // Other calcs ...

         }
      }
   }

   lattice.sendGhosts(sectNum);
}
```

Note that unlike getLocalPlanarBBox(), the minimum and maximum lattice coordinate values do not include the coordinates of ghost lattice cells.

See also

getLocalPlanarBBox() getGlobalPlanarBBox() numSectors()

Parameters

sectNum	Sector number, ranging from 0 to numSectors() - 1
bbox	Bounding box of the lattice coordinates within the sector.

ghostExtent()

```
int KMCThinFilm::Lattice::ghostExtent (
               int dim) const
```

Thickness of the ghost region along in-plane dimension *dim*.

Dimension 0 is the dimension along which the first lattice cell index varies, and dimension 1 is the dimension along which the second lattice cell index varies.

See also

commCoord()

nFloatsPerCell()

```
int KMCThinFilm::Lattice::nFloatsPerCell () const
```

Length of the double-precision floating-point array at each lattice cell.

nIntsPerCell()

```
int KMCThinFilm::Lattice::nIntsPerCell () const
```

Length of the integer array at each lattice cell.

nProcs()

```
int KMCThinFilm::Lattice::nProcs () const
```

Number of processors over which the lattice is distributed.

numSectors()

```
int KMCThinFilm::Lattice::numSectors () const
```

If the preprocessor variable KMC_PARALLEL is zero, then this always returns 1. Otherwise, this is the number of sectors (or sublattices) used in the approximate parallel Kinetic Monte Carlo algorithm used by this library.

For a discussion of what these sectors are, see Y. Shim and J. G. Amar, "Semirigorous synchronous sublattice algorithm for parallel Kinetic Monte Carlo simulation of thin film growth", Physical Review B, vol. 71, 125432 (2005) or S. Plimpton et al., "Crossing the Mesoscale No-Man's Land via Parallel Kinetic Monte Carlo", Sandia National Laboratories Technical Report SAND2009-6226 (2009).

planesReserved()

```
int KMCThinFilm::Lattice::planesReserved () const
```

Returns the number of planes reserved for the lattice (but not necessarily added to the lattice yet).

Note that while this should be at least the number of planes that were explicitly reserved, it may be more than that.

procID()

```
int KMCThinFilm::Lattice::procID () const
```

When the preprocessor variable KMC_PARALLEL is zero, this always returns zero. Otherwise, this is the MPI rank of the local process, where the communicator for that process is given by Lattice::comm().

procPerDim()

```
int KMCThinFilm::Lattice::procPerDim (
              int dim) const
```

The number of processors along in-plane dimension *dim*.

For example, if the lattice is partitioned as follows,

```
*-----*-----*-----*
|  3  |  4  |  5  |
*-----*-----*-----*
|  0  |  1  |  2  |
*-----*-----*-----*
```

then procPerDim(0) returns a value of 3, and procPerDim(2) returns a value of 2.

See also

commCoord()

recvGhosts()

```
void KMCThinFilm::Lattice::recvGhosts (
              int sectNum)
```

[**ADVANCED**] Update *all* the ghost lattice cells from data received from other processors.

This member function does not keep track of which lattice sites are actually changed during the update, unlike recvGhostsUpdate().

Parameters

sectNum	Sector number, ranging from 0 to numSectors() - 1

recvGhostsUpdate()

```
void KMCThinFilm::Lattice::recvGhostsUpdate (
              int sectNum)
```

[**ADVANCED**] Updates ghost lattice cells from data received from other processors, but only updates the cells that are supposed to have actually changed. Processors use addToExportBufferIfNeeded() to mark the cells that will be received by a call to this function.

Parameters

sectNum	Sector number, ranging from 0 to numSectors() - 1

reservePlanes()

```
void KMCThinFilm::Lattice::reservePlanes (
              int numTotalPlanesToReserve)
```

[**ADVANCED**] Reserves space in memory for adding planes to a lattice*, but does not actually add any additional planes. **Almost certainly unnecessary if LatticeParams::numPlanesToReserve has been set to an appropriate value**.

The relationship between reservePlanes() and addPlanes() is somewhat analogous to that of the std::vector member functions reserve() and push_back(). The former only allocates space in memory, while the latter is responsible for appending actual values (i.e. lattice planes). Note that if *numTotalPlanesToReserve* is too small, it is not an error, though it may affect performance somewhat.

*Or at least pointers for those planes.

See also

addPlanes()

sectorOfIndices()

```
int KMCThinFilm::Lattice::sectorOfIndices (
              const CellInds & ci) const
```

Returns the sector to which a set of cell indices *ci* belongs.

If *ci* is within a ghost region, then this function returns -1.

In serial, this always returns zero.

sendGhosts()

```
void KMCThinFilm::Lattice::sendGhosts (
            int sectNum)
```

[**ADVANCED**] Send data from *all* the ghost lattice cells to update the off-process cells to which the ghosts correspond.

This member function does not keep track of which lattice sites are actually changed during the update, unlike sendGhostsUpdate().

Parameters

sectNum	Sector number, ranging from 0 to numSectors() - 1

sendGhostsUpdate()

```
void KMCThinFilm::Lattice::sendGhostsUpdate (
            int sectNum)
```

[**ADVANCED**] Sends data from the ghost lattice cells to update the (usually off-process) cells to which the ghosts correspond, but only updates the cells that are supposed to have actually changed. Processors use addToExportBufferIfNeeded() to mark the cells that will be received by a call to this function.

Parameters

sectNum	Sector number, ranging from 0 to numSectors() - 1

setFloat()

```
void KMCThinFilm::Lattice::setFloat (
            const CellInds & ci,
            int whichInt,
            double val)
```

Sets to *val* the value of double-precision array element *whichFloat* at lattice cell indices *ci*.

The value of *whichFloat* ranges from 0 to nFloatsPerCell() - 1.

See also

getFloat() getInt() setInt()

Examples

testBallisticDep1/EventsAndActions.cpp, and testPatternedSurface1/InitLattice.cpp.

setInt()

```
void KMCThinFilm::Lattice::setInt (
            const CellInds & ci,
            int whichInt,
            int val)
```

Sets to *val* the value of integer array element *whichInt* at lattice cell indices *ci*.

The value of *whichInt* ranges from 0 to nIntsPerCell() - 1.

See also

getFloat() getInt() setFloat()

Examples

testBallisticDep1/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, and testPatternedSurface2/EventsAndActions.cpp.

wrapI()

```
int KMCThinFilm::Lattice::wrapI (
              const CellInds & ci) const
```

Returns a wrapped version of ci.i to account for periodic boundary conditions.

This is likely to only be useful in serial simulations.

Furthermore, the member functions getInt(), getFloat(), setInt(), and setFloat() *already* account for periodic boundary conditions, so this function is mostly needed for circumstances where one is doing arithmetic of cell indices.

Examples

testBallisticDep2/EventsAndActions.cpp.

wrapJ()

```
int KMCThinFilm::Lattice::wrapJ (
              const CellInds & ci) const
```

Returns a wrapped version of ci.j to account for periodic boundary conditions.

This is likely to only be useful in serial simulations, though it perhaps may be of use if row-based decomposition is used in parallel simulations.

Furthermore, the member functions getInt(), getFloat(), setInt(), and setFloat() *already* account for periodic boundary conditions, so this function is mostly needed for circumstances where one is doing arithmetic of cell indices.

Examples

testBallisticDep2/EventsAndActions.cpp.

6.10 KMCThinFilm::LatticeParams Struct Reference

```
#include <Lattice.hpp>
```

Public Types

- enum ParallelDecomp { COMPACT , ROW }

Public Attributes

- boost::array< int, 2 > ghostExtent
- boost::array< int, 2 > globalPlanarDims
- LatticeInitializer latInit
- MPI_Comm latticeCommInitial
- bool noAddingPlanesDuringSimulation
- int numFloatsPerCell
- int numIntsPerCell

- int numPlanesToReserve
- ParallelDecomp parallelDecomp
- SetEmptyCellVals setEmptyCellVals

6.10.1 Detailed Description

Parameter object used in constructing a Lattice object

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

6.10.2 Member Enumeration Documentation

ParallelDecomp

```
enum KMCThinFilm::LatticeParams::ParallelDecomp
```

Methods of parallel decomposition

Enumerator

COMPACT	The lattice is decomposed such that the perimeter of the part of the lattice owned by each process is a minimum.
ROW	The lattice is decomposed into strips of size globalPlanarDims[0]/N by globalPlanarDims[1], where N is the number of processors.

6.10.3 Member Data Documentation

ghostExtent

```
boost::array<int,2> KMCThinFilm::LatticeParams::ghostExtent
```

Array of length two, indicating the size (in lattice cells) of the ghost region along each in-plane edge of the lattice domain in a parallel simulation. **Has no effect in a serial simulation.**

Examples

testFractal_parallel/testFractal.cpp.

globalPlanarDims

```
boost::array<int,2> KMCThinFilm::LatticeParams::globalPlanarDims
```

Array of length two, indicating the number of lattice cells along the in-plane edges of the domain of the lattice.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

latInit

```
LatticeInitializer KMCThinFilm::LatticeParams::latInit
```

Function or function object used to initialize the lattice. Defaults to an AddEmptyPlanes object that adds a single lattice plane.

Examples

testPatternedSurface1/testPatternedSurface.cpp.

latticeCommInitial

`MPI_Comm KMCThinFilm::LatticeParams::latticeCommInitial`

When the preprocessor variable KMC_PARALLEL is non-zero, this is the MPI communicator used to initialize an instance of the Lattice class, and it defaults to MPI_COMM_WORLD. **Not available in the serial version of the ARL KMCThinFilm library.**

noAddingPlanesDuringSimulation

`bool KMCThinFilm::LatticeParams::noAddingPlanesDuringSimulation`

When the preprocessor variable KMC_PARALLEL is non-zero, this indicates that no planes will be added during the simulation, in order to avoid unnecessary parallel communication when this is the case. Planes, then, may only be added during lattice initialization. **Has no effect in a serial simulation.**

numFloatsPerCell

`int KMCThinFilm::LatticeParams::numFloatsPerCell`

Size of the floating-point array at each lattice cell. Defaults to zero.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatternedSurface.cpp.

numIntsPerCell

`int KMCThinFilm::LatticeParams::numIntsPerCell`

Size of the integer array at each lattice cell. Defaults to zero.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

numPlanesToReserve

`int KMCThinFilm::LatticeParams::numPlanesToReserve`

Number of planes (or at least pointers for those planes) for which to reserve space in memory. Note that this is not the number of planes actually added to a lattice. Actually adding planes is done by the Lattice::addPlanes() member function. The relationship between this parameter and Lattice::addPlanes() is somewhat analogous to that of the std↩
::vector member functions reserve() and push_back(). If this parameter is too small, it is not an error, but it may affect performance somewhat.

Examples

testBallisticDep1/testBallisticDep.cpp, and testBallisticDep2/testBallisticDep.cpp.

parallelDecomp

`ParallelDecomp KMCThinFilm::LatticeParams::parallelDecomp`

Indicates the method of parallel decomposition in a parallel KMC simulation. **Has no effect in a serial simulation.**

Examples

testFractal_parallel/testFractal.cpp.

setEmptyCellVals

`SetEmptyCellVals KMCThinFilm::LatticeParams::setEmptyCellVals`

This is empty by default, and does not need to be defined if the integer and floating-point numbers in an empty lattice cell are all supposed to be zero. If set, this function is used to set the integers and floating-point numbers in an empty lattice cell to their proper values. Note that when this function is used, the vectors *emptyIntVals* and *emptyFloatVals* have already had their sizes set to the number of integer values and number of floating-point values per lattice cell, respectively.

Examples

testBallisticDep1/testBallisticDep.cpp, and testBallisticDep2/testBallisticDep.cpp.

6.11 KMCThinFilm::LatticePlanarBBox Struct Reference

`#include <Lattice.hpp>`

Public Attributes

- int imaxP1
- int imin
- int jmaxP1
- int jmin

6.11.1 Detailed Description

Bounding box of in-plane values of lattice cell coordinates.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.hpp, testBallisticDep2/testBallisticDep.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/InitLattice.cpp, and testPatternedSurface2/EventsAndActions

6.11.2 Member Data Documentation

imaxP1

`int KMCThinFilm::LatticePlanarBBox::imaxP1`

One more than the maximum value of the first lattice cell coordinate

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/InitL and testPatternedSurface2/EventsAndActions.cpp.

imin

`int KMCThinFilm::LatticePlanarBBox::imin`

Minimum value of the first lattice cell coordinate.

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/InitL
and testPatternedSurface2/EventsAndActions.cpp.

jmaxP1

```
int KMCThinFilm::LatticePlanarBBox::jmaxP1
```

One more than the maximum value of the second lattice cell coordinate

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/InitL
and testPatternedSurface2/EventsAndActions.cpp.

jmin

```
int KMCThinFilm::LatticePlanarBBox::jmin
```

Minimum value of the second lattice cell coordinate

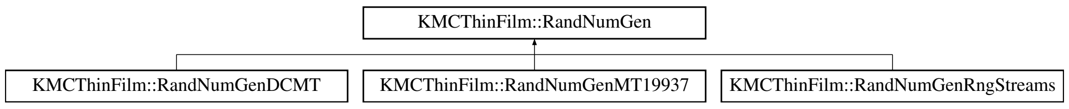
Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.cpp, testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/InitL
and testPatternedSurface2/EventsAndActions.cpp.

6.12 KMCThinFilm::RandNumGen Class Reference

`#include <RandNumGen.hpp>`

Inheritance diagram for KMCThinFilm::RandNumGen:


KMCThinFilm::RandNumGen
KMCThinFilm::RandNumGenDCMT
KMCThinFilm::RandNumGenMT19937
KMCThinFilm::RandNumGenRngStreams

Public Member Functions

- virtual double getNumInOpenIntervalFrom0To1 ()=0

6.12.1 Detailed Description

Abstract class to implement wrapper classes for random number generators

6.12.2 Member Function Documentation

getNumInOpenIntervalFrom0To1()

```
virtual double KMCThinFilm::RandNumGen::getNumInOpenIntervalFrom0To1 ()  [pure virtual]
```

Returns a (pseudo)random number between 0 and 1, **NOT** including either 0 or 1.

It is especially important for implementations of this class to exclude 0 from this function's return value. The amount of time by which a sector's clock is incremented is proportional to log(r), where r is the return value of the function, and of course, log(0) is undefined.

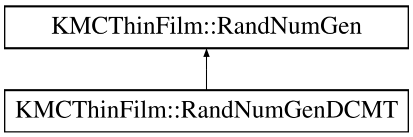
It is also ill-adviced to allow 1 to be a return value of this function, either, since log(1) = 0, which would imply that an event in a KMC simulation took no time at all.

Implemented in KMCThinFilm::RandNumGenDCMT, KMCThinFilm::RandNumGenMT19937, and KMCThinFilm::RandNumGenRn

6.13 KMCThinFilm::RandNumGenDCMT Class Reference

`#include <RandNumGenDCMT.hpp>`

Inheritance diagram for KMCThinFilm::RandNumGenDCMT:



Public Types

- enum Period {
  P521 = 521 , **P607** = 607 , **P1279** = 1279 , **P2203** = 2203 ,
  P2281 = 2281 , **P3217** = 3217 , **P4253** = 4253 , **P4423** = 4423 ,
  P9689 = 9689 , **P9941** = 9941 , **P11213** = 11213 , **P19937** = 19937 ,
  P21701 = 21701 , **P23209** = 23209 , **P44497** = 44497 }

Public Member Functions

- RandNumGenDCMT (int rank, uint32_t seed, uint32_t seed_perProc, Period period=P521)
- virtual double getNumInOpenIntervalFrom0To1 ()

6.13.1 Detailed Description

Wrapper class for the parallel pseudorandom number generator library DCMT: "Dynamic Creator of Mersenne Twisters."

DCMT is a parallel pseudorandom number generator by Makoto Matsumoto and Takuji Nishimura, and it is based on the serial Mersenne Twister pseudorandom number generator. The library may be obtained from this web page: `http://www.math.sci.hiroshima-u.ac.jp/~m-mat/MT/DC/dc.html`

Examples

testFractal_parallel/testFractal.cpp.

6.13.2 Member Enumeration Documentation

Period

`enum KMCThinFilm::RandNumGenDCMT::Period`

This enumeration forces the *period* argument of the RandNumGenDCMT constructor to be one of a certain number of discrete values.

When the value of this enumeration is Pn, where n is a number, then the period of the pseudorandom number stream is $2^n - 1$. The value of n is such that the period is a Mersenne prime.

6.13.3 Constructor & Destructor Documentation

RandNumGenDCMT()

```
KMCThinFilm::RandNumGenDCMT::RandNumGenDCMT (
              int rank,
              uint32_t seed,
              uint32_t seed_perProc,
              Period period = P521)
```

Constructor, sets the initial state for each of the parallel streams of the pseudorandom number generator.

Parameters

rank	MPI rank of the processor
seed	Global seed for the pseudorandom number generator
seed_perProc	Per-processor seed for the pseudorandom number generator, should be different for each process
period	Enumeration constant indicating the exponent for the period of the pseudorandom number stream

6.13.4 Member Function Documentation

getNumInOpenIntervalFrom0To1()

```
virtual double KMCThinFilm::RandNumGenDCMT::getNumInOpenIntervalFrom0To1 ()  [virtual]
```

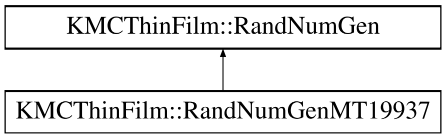
Returns a (pseudo)random number between 0 and 1, not including either 0 or 1.

Implements KMCThinFilm::RandNumGen.

6.14 KMCThinFilm::RandNumGenMT19937 Class Reference

`#include <RandNumGenMT19937.hpp>`

Inheritance diagram for KMCThinFilm::RandNumGenMT19937:



Public Member Functions

- RandNumGenMT19937 (unsigned int seed)
- virtual double getNumInOpenIntervalFrom0To1 ()

6.14.1 Detailed Description

Wrapper class for the *serial* Mersenne Twister pseudorandom number generator with a period of $2^{19937} - 1$.

This is not recommended for parallel simulations.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

6.14.2 Constructor & Destructor Documentation

RandNumGenMT19937()

```
KMCThinFilm::RandNumGenMT19937::RandNumGenMT19937 (
              unsigned int seed)  [inline]
```

Constructor, sets the initial state for the pseudorandom number generator.

6.14.3 Member Function Documentation

getNumInOpenIntervalFrom0To1()

```
virtual double KMCThinFilm::RandNumGenMT19937::getNumInOpenIntervalFrom0To1 ()  [virtual]
```

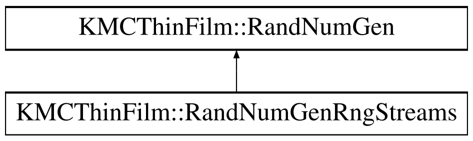
Returns a (pseudo)random number between 0 and 1, not including either 0 or 1.

Implements KMCThinFilm::RandNumGen.

6.15 KMCThinFilm::RandNumGenRngStreams Class Reference

`#include <RandNumGenRngStreams.hpp>`

Inheritance diagram for KMCThinFilm::RandNumGenRngStreams:



Public Member Functions

- RandNumGenRngStreams (std::vector< unsigned long > &seed, int rank, long advExp=127, long advConst=0)
- virtual double getNumInOpenIntervalFrom0To1 ()

6.15.1 Detailed Description

Wrapper class for the parallel pseudorandom number generator library RngStreams.

RngStreams is a parallel pseudorandom number generator by Pierre L'Ecuyer and Richard Simard. It may be obtained from this web page: `http://statmath.wu.ac.at/software/RngStreams/`

6.15.2 Constructor & Destructor Documentation

RandNumGenRngStreams()

```
KMCThinFilm::RandNumGenRngStreams::RandNumGenRngStreams (
              std::vector< unsigned long > & seed,
              int rank,
              long advExp = 127,
              long advConst = 0)
```

Constructor, sets the initial state for each of the parallel streams of the pseudorandom number generator.

The vector *seed* that seeds the pseudorandom number generator must have a length of 6, and it should be the same for every process.

Each parallel pseudorandom number stream is offset by $rank \times (2^{advExp} + advConst)$, if $advExp$ is nonnegative, or $rank \times (-2^{-advExp} + advConst)$, otherwise. The argument *rank* should be the MPI rank of the processor.

6.15.3 Member Function Documentation

getNumInOpenIntervalFrom0To1()

```
virtual double KMCThinFilm::RandNumGenRngStreams::getNumInOpenIntervalFrom0To1 ()  [virtual]
```

Returns a (pseudo)random number between 0 and 1, not including either 0 or 1.

Implements KMCThinFilm::RandNumGen.

6.16 KMCThinFilm::TimeIncr::SchemeVars Class Reference

`#include <TimeIncrSchemeVars.hpp>`

Public Member Functions

- SchemeName::Type getSchemeName () const
- double getSchemeParamIfAvailable (SchemeParam::Type paramName, bool &isAvailable) const
- double getSchemeParamOrDie (SchemeParam::Type paramName, const std::string &msgIfDie) const
- double getSchemeParamOrReturnDefaultVal (SchemeParam::Type paramName, double defaultVal) const
- void setSchemeName (SchemeName::Type name)
- void setSchemeParam (SchemeParam::Type paramName, double paramVal)

6.16.1 Detailed Description

Type of object used to store parameters for the parallel time stepping schemes.

Note that in a typical use of a class instance of this, the set∗() functions will tend to be used rather than the get∗() functions. However, the get∗() functions are exposed for the sake of testing and debugging.

Examples

testFractal_parallel/testFractal.cpp.

6.16.2 Member Function Documentation

getSchemeName()

```
SchemeName::Type KMCThinFilm::TimeIncr::SchemeVars::getSchemeName () const
```

Returns the name of the time-stepping scheme set by setSchemeName().

getSchemeParamIfAvailable()

```
double KMCThinFilm::TimeIncr::SchemeVars::getSchemeParamIfAvailable (
             SchemeParam::Type paramName,
             bool & isAvailable) const
```

Returns the value of the parameter, if it has been set.

It is not an error if the parameter was not set, but if it wasn't, then the return value is likely garbage.

Parameters

paramName	Name of parameter

Parameters

isAvailable	If this is false, then the parameter was not set.

getSchemeParamOrDie()

```
double KMCThinFilm::TimeIncr::SchemeVars::getSchemeParamOrDie (
              SchemeParam::Type paramName,
              const std::string & msgIfDie) const
```

Returns the value of the parameter or causes the KMC application to terminate with an error message.

Parameters

paramName	Name of parameter
msgIfDie	Error message printed if the parameter was not set.

getSchemeParamOrReturnDefaultVal()

```
double KMCThinFilm::TimeIncr::SchemeVars::getSchemeParamOrReturnDefaultVal (
              SchemeParam::Type paramName,
              double defaultVal) const
```

Returns the value of the parameter if it has been set, and returns a default value otherwise.

Parameters

paramName	Name of parameter
defaultVal	The value to be returned if the parameter was not set.

setSchemeName()

```
void KMCThinFilm::TimeIncr::SchemeVars::setSchemeName (
              SchemeName::Type name)
```

Sets the name of the time-stepping scheme to be used.

Examples

testFractal_parallel/testFractal.cpp.

setSchemeParam()

```
void KMCThinFilm::TimeIncr::SchemeVars::setSchemeParam (
              SchemeParam::Type paramName,
              double paramVal)
```

Sets a parameter of the time-stepping scheme to be used.

Parameters

paramName	Name of parameter
paramVal	Value of parameter

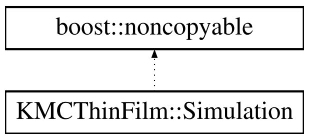
Examples

testFractal_parallel/testFractal.cpp.

6.17 KMCThinFilm::Simulation Class Reference

`#include <Simulation.hpp>`

Inheritance diagram for KMCThinFilm::Simulation:



Public Member Functions

- Simulation (const LatticeParams &paramsForLattice)
- void addCellCenteredEventGroup (int eventGroupId, const CellNeighOffsets &cno, CellCenteredGroupPropensities propensities, const EventExecutorGroup &eventExecutorGroup)
- void addOverLatticeEvent (int eventId, double propensityPerUnitArea, EventExecutorAutoTrack eventExecutor)
- void addOverLatticeEvent (int eventId, double propensityPerUnitArea, EventExecutorSemiManualTrack eventExecutor, const std::vector< CellNeighOffsets > &cnoVec)
- void addStepPeriodicAction (int actionId, PeriodicAction action, int periodOfSteps, bool doAtSimEnd)
- void addTimePeriodicAction (int actionId, PeriodicAction action, double periodOfTime, bool doAtSimEnd)
- void changeCellCenteredEventGroup (int eventGroupId, const CellNeighOffsets &cno, CellCenteredGroupPropensities propensities, const EventExecutorGroup &eventExecutorGroup)
- void changeOverLatticeEvent (int eventId, double propensityPerUnitArea, EventExecutorAutoTrack eventExecutor)
- void changeOverLatticeEvent (int eventId, double propensityPerUnitArea, EventExecutorSemiManualTrack eventExecutor, const std::vector< CellNeighOffsets > &cnoVec)
- void changeStepPeriodicAction (int actionId, PeriodicAction action, int periodOfSteps, bool doAtSimEnd)
- void changeTimePeriodicAction (int actionId, PeriodicAction action, double periodOfTime, bool doAtSimEnd)
- const MPI_Comm & comm () const
- int commCoord (int dim) const
- double elapsedTime () const
- void getLatticeGlobalPlanarBBox (LatticePlanarBBox &bbox) const
- void getLatticeLocalPlanarBBox (bool wGhost, LatticePlanarBBox &bbox) const
- void getLatticeSectorPlanarBBox (int sectNum, LatticePlanarBBox &bbox) const
- int nProcs () const
- unsigned long long numGlobalSteps () const
- unsigned long long numLocalEvents () const
- int procID () const
- int procPerDim (int dim) const
- void removeCellCenteredEventGroup (int eventGroupId)

- void removeOverLatticeEvent (int eventId)
- void removeStepPeriodicAction (int actionId)
- void removeTimePeriodicAction (int actionId)
- void reserveCellCenteredEventGroups (int numGroups, int numTotEvents)
- void reserveOverLatticeEvents (int num)
- void reserveStepPeriodicActions (int num)
- void reserveTimePeriodicActions (int num)
- void run (double runTime)
- void setRNG (RandNumGenSharedPtr rng)
- void setSolver (SolverId::Type sId)
- void setTimeIncrScheme (const TimeIncr::SchemeVars &vars)
- void trackCellsChangedByPeriodicActions (bool doTrack)

6.17.1 Detailed Description

Class for setting up and running a simulation.

Typical usage for this class is something like this:

```
KMC_MAKE_LATTICE_INTVAL_ENUM(My,
                             GaSiteStatus,
                             AsSiteStatus);

LatticeParams latParams;

latParams.numIntsPerCell = MyIntVal::SIZE;
latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = myGlobalSize(...);

#if KMC_PARALLEL
latParams.ghostExtent[0] = latParams.ghostExtent[1] = myGhostExtent(...);
#endif

Simulation sim(latParams);

sim.setSolver(SolverId::BINARY_TREE);

RandNumGenSharedPtr rng(new MyRandNumGen(...));
sim.setRNG(rng);

TimeIncr::SchemeVars schemeVars;

// Set values of schemeVars

sim.setTimeIncrScheme(schemeVars);

KMC_MAKE_ID_ENUM(MyOverLatticeEvents,
                 DEPOSITION_Ga,
                 DEPOSITION_As2);

KMC_MAKE_ID_ENUM(MyCellCenteredEvents,
                 GaHOP,
                 As2Dimer_TO_AsCrystal, ... other events ... );


sim.reserveOverLatticeEvents(MyOverLatticeEvents::SIZE);

sim.addOverLatticeEvent(MyOverLatticeEvents::DEPOSITION_Ga,
                        myGaDepRate(...), GaDepositionExecute());

sim.addOverLatticeEvent(MyOverLatticeEvents::DEPOSITION_As2,
                        myAsDepRate(...), AsDepositionExecute());

sim.addCellCenteredEventGroup(MyCellCenteredEventGroups::HOPS, ...);

// More possible events defined

sim.run(approxDepTime);
sim.removeOverLatticeEvent(MyOverLatticeEvents::DEPOSITION_Ga);
sim.removeOverLatticeEvent(MyOverLatticeEvents::DEPOSITION_As2);
sim.run(2.0*approxDepTime);
```

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

6.17.2 Constructor & Destructor Documentation

Simulation()

```
KMCThinFilm::Simulation::Simulation (
            const LatticeParams & paramsForLattice)  [explicit]
```

Constructor to initialize the simulation

Parameters

paramsForLattice	Parameters for the lattice used internally by the simulation.

6.17.3 Member Function Documentation

addCellCenteredEventGroup()

```
void KMCThinFilm::Simulation::addCellCenteredEventGroup (
            int eventGroupId,
            const CellNeighOffsets & cno,
            CellCenteredGroupPropensities propensities,
            const EventExecutorGroup & eventExecutorGroup)
```

Adds a possible cell-centered event group to the simulation.

Parameters

eventGroupId	Unique integer ID of event group
cno	Offsets used to find cells used to determine the event propensities
propensities	Function or function object used to determine the propensities of this group of events.
eventExecutorGroup	Group of function objects, one of which runs if this event is executed.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

addOverLatticeEvent() [1/2]

```
void KMCThinFilm::Simulation::addOverLatticeEvent (
            int eventId,
            double propensityPerUnitArea,
            EventExecutorAutoTrack eventExecutor)
```

Adds a possible "over-lattice" event to the simulation, that is, one that may occur at a random site over the lattice (e.g. deposition of an atom). Here, the cells affected by the event are determined by "auto-tracking."

If this event occurs, the CellInds argument of *eventExecutor* will have its first two components, i & j, set to a random in-plane position within the current sector, and its third component, k, set to one less than the height of the lattice.

See also

EventExecutorAutoTrack

Parameters

eventId	Unique integer ID of event.
propensityPerUnitArea	Propensity per unit area (e.g. deposition flux) of event
eventExecutor	Function or function object that runs if this event is executed.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

addOverLatticeEvent() [2/2]

```
void KMCThinFilm::Simulation::addOverLatticeEvent (
              int eventId,
              double propensityPerUnitArea,
              EventExecutorSemiManualTrack eventExecutor,
              const std::vector< CellNeighOffsets > & cnoVec)
```

Adds a possible "over-lattice" event to the simulation, that is, one that may occur at a random site over the lattice (e.g. deposition of an atom). Here, the cells affected by the event are determined by "semi-manual" tracking.

If this event occurs, the CellInds argument of *eventExecutor* will have its first two components, i & j, set to a random in-plane position within the current sector, and its third component, k, set to one less than the height of the lattice.

See also

EventExecutorSemiManualTrack

Parameters

eventId	Unique integer ID of event.
propensityPerUnitArea	Propensity per unit area (e.g. deposition flux) of event
eventExecutor	Function or function object that runs if this event is executed.
cnoVec	Relative positions of cells directly changed by event.

addStepPeriodicAction()

```
void KMCThinFilm::Simulation::addStepPeriodicAction (
              int actionId,
              PeriodicAction action,
              int periodOfSteps,
              bool doAtSimEnd)
```

Adds a step-periodic action to the simulation.

Step-periodic actions are actions that occur every P time steps of the simulation, with P being the period.

Parameters

actionId	Unique integer ID for the action
action	Function or function object executed by this action.

Parameters

periodOfSteps	The period of the action
doAtSimEnd	If true, this action is done at the end of a simulation run as well.

addTimePeriodicAction()

```
void KMCThinFilm::Simulation::addTimePeriodicAction (
                int actionId,
                PeriodicAction action,
                double periodOfTime,
                bool doAtSimEnd)
```

Adds a time-periodic action to the simulation.

Time-periodic actions are actions that occur every P units of simulation time, with P being the time period.

Parameters

actionId	Unique integer ID for the action
action	Function or function object executed by this action.
periodOfTime	The period of the action
doAtSimEnd	If true, this action is done at the end of a simulation run as well.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

changeCellCenteredEventGroup()

```
void KMCThinFilm::Simulation::changeCellCenteredEventGroup (
                int eventGroupId,
                const CellNeighOffsets & cno,
                CellCenteredGroupPropensities propensities,
                const EventExecutorGroup & eventExecutorGroup)
```

Changes a possible cell-centered event group that has been previously added to the simulation.

See also

addCellCenteredEventGroup()

Parameters

eventGroupId	Unique integer ID of event group
cno	Offsets used to find cells used to determine the event propensities
propensities	Function or function object used to determine the propensities of this group of events.
eventExecutorGroup	Group of function objects, one of which runs if this event is executed.

changeOverLatticeEvent() [1/2]

```
void KMCThinFilm::Simulation::changeOverLatticeEvent (
              int eventId,
              double propensityPerUnitArea,
              EventExecutorAutoTrack eventExecutor)
```

Changes a possible "over-lattice" event that has been previously added to the simulation. Here, the cells affected by the event are determined by "auto-tracking."

See also

addOverLatticeEvent() EventExecutorAutoTrack

Parameters

eventId	Integer ID of event to be changes.
propensityPerUnitArea	Propensity per unit area (e.g. deposition flux) of event
eventExecutor	Function or function object that runs if this event is executed.

changeOverLatticeEvent() [2/2]

```
void KMCThinFilm::Simulation::changeOverLatticeEvent (
              int eventId,
              double propensityPerUnitArea,
              EventExecutorSemiManualTrack eventExecutor,
              const std::vector< CellNeighOffsets > & cnoVec)
```

Changes a possible "over-lattice" event that has been previously added to the simulation. Here, the cells affected by the event are determined by "semi-manual" tracking.

See also

addOverLatticeEvent() EventExecutorSemiManualTrack

Parameters

eventId	Integer ID of event to be changes.
propensityPerUnitArea	Propensity per unit area (e.g. deposition flux) of event
eventExecutor	Function or function object that runs if this event is executed.
cnoVec	Relative positions of cells directly changed by event.

changeStepPeriodicAction()

```
void KMCThinFilm::Simulation::changeStepPeriodicAction (
              int actionId,
              PeriodicAction action,
              int periodOfSteps,
              bool doAtSimEnd)
```

Changes a step-periodic action that had been previously added to the simulation.

See also

addStepPeriodicAction()

Parameters

actionId	Integer ID of the action to be changed
action	Function or function object executed by this action.
periodOfSteps	The period of the action
doAtSimEnd	If true, this action is done at the end of a simulation run as well.

changeTimePeriodicAction()

```
void KMCThinFilm::Simulation::changeTimePeriodicAction (
              int actionId,
              PeriodicAction action,
              double periodOfTime,
              bool doAtSimEnd)
```

Changes a time-periodic action that had been previously added to the simulation.

See also

addTimePeriodicAction()

Parameters

actionId	Integer ID of the action to be changed
action	Function or function object executed by this action.
periodOfTime	The period of the action
doAtSimEnd	If true, this action is done at the end of a simulation run as well.

comm()

```
const MPI_Comm & KMCThinFilm::Simulation::comm () const
```

The MPI communicator used by the lattice in the simulation for the interchange of ghost lattice cells, if the preprocessor variable KMC_PARALLEL is non-zero. **Not available in the serial version of the ARL KMCThinFilm library.**

This is *not* the same as the *latticeCommInitial* member of the LatticeParams object used to construct the lattice.

See also

Lattice::comm()

commCoord()

```
int KMCThinFilm::Simulation::commCoord (
              int dim) const
```

Coordinates for a processor in an MPI Cartesian topology along in-plane dimension *dim* for a given processor.

See also

Lattice::commCoord()

Examples

testFractal_parallel/testFractal.cpp.

elapsedTime()

```
double KMCThinFilm::Simulation::elapsedTime () const
```

The amount of *simulated* time (not the actual wall clock time) that has passed since the beginning of the first simulation run.

See also

SimulationState::elapsedTime()

getLatticeGlobalPlanarBBox()

```
void KMCThinFilm::Simulation::getLatticeGlobalPlanarBBox (
              LatticePlanarBBox & bbox) const
```

Obtain limiting values of the global lattice coordinates.

See also

Lattice::getGlobalPlanarBBox()

Parameters

bbox	Bounding box of the global lattice coordinates.

Examples

testBallisticDep2/testBallisticDep.cpp.

getLatticeLocalPlanarBBox()

```
void KMCThinFilm::Simulation::getLatticeLocalPlanarBBox (
              bool wGhost,
              LatticePlanarBBox & bbox) const
```

Obtain limiting values of the local coordinates of the cells used in the simulation lattice.

See also

Lattice::getLocalPlanarBBox()

Parameters

wGhost	Indicates whether minimum and maximum values for lattice cell coordinates include the coordinates of ghost lattice cells.
bbox	Bounding box of the local lattice coordinates.

getLatticeSectorPlanarBBox()

```
void KMCThinFilm::Simulation::getLatticeSectorPlanarBBox (
              int sectNum,
              LatticePlanarBBox & bbox) const
```

Obtain limiting values of the local lattice coordinates for a particular sector or sublattice.

See also

Lattice::getSectorPlanarBBox()

Parameters

sectNum	Sector number, ranging from 0 to numSectors() - 1
bbox	Bounding box of the lattice coordinates within the sector.

nProcs()

```
int KMCThinFilm::Simulation::nProcs () const
```

Number of processors over which the lattice in the simulation is distributed.

See also

Lattice::nProcs()

numGlobalSteps()

```
unsigned long long KMCThinFilm::Simulation::numGlobalSteps () const
```

Number of times the global clock has been incremented. For serial simulations, this yields the same value as numLocalEvents().

See also

SimulationState::numGlobalSteps()

numLocalEvents()

```
unsigned long long KMCThinFilm::Simulation::numLocalEvents () const
```

Number of events that have occurred on a particular processor.

See also

SimulationState::numLocalEvents()

procID()

```
int KMCThinFilm::Simulation::procID () const
```

When the preprocessor variable KMC_PARALLEL is zero, this always returns zero. Otherwise, this is the MPI rank of the local process, where the communicator for that process is given by Simulation::comm().

See also

Lattice::procID()

Examples

testFractal_parallel/testFractal.cpp.

procPerDim()

```
int KMCThinFilm::Simulation::procPerDim (
              int dim) const
```

The number of processors along in-plane dimension *dim*.

See also

Lattice::procPerDim()

removeCellCenteredEventGroup()

```
void KMCThinFilm::Simulation::removeCellCenteredEventGroup (
              int eventGroupId)
```

Removes a previously added possible cell-centered event group from the simulation.

See also

addCellCenteredEventGroup() changeCellCenteredEventGroup()

Parameters

eventGroupId	Integer ID of event group to be removed.

removeOverLatticeEvent()

```
void KMCThinFilm::Simulation::removeOverLatticeEvent (
              int eventId)
```

Removes a previously added possible "over-lattice" from the simulation.

See also

addOverLatticeEvent() changeOverLatticeEvent()

Parameters

eventId	Integer ID of event to be removed.

Examples

testFractal/testFractal.cpp, testFractal_parallel/testFractal.cpp, testFractal_semi_manual_track/testFractal.cpp, testPatternedSurfac
and testPatternedSurface2/testPatternedSurface.cpp.

removeStepPeriodicAction()

```
void KMCThinFilm::Simulation::removeStepPeriodicAction (
              int actionId)
```

Removes a previously-added step-periodic action from the simulation.

See also

addStepPeriodicAction() changeStepPeriodicAction()

removeTimePeriodicAction()

```
void KMCThinFilm::Simulation::removeTimePeriodicAction (
              int actionId)
```

Removes a previously-added time-periodic action from the simulation.

See also

addTimePeriodicAction() changeTimePeriodicAction()

Parameters

actionId	Integer ID of the action to be removed

reserveCellCenteredEventGroups()

```
void KMCThinFilm::Simulation::reserveCellCenteredEventGroups (
            int numGroups,
            int numTotEvents)
```

Sets the number of possible cell-centered event groups for the simulation to *numGroups*, and the number of individual possible cell-centered events to *numTotEvents*.

Should be called before a call to addCellCenteredEventGroup(), and the number of calls to addCellCenteredEventGroup() should be the same as *numGroups*.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

reserveOverLatticeEvents()

```
void KMCThinFilm::Simulation::reserveOverLatticeEvents (
            int num)
```

Sets the number of "over-lattice" possible events for the simulation, that is, events that occur at a random site over the lattice (e.g. deposition of an atom), to *num*.

Should be called before a call to addOverLatticeEvent(), and the number of calls to addOverLatticeEvent() should be the same as *num*.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

reserveStepPeriodicActions()

```
void KMCThinFilm::Simulation::reserveStepPeriodicActions (
            int num)
```

Sets the number of step-periodic actions for the simulation to *num*.

Should be called before a call to addStepPeriodicAction(), and the number of calls to addStepPeriodicAction() should be the same as *num*.

reserveTimePeriodicActions()

```
void KMCThinFilm::Simulation::reserveTimePeriodicActions (
            int num)
```

Sets the number of time-periodic actions for the simulation to *num*.

Should be called before a call to addTimePeriodicAction(), and the number of calls to addTimePeriodicAction() should be the same as *num*.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

run()

```
void KMCThinFilm::Simulation::run (
            double runTime)
```

Runs the simulation

Parameters

runTime	Length of simulation time

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

setRNG()

```
void KMCThinFilm::Simulation::setRNG (
              RandNumGenSharedPtr rng)
```

Sets the random-number generator of the simulation.

This must be called **after** setSolver() has been called.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

setSolver()

```
void KMCThinFilm::Simulation::setSolver (
              SolverId::Type sId)
```

Sets the type of solver, i.e. the means of storing and choosing events, for the simulation.

Examples

testBallisticDep1/testBallisticDep.cpp, testBallisticDep2/testBallisticDep.cpp, testFractal/testFractal.cpp, testFractal_parallel/testF
testFractal_semi_manual_track/testFractal.cpp, testPatternedSurface1/testPatternedSurface.cpp, and testPatternedSurface2/testPatt

setTimeIncrScheme()

```
void KMCThinFilm::Simulation::setTimeIncrScheme (
              const TimeIncr::SchemeVars & vars)
```

Sets the parallel time-incrementing scheme.

If KMC_PARALLEL equals zero, this does nothing. This must be called **after** setSolver() has been called.

Examples

testFractal_parallel/testFractal.cpp.

trackCellsChangedByPeriodicActions()

```
void KMCThinFilm::Simulation::trackCellsChangedByPeriodicActions (
              bool doTrack)
```

[**ADVANCED**] If *doTrack* is true, store indices of the lattice cells changed by the periodic actions that occur.

If a periodic action actually *changes* the lattice (i.e. calls Lattice::setInt(), Lattice::setFloat(), or Lattice::addPlanes()), then by default, the event list is rebuilt, which is an O(N) operation, with N being the number of lattice cells. This function changes this behavior. Since the indices of changed lattice cells are actually stored, only the entries in the event list affected by the changed lattice cells need to be changed. Note, though, that if the periodic actions change a large number of lattice sites, then the default behavior is probably more optimal, or at least less memory-hungry.

6.18 KMCThinFilm::SimulationState Class Reference

```
#include <SimulationState.hpp>
```

Public Member Functions

- double elapsedTime () const
- double globalTimeIncrement () const
- double maxTime () const
- unsigned long long numGlobalSteps () const
- unsigned long long numLocalEvents () const

Friends

- class **Simulation**

6.18.1 Detailed Description

Data structure holding the current state of the simulation.

This data structure provides an interface to function objects with the KMCThinFilm::EventExecutor and KMCThinFilm::PeriodicAction signatures for accessing such things as the current elapsed time of the simulation, the number of local events that have occurred, and so on.

See also

EventExecutor PeriodicAction

Examples

testBallisticDep1/EventsAndActions.cpp, testBallisticDep1/EventsAndActions.hpp, testBallisticDep2/EventsAndActions.cpp, testBallisticDep2/EventsAndActions.hpp, testFractal/EventsAndActions.cpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.cpp, testPatternedSurface1/EventsAndActions.hpp, testPatternedSurface2/EventsAndAc and testPatternedSurface2/EventsAndActions.hpp.

6.18.2 Member Function Documentation

elapsedTime()

```
double KMCThinFilm::SimulationState::elapsedTime () const  [inline]
```

The amount of *simulated* time (not the actual wall clock time) that has passed since the beginning of the first simulation run.

Examples

testFractal/EventsAndActions.cpp, testFractal_semi_manual_track/EventsAndActions.cpp, testPatternedSurface1/EventsAndActic and testPatternedSurface2/EventsAndActions.cpp.

globalTimeIncrement()

```
double KMCThinFilm::SimulationState::globalTimeIncrement () const  [inline]
```

The current value by which the global elasped time is incremented in a parallel simulation.

This returns zero in a serial simulation.

maxTime()

```
double KMCThinFilm::SimulationState::maxTime () const  [inline]
```

The maximum amount of simulation time (*not* wall-clock time) alloted for all the simulation runs.

numGlobalSteps()

```
unsigned long long KMCThinFilm::SimulationState::numGlobalSteps () const  [inline]
```

Number of times the global clock has been incremented.

In a serial simulation, this returns the same value as numLocalEvents().

numLocalEvents()

```
unsigned long long KMCThinFilm::SimulationState::numLocalEvents () const  [inline]
```

Number of events that have occurred on a particular processor.

Chapter 7

File Documentation

7.1 CellCenteredGroupPropensities.hpp File Reference

Defines the signature of the function object used to determine the propensities of a group of related events.

```
#include <vector>
#include <boost/function.hpp>
#include "CellNeighProbe.hpp"
```

Namespaces

- namespace KMCThinFilm

Typedefs

- typedef boost::function< void(const CellNeighProbe &, std::vector< double > &)> KMCThinFilm::CellCenteredGroupPropensit

7.1.1 Detailed Description

Defines the signature of the function object used to determine the propensities of a group of related events.

7.2 CellCenteredGroupPropensities.hpp

Go to the documentation of this file.

```
#ifndef PROPENSITY_GROUP_CALC_HPP
#define PROPENSITY_GROUP_CALC_HPP

#include <vector>
#include <boost/function.hpp>

#include "CellNeighProbe.hpp"


namespace KMCThinFilm {
  typedef boost::function<void (const CellNeighProbe &, std::vector<double> &)> CellCenteredGroupPropensities;
}

#endif /* PROPENSITY_GROUP_CALC_HPP */
```

7.3 CellInds.hpp File Reference

Defines the CellInds and CellInds offset classes, and their respective operators.

```
#include "IJK.hpp"
```

Classes

- class KMCThinFilm::CellInds
- class KMCThinFilm::CellIndsOffset

Namespaces

- namespace KMCThinFilm

Functions

- CellInds KMCThinFilm::operator+ (const CellInds &ci, const CellIndsOffset &offset)
- CellIndsOffset KMCThinFilm::operator+ (const CellIndsOffset &offset1, const CellIndsOffset &offset2)

7.3.1 Detailed Description

Defines the CellInds and CellInds offset classes, and their respective operators.

7.4 CellInds.hpp

Go to the documentation of this file.

```
#ifndef CELL_INDS_HPP
#define CELL_INDS_HPP

#include "IJK.hpp"

// Note: This header file is documented via Doxygen
// <http://www.doxygen.org>. Comments for Doxygen begin with '/*!' or
// '//!', and descriptions of functions, class and member functions
// occur *before* their corresponding class declarations and function
// prototypes.


namespace KMCThinFilm {


  class CellInds : public IJK {
  public:

    CellInds() {i = j = k = 0;}

    CellInds(int ii ,
             int jj ,
             int kk = 0 ) {
      i = ii; j = jj; k = kk;
    }

  };

  class CellIndsOffset : public IJK {
  public:

    CellIndsOffset() {i = j = k = 0;}

    CellIndsOffset(int ii ,
                   int jj ,
                   int kk = 0 ) {
      i = ii; j = jj; k = kk;
    }

    CellIndsOffset operator-() const;
  };

  CellInds operator+(const CellInds & ci, const CellIndsOffset & offset);

  CellIndsOffset operator+(const CellIndsOffset & offset1, const CellIndsOffset & offset2);

}

#endif /* CELL_INDS_HPP */
```

7.5 CellNeighOffsets.hpp File Reference

Defines the CellNeighOffsets class.

```
#include "CellInds.hpp"
#include <boost/scoped_ptr.hpp>
```

Classes

- class KMCThinFilm::CellNeighOffsets

Namespaces

- namespace KMCThinFilm

7.5.1 Detailed Description

Defines the CellNeighOffsets class.

7.6 CellNeighOffsets.hpp

Go to the documentation of this file.

```
#ifndef CELL_NEIGH_OFFSETS_HPP
#define CELL_NEIGH_OFFSETS_HPP

#include "CellInds.hpp"
#include <boost/scoped_ptr.hpp>


namespace KMCThinFilm {

  class CellNeighOffsets {
  public:

    explicit CellNeighOffsets(int numberOfOffsets );

    CellNeighOffsets(const CellNeighOffsets & cno);
    CellNeighOffsets & operator=(const CellNeighOffsets & rhs);

    void addOffset(int whichOffset ,
                   const CellIndsOffset & offset );

    void resetOffsets(int numberOfNewOffsets);

    void clearOffsets();

    const CellIndsOffset & getOffset(int whichOffset  ) const;

    int numOffsets() const;

    ~CellNeighOffsets();

  private:
    class Impl_;
    boost::scoped_ptr<Impl_> pImpl_;
  };

}

#endif /* CELL_NEIGH_OFFSETS_HPP */
```

7.7 CellNeighProbe.hpp File Reference

Defines the CellNeighProbe and CellToProbe classes.

```
#include <vector>
#include <boost/scoped_ptr.hpp>
#include "CellInds.hpp"
```

Classes

- class KMCThinFilm::CellNeighProbe
- class KMCThinFilm::CellToProbe

Namespaces

- namespace KMCThinFilm

7.7.1 Detailed Description

Defines the CellNeighProbe and CellToProbe classes.

7.8 CellNeighProbe.hpp

Go to the documentation of this file.

```
#ifndef CELL_NEIGH_PROBE_HPP
#define CELL_NEIGH_PROBE_HPP

#include <vector>
#include <boost/scoped_ptr.hpp>

#include "CellInds.hpp"


namespace KMCThinFilm {

  class Lattice;

  class CellToProbe {
    friend class CellNeighProbe;
  public:

    CellToProbe()
      : ci_((CellInds())) /* Extra parens are to avoid C++'s "most
                             vexing parse", but I'm not sure if
                             they're needed. */
    {}

    const CellInds & inds() {return ci_;}
  private:
    explicit CellToProbe(const CellInds & ci)
      : ci_(ci)
    {}

    CellInds ci_;
  };

  class CellNeighProbe {
  public:

    explicit CellNeighProbe(const Lattice * lattice = NULL);

    CellNeighProbe(const CellNeighProbe & ctp);
    CellNeighProbe & operator=(const CellNeighProbe & rhs);

    void attachLattice(const Lattice * lattice);

    void attachCellInds(const CellInds * ci ,
                         const std::vector<CellIndsOffset> * cioVecPtr );

    CellToProbe getCellToProbe(int probedCellInd ) const;

    double getFloat(const CellToProbe & ctp, int whichFloat) const;

    int getInt(const CellToProbe & ctp, int whichInt) const;

    bool exceedsLatticeHeight(const CellToProbe & ctp) const;

    bool belowLatticeBottom(const CellToProbe & ctp) const;

    ~CellNeighProbe();

  private:
    class Impl_;
    boost::scoped_ptr<Impl_> pImpl_;
  };

}

#endif /* CELL_NEIGH_PROBE_HPP */
```

7.9 CellsToChange.hpp

```
#ifndef CELLS_TO_CHANGE_HPP
#define CELLS_TO_CHANGE_HPP

#include <boost/scoped_ptr.hpp>
#include <vector>

namespace KMCThinFilm {

  class Lattice;
  class CellInds;
  class CellIndsOffset;

  class CellsToChange {
    friend class Simulation;
  public:

    void setCenter(const CellInds & ci);

    const CellInds & getCellInds(int whichOffset) const;

    int getInt(int whichOffset, int whichInt) const;

    double getFloat(int whichOffset, int whichFloat) const;

    void setInt(int whichOffset, int whichInt, int val);

    void setFloat(int whichOffset, int whichFloat, double val);

    void addLatticePlanes(int numPlanesToAdd);

    CellsToChange(const CellsToChange & ctc);
    CellsToChange & operator=(const CellsToChange & rhs);
    ~CellsToChange();

  private:

    // For use by Simulation class
    explicit CellsToChange(Lattice * lattice, int numOffsets);
    void addOffset(const CellIndsOffset & offset);
    const CellInds & getCenter() const;
    const std::vector<CellIndsOffset> & getCellIndsOffsetVec() const;
    const std::vector<CellInds> & getCellIndsVec() const;
    void clear();

    class Impl_;
    boost::scoped_ptr<Impl_> pImpl_;
  };

}

#endif /* CELLS_TO_CHANGE_HPP */
```

7.10 ErrorHandling.hpp File Reference

Defines functions that exit the KMC application in case of an error.

```
#include <string>
```

Namespaces

- namespace KMCThinFilm

Functions

- void KMCThinFilm::abortOnCondition (bool condition, const std::string &msg)
- void KMCThinFilm::abortWithMsg (const std::string &msg)
- void KMCThinFilm::exitOnCondition (bool condition, const std::string &msg)
- void KMCThinFilm::exitWithMsg (const std::string &msg)

7.10.1 Detailed Description

Defines functions that exit the KMC application in case of an error.

These are free functions that don't belong to any particular class.

7.11 ErrorHandling.hpp

Go to the documentation of this file.

```
#ifndef ERROR_HANDLING_FUNCS_HPP
#define ERROR_HANDLING_FUNCS_HPP

#include <string>

// Note: This header file is documented via Doxygen
// <http://www.doxygen.org>. Comments for Doxygen begin with '/*!' or
// '//!', and descriptions of functions, class and member functions
// occur *before* their corresponding class declarations and function
// prototypes.


namespace KMCThinFilm {

  void exitWithMsg(const std::string & msg);

  void abortWithMsg(const std::string & msg);

  void exitOnCondition(bool condition, const std::string & msg);

  void abortOnCondition(bool condition, const std::string & msg);

}

#endif /* ERROR_HANDLING_FUNCS_HPP */
```

7.12 EventExecutor.hpp File Reference

Defines the possible signatures for a function object called when an event executes.

```
#include <vector>
#include <boost/function.hpp>
#include "CellInds.hpp"
#include "CellsToChange.hpp"
#include "Lattice.hpp"
```

```
#include "SimulationState.hpp"
```

Namespaces

- namespace KMCThinFilm

Typedefs

- typedef boost::function< void(const CellInds &, const SimulationState &, Lattice &)> KMCThinFilm::EventExecutorAutoTrack
- typedef boost::function< void(const CellInds &, const SimulationState &, const Lattice &, std::vector< CellsToChange > &)> KMCThinFilm::EventExecutorSemiManualTrack

7.12.1 Detailed Description

Defines the possible signatures for a function object called when an event executes.

7.13 EventExecutor.hpp

Go to the documentation of this file.

```
#ifndef EVENT_EXECUTOR_HPP
#define EVENT_EXECUTOR_HPP

#include <vector>

#include <boost/function.hpp>

#include "CellInds.hpp"
#include "CellsToChange.hpp"
#include "Lattice.hpp"
#include "SimulationState.hpp"


namespace KMCThinFilm {

  typedef boost::function<void (const CellInds &, const SimulationState &, Lattice &)> EventExecutorAutoTrack;

  typedef boost::function<void (const CellInds &,
                                const SimulationState &,
                                const Lattice &,
                                std::vector<CellsToChange> &)> EventExecutorSemiManualTrack;
}

#endif /* EVENT_EXECUTOR_HPP */
```

7.14 EventExecutorGroup.hpp File Reference

Defines the class EventExecutorGroup.

```
#include <vector>
#include <boost/scoped_ptr.hpp>
#include "CellInds.hpp"
#include "CellNeighOffsets.hpp"
#include "EventExecutor.hpp"
```

Classes

- class KMCThinFilm::EventExecutorGroup

Namespaces

- namespace KMCThinFilm

7.14.1 Detailed Description

Defines the class EventExecutorGroup.

7.15 EventExecutorGroup.hpp

Go to the documentation of this file.

```
#ifndef EVENT_EXECUTOR_GROUP_HPP
#define EVENT_EXECUTOR_GROUP_HPP

#include <vector>

#include <boost/scoped_ptr.hpp>

#include "CellInds.hpp"
#include "CellNeighOffsets.hpp"
#include "EventExecutor.hpp"


namespace KMCThinFilm {

  class EventExecutorGroup {
    friend class Simulation;
  public:

    explicit EventExecutorGroup(int numEventsInGroup );

    EventExecutorGroup(const EventExecutorGroup & eeg);
    EventExecutorGroup & operator=(const EventExecutorGroup & rhs);

    void addEventExecutor(int whichEvent ,
                          EventExecutorAutoTrack evExec );

    void addEventExecutor(int whichEvent ,
                          EventExecutorSemiManualTrack evExec ,
                          const std::vector<CellNeighOffsets> & cnoVec  );

    void resetGroup(int numEventsInGroup);

    void clearGroup();

    int numEventExecutors() const;

    ~EventExecutorGroup();

  private:

    struct EventExecEnum {
      enum Type {AUTO, SEMIMANUAL};
    };

    // To be used by Simulation class
    EventExecEnum::Type getEventExecutorType(int whichEvent) const;

    EventExecutorAutoTrack getEventExecutorAutoTrack(int whichEvent) const;
    EventExecutorSemiManualTrack getEventExecutorSemiManualTrack(int whichEvent) const;
    const std::vector<CellNeighOffsets> & getEventExecutorOffsetsVec(int whichEvent) const;

    class Impl_;
    boost::scoped_ptr<Impl_> pImpl_;
  };

}

#endif /* EVENT_EXECUTOR_GROUP_HPP */
```

7.16 IdsOfSolvers.hpp File Reference

Defines the enumeration SolverId::Type.

Namespaces

- namespace KMCThinFilm
- namespace KMCThinFilm::SolverId

Enumerations

- enum KMCThinFilm::SolverId::Type { KMCThinFilm::SolverId::DYNAMIC_SCHULZE , KMCThinFilm::SolverId::BINARY_
  }

7.16.1 Detailed Description

Defines the enumeration SolverId::Type.

7.17 IdsOfSolvers.hpp

Go to the documentation of this file.

```
#ifndef IDS_OF_SOLVERS_HPP
#define IDS_OF_SOLVERS_HPP

// This file is named IdsOfSolvers.hpp rather than SolverIds.hpp
// because the Doxygen documentation excludes files with the pattern
// "Solver*.hpp".


namespace KMCThinFilm {

  namespace SolverId {

    enum Type {
      DYNAMIC_SCHULZE ,

      BINARY_TREE
    };

  }

}

#endif /* IDS_OF_SOLVERS_HPP */
```

7.18 Lattice.hpp File Reference

Defines the Lattice class and related parameter objects and typedefs.

```
#include "KMC_Config.hpp"
#include <vector>
#include <deque>
#include <set>
#include <mpi.h>
#include <boost/array.hpp>
#include <boost/function.hpp>
#include <boost/noncopyable.hpp>
#include <boost/scoped_ptr.hpp>
#include "CellInds.hpp"
```

Classes

- class KMCThinFilm::AddEmptyPlanes
- class KMCThinFilm::Lattice
- struct KMCThinFilm::LatticeParams
- struct KMCThinFilm::LatticePlanarBBox

Namespaces

- namespace KMCThinFilm

Typedefs

- typedef boost::function< void(Lattice &lattice)> KMCThinFilm::LatticeInitializer
- typedef boost::function< void(const CellInds &, const Lattice &, std::vector< int > &emptyIntVals, std::↩ ::vector< double > &emptyFloatVals)> KMCThinFilm::SetEmptyCellVals

7.18.1 Detailed Description

Defines the Lattice class and related parameter objects and typedefs.

7.19 Lattice.hpp

Go to the documentation of this file.

```
#ifndef LATTICE_HPP
#define LATTICE_HPP

#include "KMC_Config.hpp"

#include <vector>
#include <deque>
#include <set>

#if KMC_PARALLEL
#include <mpi.h>
#endif

#include <boost/array.hpp>
#include <boost/function.hpp>
#include <boost/noncopyable.hpp>
#include <boost/scoped_ptr.hpp>

#include "CellInds.hpp"

// Note: This header file is documented via Doxygen
// <http://www.doxygen.org>. Comments for Doxygen begin with '/*!' or
// '//!', and descriptions of functions, class and member functions
// occur *before* their corresponding class declarations and function
// prototypes.


namespace KMCThinFilm {

  class Lattice; // Need to forward declare this for the sake of the
                 // following typedefs.

  typedef boost::function<void (const CellInds &,
                                const Lattice &,
                                std::vector<int> & emptyIntVals,
                                std::vector<double> & emptyFloatVals)> SetEmptyCellVals;

  struct LatticePlanarBBox {
    int imin ,
      imaxP1 ,
      jmin ,
      jmaxP1 ;
  };

  typedef boost::function<void (Lattice & lattice)> LatticeInitializer;

  class AddEmptyPlanes {
  public:
    AddEmptyPlanes(int numPlanesToAdd);

    void operator()(Lattice & lattice) const;
  private:
    int numPlanesToAdd_;
  };

  struct LatticeParams {

    enum ParallelDecomp {
      COMPACT ,
      ROW
    };

    LatticeParams()
      : numIntsPerCell(0),
        numFloatsPerCell(0),
        numPlanesToReserve(1),
        parallelDecomp(ROW),
        noAddingPlanesDuringSimulation(false)
#if KMC_PARALLEL
      , latticeCommInitial(MPI_COMM_WORLD)
#endif
    {
      globalPlanarDims[0] = globalPlanarDims[1] = ghostExtent[0] = ghostExtent[1] = 0;
      latInit = AddEmptyPlanes(1);
    }

    boost::array<int,2> globalPlanarDims ,

      ghostExtent ;

    int numIntsPerCell ,
      numFloatsPerCell ;

    int numPlanesToReserve ;

    ParallelDecomp parallelDecomp ;

    LatticeInitializer latInit ;

    bool noAddingPlanesDuringSimulation ;

#if KMC_PARALLEL
    MPI_Comm latticeCommInitial ;
#endif

    SetEmptyCellVals setEmptyCellVals ;
  };

  class Lattice : private boost::noncopyable {
    friend class Simulation;
  public:

    Lattice(const LatticeParams & paramsForLattice);

    int nProcs() const;

    int procID() const;


#if KMC_PARALLEL
    const MPI_Comm & comm() const;
#endif

    int numSectors() const;

    int procPerDim(int dim) const;

    int commCoord(int dim) const;

    int ghostExtent(int dim) const;

    void addPlanes(int numPlanesToAdd);

    void reservePlanes(int numTotalPlanesToReserve);

    int planesReserved() const;

    void recvGhosts(int sectNum );

    void sendGhosts(int sectNum );

    void getLocalPlanarBBox(bool wGhost ,
                            LatticePlanarBBox & bbox ) const;

    void getSectorPlanarBBox(int sectNum ,
                             LatticePlanarBBox & bbox ) const;

    void getGlobalPlanarBBox(LatticePlanarBBox & bbox ) const;

    int currHeight() const;

    int nIntsPerCell() const;

    int nFloatsPerCell() const;

    int getInt(const CellInds & ci, int whichInt) const;

    double getFloat(const CellInds & ci, int whichFloat) const;

    void setInt(const CellInds & ci, int whichInt, int val);

    void setFloat(const CellInds & ci, int whichInt, double val);

    int wrapI(const CellInds & ci) const;

    int wrapJ(const CellInds & ci) const;

    int sectorOfIndices(const CellInds & ci) const;

    void recvGhostsUpdate(int sectNum );

    void sendGhostsUpdate(int sectNum );

    const std::vector<std::vector<IJK> > & getReceivedGhostInds() const;

    const std::vector<std::vector<IJK> > & getReceivedLocalInds() const;

    void clearGhostsToSend();

    bool addToExportBufferIfNeeded(const CellInds & ci);

    ~Lattice(); // Destructor should be public so that "smart"
                // pointers to the Lattice class can delete it.

#if KMC_PARALLEL
    bool addToExportBufferIfNeeded(const CellInds & ci, int & sectNum);

    // This is only needed for debugging.
    int exportVal(const CellInds & ci) const;
#endif

  private:

    // Used in Simulation class

    struct TrackType {
      enum Type {
        NONE,
        CHECK_ONLY_IF_CHANGE_OCCURS,
        RECORD_CHANGED_CELL_INDS,
        RECORD_ONLY_OTHER_CHANGED_CELL_INDS,
      };
    };

    // Using a std::set turns out to be faster than a
    // boost::unordered_set for the purposes of this library (probably
    // because there are usually so few elements in ChangedCellInds).
    typedef std::set<CellInds> ChangedCellInds;

    typedef std::deque<CellInds> OtherCheckedCellInds;

    void trackChanges(TrackType::Type trackType);
    bool hasChanged() const;

    const ChangedCellInds & getChangedCellInds() const;
    const OtherCheckedCellInds & getOtherCheckedCellInds() const;
    void wrapIndsIfNeeded(CellInds & ci) const;

    class Impl_;
    boost::scoped_ptr<Impl_> pImpl_;
  };

}

#endif /* LATTICE_HPP */
```

7.20 MakeEnum.hpp File Reference

Defines convenience macros to set up enumeration constants for a KMC simulation.

Macros

- #define KMC_MAKE_ID_ENUM(EnumName, ...)
- #define KMC_MAKE_LATTICE_FLOATVAL_ENUM(EnumName, ...)
- #define KMC_MAKE_LATTICE_INTVAL_ENUM(EnumName, ...)
- #define KMC_MAKE_OFFSET_ENUM(EnumName, ...)

7.20.1 Detailed Description

Defines convenience macros to set up enumeration constants for a KMC simulation.

7.20.2 Macro Definition Documentation

KMC_MAKE_ID_ENUM

```
#define KMC_MAKE_ID_ENUM(
                EnumName,
                ...)
```

Value:

```
  struct EnumName {                                  \
    enum Type {__VA_ARGS__, SIZE};                   \
  }
```

Convenience macro for defining enumeration constants that identify possible events.

For example, the following

```
KMC_MAKE_ID_ENUM(MyCellCenteredEvents,
                 HOP_UP,
             HOP_DOWN,
             HOP_LEFT,
             HOP_RIGHT);
```

creates an enumeration with values MyCellCenteredEvents::HOP_UP, MyCellCenteredEvents::HOP_DOWN, My↩
CellCenteredEvents::HOP_LEFT, and MyCellCenteredEvents::HOP_RIGHT and also defines MyCellCenteredEvents↩
::SIZE, which indicates the number of enumeration constants defined.

Note that this macro only works with preprocessors that do C99-style variadic macros.

See also

Simulation::addCellCenteredEvent()

Examples

testBallisticDep1/EventsAndActions.hpp, testBallisticDep2/EventsAndActions.hpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.hpp, and testPatternedSurface2/

KMC_MAKE_LATTICE_FLOATVAL_ENUM

```
#define KMC_MAKE_LATTICE_FLOATVAL_ENUM(
                EnumName,
                ...)
```

Value:

```
  struct EnumName##FloatVal {                   \
    enum Type {__VA_ARGS__, SIZE};                      \
  }
```

Convenience macro for defining enumeration constants for the floating-point parameters defined in each lattice cell.

For example, the following

```
KMC_MAKE_LATTICE_FLOATVAL_ENUM(SiCoords, SiX, SiY, SiZ);
```

creates an enumeration with values SiCoordsFloatVal::SiX, SiCoordsFloatVal::SiY, and SiCoordsFloatVal::SiZ and also defines SiCoordsFloatVal::SIZE, which indicates the number of enumeration constants defined.

Note that this macro only works with preprocessors that do C99-style variadic macros.

Examples

testBallisticDep1/EventsAndActions.hpp, testBallisticDep2/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.hpp and testPatternedSurface2/EventsAndActions.hpp.

KMC_MAKE_LATTICE_INTVAL_ENUM

```
#define KMC_MAKE_LATTICE_INTVAL_ENUM(
                EnumName,
                ...)
```

Value:

```
  struct EnumName##IntVal {                             \
    enum Type {__VA_ARGS__, SIZE};                      \
  }
```

Convenience macro for defining enumeration constants for the integers defined in each lattice cell.

For example, the following

```
KMC_MAKE_LATTICE_INTVAL_ENUM(GaInN_, Ga, In, N);
```

creates an enumeration with values GaInN_IntVal::Ga, GaInN_IntVal::In, and GaInN_IntVal::N, and also defines Ga↩ InN_IntVal::SIZE, which indicates the number of enumeration constants defined.

Note that this macro only works with preprocessors that do C99-style variadic macros.

Examples

testBallisticDep1/EventsAndActions.hpp, testBallisticDep2/EventsAndActions.hpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.hpp, and testPatternedSurface2/

KMC_MAKE_OFFSET_ENUM

```
#define KMC_MAKE_OFFSET_ENUM(
                EnumName,
                ...)
```

Value:

```
  struct EnumName {                             \
    enum Type {SELF, __VA_ARGS__, SIZE};        \
  }
```

Convenience macro for defining enumeration constants for the offsets used in a cell propensity calculation.

For example, the following

```
KMC_MAKE_OFFSET_ENUM(HopOffset,
                     UP, DOWN, LEFT, RIGHT);
```

creates an enumeration that not only contains the values HopOffset::UP, HopOffset::DOWN, HopOffset::LEFT, and HopOffset::RIGHT, but also the values HopOffset::SELF – *which always has an integer value of zero* – and Hop↩ Offset::SIZE, which indicates the number of enumeration constants defined.

Note that this macro only works with preprocessors that do C99-style variadic macros.

See also

CellNeighOffsets CellNeighProbe

Examples

testBallisticDep1/EventsAndActions.hpp, testBallisticDep2/EventsAndActions.hpp, testFractal/EventsAndActions.hpp, testFractal_semi_manual_track/EventsAndActions.hpp, testPatternedSurface1/EventsAndActions.hpp, and testPatternedSurface2/

7.21 MakeEnum.hpp

Go to the documentation of this file.

```
#ifndef MAKE_ENUM_HPP
#define MAKE_ENUM_HPP


#define KMC_MAKE_LATTICE_INTVAL_ENUM(EnumName, ...)     \
  struct EnumName##IntVal {                             \
    enum Type {__VA_ARGS__, SIZE};                      \
  }

#define KMC_MAKE_LATTICE_FLOATVAL_ENUM(EnumName, ...)   \
  struct EnumName##FloatVal {                   \
    enum Type {__VA_ARGS__, SIZE};                      \
  }

#define KMC_MAKE_OFFSET_ENUM(EnumName, ...)     \
  struct EnumName {                             \
    enum Type {SELF, __VA_ARGS__, SIZE};        \
  }


#define KMC_MAKE_ID_ENUM(EnumName, ...) \
  struct EnumName {                                     \
    enum Type {__VA_ARGS__, SIZE};                      \
  }

#endif /* MAKE_ENUM_HPP */
```

7.22 PeriodicAction.hpp File Reference

Defines the signature for the function object called periodically during a simulation.

```
#include <boost/function.hpp>
#include "SimulationState.hpp"
#include "Lattice.hpp"
```

Namespaces

- namespace KMCThinFilm

Typedefs

- typedef boost::function< void(const SimulationState &, Lattice &)> KMCThinFilm::PeriodicAction

7.22.1 Detailed Description

Defines the signature for the function object called periodically during a simulation.

7.23 PeriodicAction.hpp

Go to the documentation of this file.

```
#ifndef POST_GLOBAL_TIME_STEP_ACTION
#define POST_GLOBAL_TIME_STEP_ACTION

#include <boost/function.hpp>

#include "SimulationState.hpp"
#include "Lattice.hpp"


namespace KMCThinFilm {
  typedef boost::function<void (const SimulationState &, Lattice &)> PeriodicAction;
}

#endif /* POST_GLOBAL_TIME_STEP_ACTION */
```

7.24 RandNumGen.hpp File Reference

Defines abstract class RandNumGen.

`#include <boost/shared_ptr.hpp>`

Classes

- class KMCThinFilm::RandNumGen

Namespaces

- namespace KMCThinFilm

Typedefs

- typedef boost::shared_ptr< RandNumGen > KMCThinFilm::RandNumGenSharedPtr

7.24.1 Detailed Description

Defines abstract class RandNumGen.

7.25 RandNumGen.hpp

Go to the documentation of this file.

```
#ifndef RAND_NUM_GEN_HPP
#define RAND_NUM_GEN_HPP

#include <boost/shared_ptr.hpp>

// Note: This header file is documented via Doxygen
// <http://www.doxygen.org>. Comments for Doxygen begin with '/*!' or
// '//!', and descriptions of functions, class and member functions
// occur *before* their corresponding class declarations and function
// prototypes.


namespace KMCThinFilm {

  class RandNumGen {
```

```
  public:
    virtual double getNumInOpenIntervalFrom0To1() = 0;

    virtual ~RandNumGen() {}

  };

  typedef boost::shared_ptr<RandNumGen> RandNumGenSharedPtr;

}

#endif /* RAND_NUM_GEN_HPP */
```

7.26 RandNumGenDCMT.hpp File Reference

Defines the RandNumGenDCMT class.

```
#include <dc.h>
#include "RandNumGen.hpp"
```

Classes

- class KMCThinFilm::RandNumGenDCMT

Namespaces

- namespace KMCThinFilm

7.26.1 Detailed Description

Defines the RandNumGenDCMT class.

7.27 RandNumGenDCMT.hpp

Go to the documentation of this file.

```
#ifndef RAND_NUM_GEN_DCMT_HPP
#define RAND_NUM_GEN_DCMT_HPP

extern "C" {
#include <dc.h>
}

#include "RandNumGen.hpp"


namespace KMCThinFilm {

  class RandNumGenDCMT : public RandNumGen {
  public:

    enum Period {
      P521 = 521,
      P607 = 607,
      P1279 = 1279,
      P2203 = 2203,
      P2281 = 2281,
      P3217 = 3217,
      P4253 = 4253,
      P4423 = 4423,
      P9689 = 9689,
      P9941 = 9941,
      P11213 = 11213,
      P19937 = 19937,
      P21701 = 21701,
      P23209 = 23209,
      P44497 = 44497
    };
    RandNumGenDCMT(int rank ,
                   uint32_t seed ,
                   uint32_t seed_perProc ,
                   Period period = P521 );

    virtual double getNumInOpenIntervalFrom0To1();

    virtual ~RandNumGenDCMT();
  private:
    uint32_t seed_;
    mt_struct * mts_;
  };

}

#endif /* RAND_NUM_GEN_DCMT_HPP */
```

7.28 RandNumGenMT19937.hpp File Reference

Defines the RandNumGenMT19937 class.

```
#include "RandNumGen.hpp"
#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_01.hpp>
```

Classes

- class KMCThinFilm::RandNumGenMT19937

Namespaces

- namespace KMCThinFilm

7.28.1 Detailed Description

Defines the RandNumGenMT19937 class.

7.29 RandNumGenMT19937.hpp

Go to the documentation of this file.

```
#ifndef RAND_NUM_GEN_MT19937_HPP
#define RAND_NUM_GEN_MT19937_HPP


#include "RandNumGen.hpp"

#include <boost/random/mersenne_twister.hpp>
#include <boost/random/uniform_01.hpp>

namespace KMCThinFilm {

  class RandNumGenMT19937 : public RandNumGen {
  public:
    RandNumGenMT19937(unsigned int seed)
      : rng_(seed)
    {}

    virtual double getNumInOpenIntervalFrom0To1();

  private:
    boost::random::mt19937 rng_;
    boost::random::uniform_01<double> uni01_;
  };

}
```

```
#endif /* RAND_NUM_GEN_MT19937_HPP */
```

7.30 RandNumGenRngStreams.hpp File Reference

Defines the RandNumGenRngStreams class.

```
#include <vector>
#include <string>
#include <RngStream.h>
#include "RandNumGen.hpp"
```

Classes

- class KMCThinFilm::RandNumGenRngStreams

Namespaces

- namespace KMCThinFilm

7.30.1 Detailed Description

Defines the RandNumGenRngStreams class.

7.31 RandNumGenRngStreams.hpp

Go to the documentation of this file.

```
#ifndef RAND_NUM_GEN_RNG_STREAMS_HPP
#define RAND_NUM_GEN_RNG_STREAMS_HPP

#include <vector>
#include <string>

extern "C" {
#include <RngStream.h>
}


#include "RandNumGen.hpp"

namespace KMCThinFilm {

  class RandNumGenRngStreams : public RandNumGen {
  public:

    RandNumGenRngStreams(std::vector<unsigned long> & seed,
                         int rank,
                         long advExp = 127,
                         long advConst = 0);

    virtual double getNumInOpenIntervalFrom0To1();

    virtual ~RandNumGenRngStreams();
  private:
    RngStream rng_;
  };

}

#endif /* RAND_NUM_GEN_RNG_STREAMS_HPP */
```

7.32 Simulation.hpp File Reference

Defines the Simulation class.

```
#include "KMC_Config.hpp"
#include <mpi.h>
#include <boost/noncopyable.hpp>
#include <boost/scoped_ptr.hpp>
#include "CellCenteredGroupPropensities.hpp"
#include "EventExecutorGroup.hpp"
#include "PeriodicAction.hpp"
#include "CellNeighOffsets.hpp"
#include "RandNumGen.hpp"
#include "TimeIncrSchemeVars.hpp"
#include "IdsOfSolvers.hpp"
```

Classes

- class KMCThinFilm::Simulation

Namespaces

- namespace KMCThinFilm

7.32.1 Detailed Description

Defines the Simulation class.

7.33 Simulation.hpp

Go to the documentation of this file.

```
#ifndef SIMULATION_HPP
#define SIMULATION_HPP

#include "KMC_Config.hpp"

#if KMC_PARALLEL
#include <mpi.h>
#endif

#include <boost/noncopyable.hpp>
#include <boost/scoped_ptr.hpp>

#include "CellCenteredGroupPropensities.hpp"
#include "EventExecutorGroup.hpp"
#include "PeriodicAction.hpp"
#include "CellNeighOffsets.hpp"
#include "RandNumGen.hpp"
#include "TimeIncrSchemeVars.hpp"
#include "IdsOfSolvers.hpp"


namespace KMCThinFilm {

  class Lattice;

  class Simulation : private boost::noncopyable {
  public:

    explicit Simulation(const LatticeParams & paramsForLattice );

    ~Simulation();

    int nProcs() const;

    int procID() const;


#if KMC_PARALLEL
    const MPI_Comm & comm() const;
#endif

    int procPerDim(int dim) const;

    int commCoord(int dim) const;

    void getLatticeLocalPlanarBBox(bool wGhost ,
                                   LatticePlanarBBox & bbox ) const;

    void getLatticeSectorPlanarBBox(int sectNum ,
                                    LatticePlanarBBox & bbox ) const;

    void getLatticeGlobalPlanarBBox(LatticePlanarBBox & bbox ) const;

    double elapsedTime() const;

    unsigned long long numLocalEvents() const;

    unsigned long long numGlobalSteps() const;

    void reserveCellCenteredEventGroups(int numGroups, int numTotEvents);

    void addCellCenteredEventGroup(int eventGroupId ,
                                   const CellNeighOffsets & cno ,
                                   CellCenteredGroupPropensities propensities ,
                                   const EventExecutorGroup & eventExecutorGroup );

    void changeCellCenteredEventGroup(int eventGroupId ,
                                      const CellNeighOffsets & cno ,
                                      CellCenteredGroupPropensities propensities ,
                                      const EventExecutorGroup & eventExecutorGroup );

    void removeCellCenteredEventGroup(int eventGroupId );

    void reserveOverLatticeEvents(int num);

    void addOverLatticeEvent(int eventId ,
                             double propensityPerUnitArea ,
                             EventExecutorAutoTrack eventExecutor );

    void addOverLatticeEvent(int eventId ,
                             double propensityPerUnitArea ,
                             EventExecutorSemiManualTrack eventExecutor ,
                             const std::vector<CellNeighOffsets> & cnoVec );

    void changeOverLatticeEvent(int eventId ,
                                double propensityPerUnitArea ,
                                EventExecutorAutoTrack eventExecutor );

    void changeOverLatticeEvent(int eventId ,
                                double propensityPerUnitArea ,
                                EventExecutorSemiManualTrack eventExecutor ,
                                const std::vector<CellNeighOffsets> & cnoVec );

    void removeOverLatticeEvent(int eventId );

    void trackCellsChangedByPeriodicActions(bool doTrack);

    void reserveTimePeriodicActions(int num);

    void addTimePeriodicAction(int actionId ,
                               PeriodicAction action ,
                               double periodOfTime ,
                               bool doAtSimEnd );

    void changeTimePeriodicAction(int actionId ,
                                  PeriodicAction action ,
                                  double periodOfTime ,
                                  bool doAtSimEnd );

    void removeTimePeriodicAction(int actionId );

    void reserveStepPeriodicActions(int num);

    void addStepPeriodicAction(int actionId ,
                                     PeriodicAction action ,
                                     int periodOfSteps ,
                                     bool doAtSimEnd );

    void changeStepPeriodicAction(int actionId ,
                                        PeriodicAction action ,
                                        int periodOfSteps ,
                                        bool doAtSimEnd );

    void removeStepPeriodicAction(int actionId);

    void setSolver(SolverId::Type sId);

    void setTimeIncrScheme(const TimeIncr::SchemeVars & vars);

    void setRNG(RandNumGenSharedPtr rng);

    void run(double runTime );
  private:
    class Impl_;
    boost::scoped_ptr<Impl_> pImpl_;
  };

}

#endif /* SIMULATION_HPP */
```

7.34 SimulationState.hpp File Reference

Defines the SimulationState class.

`#include "KMC_Config.hpp"`

Classes

- class KMCThinFilm::SimulationState

Namespaces

- namespace KMCThinFilm

7.34.1 Detailed Description

Defines the SimulationState class.

7.35 SimulationState.hpp

Go to the documentation of this file.

```
#ifndef SIMULATION_STATE_HPP
#define SIMULATION_STATE_HPP

// Note: This header file is documented via Doxygen
// <http://www.doxygen.org>. Comments for Doxygen begin with '/*!' or
// '//!', and descriptions of functions, class and member functions
// occur *before* their corresponding class declarations and function
// prototypes.


#include "KMC_Config.hpp"

namespace KMCThinFilm {

  class SimulationState {
    friend class Simulation;
  public:
    SimulationState()
      : elapsed_time_(0), t_stop_(0), maxTime_(0),
#if KMC_PARALLEL
        t_sector_(0),
#endif
        num_local_events_(0), num_global_steps_(0)
    {}

    double elapsedTime() const {
#if KMC_PARALLEL
      // This means that I'll have to make sure t_sector_ is zero
      // outside a looping over sectors.
      return elapsed_time_ + t_sector_;
#else
      return elapsed_time_;
#endif
    }

    double maxTime() const {return maxTime_;}

    double globalTimeIncrement() const {return t_stop_;}

    unsigned long long numLocalEvents() const {return num_local_events_;}

    unsigned long long numGlobalSteps() const {return num_global_steps_;}

  private:
    double elapsed_time_, t_stop_, maxTime_;
#if KMC_PARALLEL
    double t_sector_;
#endif
    unsigned long long num_local_events_, num_global_steps_;
  };

}

#endif /* SIMULATION_STATE_HPP */
```

7.36 TimeIncrSchemeVars.hpp File Reference

Defines the TimeIncr::SchemeVars class and related enumerations.

```
#include <string>
#include <boost/scoped_ptr.hpp>
```

Classes

- class KMCThinFilm::TimeIncr::SchemeVars

Namespaces

- namespace KMCThinFilm
- namespace KMCThinFilm::TimeIncr
- namespace KMCThinFilm::TimeIncr::SchemeName
- namespace KMCThinFilm::TimeIncr::SchemeParam

Enumerations

- enum KMCThinFilm::TimeIncr::SchemeName::Type { KMCThinFilm::TimeIncr::SchemeName::MAX_AVG_PROPENSITY_P
, KMCThinFilm::TimeIncr::SchemeName::MAX_SINGLE_PROPENSITY , KMCThinFilm::TimeIncr::SchemeName::FIXED_V
}
- enum KMCThinFilm::TimeIncr::SchemeParam::Type { KMCThinFilm::TimeIncr::SchemeParam::NSTOP , KMCThinFilm::Time
, KMCThinFilm::TimeIncr::SchemeParam::TSTOP }

7.36.1 Detailed Description

Defines the TimeIncr::SchemeVars class and related enumerations.

7.37 TimeIncrSchemeVars.hpp

Go to the documentation of this file.

```
#ifndef TIME_INCR_SCHEMES_HPP
#define TIME_INCR_SCHEMES_HPP

#include <string>
#include <boost/scoped_ptr.hpp>


namespace KMCThinFilm {

  namespace TimeIncr {

    namespace SchemeName {

      enum Type {
        MAX_AVG_PROPENSITY_PER_POSS_EVENT ,
        MAX_SINGLE_PROPENSITY ,
        FIXED_VALUE
        , BAD_VALUE
      };
    }

    namespace SchemeParam {

      enum Type {
        NSTOP ,
        TSTOP_MAX ,
        TSTOP
      };
    }

    class SchemeVars {
    public:

      SchemeVars();

      explicit SchemeVars(const SchemeVars & params);
      SchemeVars & operator=(const SchemeVars & rhs);

      void setSchemeName(SchemeName::Type name);

      void setSchemeParam(SchemeParam::Type paramName ,
                          double paramVal );

      SchemeName::Type getSchemeName() const;

      double getSchemeParamIfAvailable(SchemeParam::Type paramName ,
                                       bool & isAvailable ) const;

      double getSchemeParamOrDie(SchemeParam::Type paramName ,
                                 const std::string & msgIfDie ) const;

      double getSchemeParamOrReturnDefaultVal(SchemeParam::Type paramName ,
                                              double defaultVal ) const;

      ~SchemeVars();

    private:
      class Impl_;
      boost::scoped_ptr<Impl_> pImpl_;
    };

  }

}

#endif /* TIME_INCR_SCHEMES_HPP */
```

Chapter 8

Examples

8.1 testFractal/testFractal.cpp

Driver file for the example implementation of a “fractal” solid-on-solid model, using auto-tracking

This example is discussed in detail in the section “Using auto-tracking.”

```
/*
   Any comments in this code of the form "//! [...]" are used to
   assist Doxygen in documenting this file.  */

#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>

#include "EventsAndActions.hpp"

using namespace KMCThinFilm;

int main() {

  double F = 1, DoverF = 1e5, maxCoverage = 4;
  int domainSize = 256;
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  double approxDepTime = maxCoverage/F;

  LatticeParams latParams;
  latParams.numIntsPerCell = FIntVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;

  Simulation sim(latParams);

  sim.setSolver(sId);

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  sim.setRNG(rng);

  sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(FOverLatticeEvents::DEPOSITION,
                          F, DepositionExecute);

  CellNeighOffsets hopCNO(HopOffset::SIZE);

  hopCNO.addOffset(HopOffset::UP,    CellIndsOffset(0,+1,0));
  hopCNO.addOffset(HopOffset::DOWN,  CellIndsOffset(0,-1,0));
  hopCNO.addOffset(HopOffset::LEFT,  CellIndsOffset(-1,0,0));
  hopCNO.addOffset(HopOffset::RIGHT, CellIndsOffset(+1,0,0));

  hopCNO.addOffset(HopOffset::RIGHT_UP,   CellIndsOffset(+1,+1,0));
  hopCNO.addOffset(HopOffset::RIGHT_DOWN, CellIndsOffset(+1,-1,0));
  hopCNO.addOffset(HopOffset::LEFT_UP,    CellIndsOffset(-1,+1,0));
  hopCNO.addOffset(HopOffset::LEFT_DOWN,  CellIndsOffset(-1,-1,0));

  sim.reserveCellCenteredEventGroups(1,FCellCenteredEvents::SIZE);

  EventExecutorGroup hopExecs(FCellCenteredEvents::SIZE);
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_LEFT,
                            HoppingExecute(FCellCenteredEvents::HOP_LEFT));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_RIGHT,
                            HoppingExecute(FCellCenteredEvents::HOP_RIGHT));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_UP,
                            HoppingExecute(FCellCenteredEvents::HOP_UP));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_DOWN,
                            HoppingExecute(FCellCenteredEvents::HOP_DOWN));

  sim.addCellCenteredEventGroup(1, hopCNO,
                                HoppingPropensity(DoverF*F),
                                hopExecs);

  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintASCII("snapshot"),
                            0.05*approxDepTime, true);

  sim.run(approxDepTime);
  sim.removeOverLatticeEvent(FOverLatticeEvents::DEPOSITION);
  sim.run(0.1*approxDepTime);

  return 0;
}
```

8.2 testFractal/EventsAndActions.hpp

Header file for the example implementation of a “fractal” solid-on-solid model, using auto-tracking

This example is discussed in detail in the section “Using auto-tracking.”

```
#ifndef EVENTS_AND_ACTIONS_HPP
#define EVENTS_AND_ACTIONS_HPP

/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include <string>

#include <KMCThinFilm/CellCenteredGroupPropensities.hpp>
#include <KMCThinFilm/EventExecutor.hpp>
#include <KMCThinFilm/MakeEnum.hpp>

KMC_MAKE_ID_ENUM(FOverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(FCellCenteredEvents,
                 HOP_UP,
                 HOP_DOWN,
                 HOP_LEFT,
                 HOP_RIGHT);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);

KMC_MAKE_LATTICE_INTVAL_ENUM(F, HEIGHT);

/* I have the offsets for hopping include both nearest and
   next-nearest neighbors to make it more obvious what the difference
   is between the FCellCenteredEvents enumeration and the HopOffset
   enumeration. */

KMC_MAKE_OFFSET_ENUM(HopOffset,
                     UP, DOWN, LEFT, RIGHT,
                     RIGHT_UP, RIGHT_DOWN, LEFT_UP, LEFT_DOWN);

void DepositionExecute(const KMCThinFilm::CellInds & ci,
                       const KMCThinFilm::SimulationState & simState,
                       KMCThinFilm::Lattice & lattice);

class HoppingPropensity {
```

```
public:
  HoppingPropensity(double D) : D_(D) {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double D_;
};

class HoppingExecute {
public:
  HoppingExecute(FCellCenteredEvents::Type hopDir);
  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice) const;
private:
  int jump_i_, jump_j_;
};

class PrintASCII {
public:
  PrintASCII(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};

#endif /* EVENTS_AND_ACTIONS_HPP */
```

8.3 testFractal/EventsAndActions.cpp

Implementation file for the example implementation of a “fractal” solid-on-solid model, using auto-tracking

This example is discussed in detail in the section “Using auto-tracking.”

```
/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include "EventsAndActions.hpp"

#include <fstream>
#include <boost/lexical_cast.hpp>

#include <KMCThinFilm/ErrorHandling.hpp>

using namespace KMCThinFilm;

void DepositionExecute(const CellInds & ci,
                       const SimulationState & simState,
                       Lattice & lattice) {

  int currVal = lattice.getInt(ci, FIntVal::HEIGHT);
  lattice.setInt(ci, FIntVal::HEIGHT, currVal + 1);

}

void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  int currHeight = cnp.getInt(cnp.getCellToProbe(HopOffset::SELF), FIntVal::HEIGHT);

  if (currHeight > 0) {
    if ((currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT_UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT_DOWN), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT_UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT_DOWN), FIntVal::HEIGHT))) {

      propensityVec[FCellCenteredEvents::HOP_LEFT] =
        propensityVec[FCellCenteredEvents::HOP_RIGHT] =
        propensityVec[FCellCenteredEvents::HOP_UP] =
        propensityVec[FCellCenteredEvents::HOP_DOWN] = D_;
    }
  }

}

HoppingExecute::HoppingExecute(FCellCenteredEvents::Type hopDir) {

  switch (hopDir) {
  case FCellCenteredEvents::HOP_UP:
    jump_i_ = 0;
    jump_j_ = 1;
    break;
  case FCellCenteredEvents::HOP_DOWN:
    jump_i_ = 0;
    jump_j_ = -1;
    break;
  case FCellCenteredEvents::HOP_LEFT:
    jump_i_ = -1;
    jump_j_ = 0;
    break;
  case FCellCenteredEvents::HOP_RIGHT:
    jump_i_ = 1;
    jump_j_ = 0;
    break;
  default:
    exitWithMsg("Bad hop direction!");
  }
}

void HoppingExecute::operator()(const CellInds & ci,
                                const SimulationState & simState,
                                Lattice & lattice) const {

  CellInds ciTo(ci.i + jump_i_, ci.j + jump_j_, ci.k);

  int currFrom = lattice.getInt(ci, FIntVal::HEIGHT);
  int currTo = lattice.getInt(ciTo, FIntVal::HEIGHT);

  lattice.setInt(ci, FIntVal::HEIGHT, currFrom - 1);
  lattice.setInt(ciTo, FIntVal::HEIGHT, currTo + 1);
}

void PrintASCII::operator()(const SimulationState & simState, Lattice & lattice) {

  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".dat";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int iminGlobal = localPlanarBBox.imin;
  int jminGlobal = localPlanarBBox.jmin;

  // "P1" here is short for "Plus 1".
  int imaxP1Global = localPlanarBBox.imaxP1;
  int jmaxP1Global = localPlanarBBox.jmaxP1;

  outFile << "# " << iminGlobal << " " << imaxP1Global << " " << jminGlobal << " " << jmaxP1Global
          << " time:" << simState.elapsedTime() << "\n";

  CellInds ci; ci.k = 0;
  for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
    for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {
      outFile << ci.i << " " << ci.j << " "
              << lattice.getInt(ci, FIntVal::HEIGHT) << "\n";
    }
  }

  outFile.close();
}
```

8.4 testFractal_semi_manual_track/testFractal.cpp

Driver file for the example implementation of a "fractal" solid-on-solid model, using semi-manual tracking

This example is discussed in detail in the section "Using semi-manual tracking."

```
/*
   Any comments in this code of the form "//! [...]" are used to
   assist Doxygen in documenting this file.  */

#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>

#include "EventsAndActions.hpp"

using namespace KMCThinFilm;

int main() {

  double F = 1, DoverF = 1e5, maxCoverage = 4;
  int domainSize = 256;
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  double approxDepTime = maxCoverage/F;

  LatticeParams latParams;
  latParams.numIntsPerCell = FIntVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;

  Simulation sim(latParams);

  sim.setSolver(sId);

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  sim.setRNG(rng);

  std::vector<CellNeighOffsets> tmpExecCNO;
  tmpExecCNO.reserve(1);

  sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);

  tmpExecCNO.push_back(CellNeighOffsets(1));
  sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(FOverLatticeEvents::DEPOSITION,
                          F, DepositionExecute, tmpExecCNO);

  CellNeighOffsets hopCNO(HopOffset::SIZE);

  hopCNO.addOffset(HopOffset::UP,    CellIndsOffset(0,+1,0));
  hopCNO.addOffset(HopOffset::DOWN,  CellIndsOffset(0,-1,0));
  hopCNO.addOffset(HopOffset::LEFT,  CellIndsOffset(-1,0,0));
  hopCNO.addOffset(HopOffset::RIGHT, CellIndsOffset(+1,0,0));

  hopCNO.addOffset(HopOffset::RIGHT_UP,   CellIndsOffset(+1,+1,0));
  hopCNO.addOffset(HopOffset::RIGHT_DOWN, CellIndsOffset(+1,-1,0));
  hopCNO.addOffset(HopOffset::LEFT_UP,    CellIndsOffset(-1,+1,0));
  hopCNO.addOffset(HopOffset::LEFT_DOWN,  CellIndsOffset(-1,-1,0));

  sim.reserveCellCenteredEventGroups(1,FCellCenteredEvents::SIZE);

  EventExecutorGroup hopExecs(FCellCenteredEvents::SIZE);
  tmpExecCNO.clear();
  tmpExecCNO.push_back(CellNeighOffsets(2));
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::LEFT));

  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_LEFT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::RIGHT));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_RIGHT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::UP));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_UP,
                          HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::DOWN));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_DOWN,
                          HoppingExecute(), tmpExecCNO);

  sim.addCellCenteredEventGroup(1, hopCNO,
                              HoppingPropensity(DoverF*F),
                              hopExecs);

  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                          PrintASCII("snapshot"),
                          0.05*approxDepTime, true);

  sim.run(approxDepTime);
  sim.removeOverLatticeEvent(FOverLatticeEvents::DEPOSITION);
  sim.run(0.1*approxDepTime);

  return 0;
}
```

8.5 testFractal_semi_manual_track/EventsAndActions.hpp

Header file for the example implementation of a “fractal” solid-on-solid model, using semi-manual tracking

This example is discussed in detail in the section “Using semi-manual tracking.”

```
#ifndef EVENTS_AND_ACTIONS_HPP
#define EVENTS_AND_ACTIONS_HPP

/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include <string>

#include <KMCThinFilm/CellCenteredGroupPropensities.hpp>
#include <KMCThinFilm/EventExecutor.hpp>
#include <KMCThinFilm/MakeEnum.hpp>

KMC_MAKE_ID_ENUM(FOverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(FCellCenteredEvents,
                 HOP_UP,
                 HOP_DOWN,
                 HOP_LEFT,
                 HOP_RIGHT);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);

KMC_MAKE_LATTICE_INTVAL_ENUM(F, HEIGHT);

/* I have the offsets for hopping include both nearest and
   next-nearest neighbors to make it more obvious what the difference
   is between the FCellCenteredEvents enumeration and the HopOffset
   enumeration. */

KMC_MAKE_OFFSET_ENUM(HopOffset,
                     UP, DOWN, LEFT, RIGHT,
                     RIGHT_UP, RIGHT_DOWN, LEFT_UP, LEFT_DOWN);

void DepositionExecute(const KMCThinFilm::CellInds & ci,
                       const KMCThinFilm::SimulationState & simState,
                       const KMCThinFilm::Lattice & lattice,
                       std::vector<KMCThinFilm::CellsToChange> & ctcVec);

class HoppingPropensity {
public:
  HoppingPropensity(double D) : D_(D) {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double D_;
};

class HoppingExecute {
public:
  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec) const;
};

class PrintASCII {
public:
  PrintASCII(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};

#endif /* EVENTS_AND_ACTIONS_HPP */
```

8.6 testFractal_semi_manual_track/EventsAndActions.cpp

Implementation file for the example implementation of a “fractal” solid-on-solid model, using semi-manual tracking

This example is discussed in detail in the section “Using semi-manual tracking.”

```
/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include "EventsAndActions.hpp"

#include <fstream>
#include <boost/lexical_cast.hpp>

#include <KMCThinFilm/ErrorHandling.hpp>

using namespace KMCThinFilm;

void DepositionExecute(const CellInds & ci,
                       const SimulationState & simState,
                       const KMCThinFilm::Lattice & lattice,
                       std::vector<KMCThinFilm::CellsToChange> & ctcVec) {

  CellsToChange & ctc = ctcVec[0];
  ctc.setCenter(ci);

  int currVal = lattice.getInt(ci, FIntVal::HEIGHT);
  ctc.setInt(0, FIntVal::HEIGHT, currVal + 1);
}

void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  int currHeight = cnp.getInt(cnp.getCellToProbe(HopOffset::SELF), FIntVal::HEIGHT);

  if (currHeight > 0) {
    if ((currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT_UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT_DOWN), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT_UP), FIntVal::HEIGHT))
        && (currHeight > cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT_DOWN), FIntVal::HEIGHT))) {

      propensityVec[FCellCenteredEvents::HOP_LEFT] =
        propensityVec[FCellCenteredEvents::HOP_RIGHT] =
        propensityVec[FCellCenteredEvents::HOP_UP] =
        propensityVec[FCellCenteredEvents::HOP_DOWN] = D_;
    }
  }

}

void HoppingExecute::operator()(const CellInds & ci,
                                const SimulationState & simState,
                                const KMCThinFilm::Lattice & lattice,
                                std::vector<KMCThinFilm::CellsToChange> & ctcVec) const {

  CellsToChange & ctc = ctcVec[0];
  ctc.setCenter(ci);

  int currFrom = lattice.getInt(ci, FIntVal::HEIGHT);
  int currTo = ctc.getInt(1, FIntVal::HEIGHT);

  ctc.setInt(0, FIntVal::HEIGHT, currFrom - 1);
  ctc.setInt(1, FIntVal::HEIGHT, currTo + 1);
}

void PrintASCII::operator()(const SimulationState & simState, Lattice & lattice) {

  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".dat";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int iminGlobal = localPlanarBBox.imin;
  int jminGlobal = localPlanarBBox.jmin;

  // "P1" here is short for "Plus 1".
  int imaxP1Global = localPlanarBBox.imaxP1;
  int jmaxP1Global = localPlanarBBox.jmaxP1;

  outFile << "# " << iminGlobal << " " << imaxP1Global << " " << jminGlobal << " " << jmaxP1Global
          << " time:" << simState.elapsedTime() << "\n";

  CellInds ci; ci.k = 0;
  for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
    for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {
      outFile << ci.i << " " << ci.j << " "
              << lattice.getInt(ci, FIntVal::HEIGHT) << "\n";
    }
  }

  outFile.close();
}
```

8.7 testFractal_parallel/testFractal.cpp

Driver file for the example implementation of a “fractal” solid-on-solid model that may be used in parallel.

This example is discussed in detail in the section “Example: Parallelizing an implementation of a “fractal” solid-on-solid model.”

```
/*
   Any comments in this code of the form "//! [...]" are used to
   assist Doxygen in documenting this file.  */

#include <KMCThinFilm/Simulation.hpp>

#if KMC_PARALLEL
#include <KMCThinFilm/RandNumGenDCMT.hpp>
#else
#include <KMCThinFilm/RandNumGenMT19937.hpp>
```

```
#endif

#include "EventsAndActions.hpp"

#include <boost/lexical_cast.hpp>

using namespace KMCThinFilm;

int main(int argc, char *argv[]) {

#if KMC_PARALLEL
  MPI_Init(&argc, &argv);
#endif

  double F = 1, DoverF = 1e5, maxCoverage = 4;
  int domainSize = 256;
  unsigned int seedGlobal = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  TimeIncr::SchemeVars schemeVars;
  schemeVars.setSchemeName(TimeIncr::SchemeName::MAX_AVG_PROPENSITY_PER_POSS_EVENT);

  schemeVars.setSchemeParam(TimeIncr::SchemeParam::NSTOP,
#ifndef BAD_NSTOP
                            1
#else
                            100
#endif
                            );

  double approxDepTime = maxCoverage/F;

  LatticeParams latParams;
  latParams.numIntsPerCell = FIntVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;
  latParams.ghostExtent[0] = latParams.ghostExtent[1] = 1;

#ifdef USE_COMPACT_DECOMP
  latParams.parallelDecomp = LatticeParams::COMPACT;
#endif

  Simulation sim(latParams);

  sim.setSolver(sId);

#if KMC_PARALLEL
  RandNumGenSharedPtr rng(new RandNumGenDCMT(sim.procID(),
                                             seedGlobal,
                                             123*sim.procID() + 456,
                                             RandNumGenDCMT::P521));
#else
  RandNumGenSharedPtr rng(new RandNumGenMT19937(seedGlobal));
#endif

  sim.setRNG(rng);

  sim.setTimeIncrScheme(schemeVars);

  std::vector<CellNeighOffsets> tmpExecCNO;
  tmpExecCNO.reserve(1);

  sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);

  tmpExecCNO.push_back(CellNeighOffsets(1));
  sim.reserveOverLatticeEvents(FOverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(FOverLatticeEvents::DEPOSITION,
                          F, DepositionExecute, tmpExecCNO);

  CellNeighOffsets hopCNO(HopOffset::SIZE);

  hopCNO.addOffset(HopOffset::UP,    CellIndsOffset(0,+1,0));
  hopCNO.addOffset(HopOffset::DOWN,  CellIndsOffset(0,-1,0));
  hopCNO.addOffset(HopOffset::LEFT,  CellIndsOffset(-1,0,0));
  hopCNO.addOffset(HopOffset::RIGHT, CellIndsOffset(+1,0,0));

  hopCNO.addOffset(HopOffset::RIGHT_UP,   CellIndsOffset(+1,+1,0));
  hopCNO.addOffset(HopOffset::RIGHT_DOWN, CellIndsOffset(+1,-1,0));
  hopCNO.addOffset(HopOffset::LEFT_UP,    CellIndsOffset(-1,+1,0));
  hopCNO.addOffset(HopOffset::LEFT_DOWN,  CellIndsOffset(-1,-1,0));

  sim.reserveCellCenteredEventGroups(1,FCellCenteredEvents::SIZE);

  EventExecutorGroup hopExecs(FCellCenteredEvents::SIZE);
  tmpExecCNO.clear();
  tmpExecCNO.push_back(CellNeighOffsets(2));
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::LEFT));

  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_LEFT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::RIGHT));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_RIGHT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::UP));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_UP,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::DOWN));
  hopExecs.addEventExecutor(FCellCenteredEvents::HOP_DOWN,
                            HoppingExecute(), tmpExecCNO);

  sim.addCellCenteredEventGroup(1, hopCNO,
                                HoppingPropensity(DoverF*F),
                                hopExecs);

  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintASCII("outFile_ProcCoords" +
                                       boost::lexical_cast<std::string>(sim.commCoord(0)) + "_" +
                                       boost::lexical_cast<std::string>(sim.commCoord(1)) + "_snapshot"),
                            0.05*approxDepTime, true);

  sim.run(approxDepTime);
  sim.removeOverLatticeEvent(FOverLatticeEvents::DEPOSITION);
  sim.run(0.1*approxDepTime);

#if KMC_PARALLEL
  MPI_Finalize();
#endif

  return 0;
}
```

8.8 testBallisticDep1/testBallisticDep.cpp

Driver file for an example implementation of a ballistic deposition model

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “First implementation.”

```
#include <iostream>

#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>
#include <KMCThinFilm/MakeEnum.hpp>

#include "EventsAndActions.hpp"
#include "InitLattice.hpp"

using namespace KMCThinFilm;

int main() {

  // Parameters used in the simulation
  double F = 1, maxCoverage = 4;
  int domainSize = 100;
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  double approxDepTime = maxCoverage/F;

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  LatticeParams latParams;
  latParams.numIntsPerCell = BDIntVal::SIZE;
  latParams.numFloatsPerCell = BDFloatVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;
  latParams.setEmptyCellVals = SetEmptyCellWithRandColor(rng);
  latParams.numPlanesToReserve = 100;

  Simulation sim(latParams);

  sim.setSolver(sId);

  sim.setRNG(rng);

  sim.reserveOverLatticeEvents(OverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(OverLatticeEvents::DEPOSITION,
                          F, DepositionExecute());

  CellNeighOffsets mixCNO(MIX_OFFSET::SIZE);
  mixCNO.addOffset(MIX_OFFSET::NORTH, CellIndsOffset(+1, 0, 0));
  mixCNO.addOffset(MIX_OFFSET::SOUTH, CellIndsOffset(-1, 0, 0));
  mixCNO.addOffset(MIX_OFFSET::WEST,  CellIndsOffset( 0,-1, 0));
  mixCNO.addOffset(MIX_OFFSET::EAST,  CellIndsOffset( 0,+1, 0));
  mixCNO.addOffset(MIX_OFFSET::UP,    CellIndsOffset( 0, 0,+1));
  mixCNO.addOffset(MIX_OFFSET::DOWN,  CellIndsOffset( 0, 0,-1));

  int numMixes;

  EventExecutorGroup mixExec(CellCenteredEvents::SIZE);

  mixExec.addEventExecutor(CellCenteredEvents::COLOR_MIXING,
                           ColorMixExecute(rng, &mixCNO, &numMixes));

  sim.reserveCellCenteredEventGroups(1, CellCenteredEvents::SIZE);
  sim.addCellCenteredEventGroup(1, mixCNO,
                                ColorMixPropensity(10*F),
                                mixExec);

  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintPoint3D("snapshot"),
                            0.05*approxDepTime, true);

  sim.run(approxDepTime);

  std::cout << "Number of color mixes = " << numMixes << "\n";

  return 0;
}
```

8.9 testBallisticDep1/EventsAndActions.hpp

Header file for an example implementation of a ballistic deposition model

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “First implementation.”

```
#ifndef EVENTS_AND_ACTIONS_HPP
#define EVENTS_AND_ACTIONS_HPP

#include <KMCThinFilm/CellNeighOffsets.hpp>
#include <KMCThinFilm/CellCenteredGroupPropensities.hpp>
#include <KMCThinFilm/EventExecutor.hpp>
#include <KMCThinFilm/MakeEnum.hpp>
#include <KMCThinFilm/RandNumGen.hpp>

#include <vector>
#include <string>

KMC_MAKE_ID_ENUM(OverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(CellCenteredEvents,
                 COLOR_MIXING);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);

KMC_MAKE_LATTICE_INTVAL_ENUM(BD, IS_OCCUPIED, ACTIVE_ZONE_HEIGHT);

KMC_MAKE_LATTICE_FLOATVAL_ENUM(BD, COLOR);

KMC_MAKE_OFFSET_ENUM(MIX_OFFSET,
                     NORTH, SOUTH, WEST, EAST /* First four neighbors are lateral */,
                     UP, DOWN);

class DepositionExecute {
public:
  DepositionExecute();

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);
private:
  std::vector<KMCThinFilm::CellIndsOffset> neighOffsets_;
};

class ColorMixPropensity {
public:
  ColorMixPropensity(double mixPropPerNeighbor)
    : mixPropPerNeighbor_(mixPropPerNeighbor)
  {}

  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double mixPropPerNeighbor_;
};

class ColorMixExecute {
public:
  ColorMixExecute(KMCThinFilm::RandNumGenSharedPtr rng_,
                  const KMCThinFilm::CellNeighOffsets * mixCNO,
                  int * numMixes);

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);
private:
  KMCThinFilm::RandNumGenSharedPtr rng_;
  const KMCThinFilm::CellNeighOffsets * mixCNO_;
  int * numMixes_;
};

class PrintPoint3D {
public:
  PrintPoint3D(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};

#endif /* EVENTS_AND_ACTIONS_HPP */
```

8.10 testBallisticDep1/EventsAndActions.cpp

Primary implementation file for an example implementation of a ballistic deposition model

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “First implementation.”

```
#include "EventsAndActions.hpp"

#include <fstream>

#include <boost/lexical_cast.hpp>

using namespace KMCThinFilm;

DepositionExecute::DepositionExecute() {

  neighOffsets_.reserve(4);
  neighOffsets_.push_back(CellIndsOffset( 0,-1));
  neighOffsets_.push_back(CellIndsOffset( 0,+1));
  neighOffsets_.push_back(CellIndsOffset(-1, 0));
  neighOffsets_.push_back(CellIndsOffset(+1, 0));

}

void DepositionExecute::operator()(const CellInds & ci,
                                   const SimulationState & simState,
                                   Lattice & lattice) {

  /* Ballistic deposition algorithm for cubic lattices from Meakin and
     Krug, Physical Review A, vol. 46, num. 6, pp. 3390-3399 (1992).*/

  CellInds ciInPlane(ci.i, ci.j, 0);

  int kDepAtom = lattice.getInt(ciInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT);

  CellInds ciTo(ci.i, ci.j, kDepAtom);

  lattice.addPlanes(ciTo.k - ci.k);
  lattice.setInt(ciTo, BDIntVal::IS_OCCUPIED, 1);

  for (std::vector<CellIndsOffset>::const_iterator offsetItr = neighOffsets_.begin(),
         offsetItrEnd = neighOffsets_.end(); offsetItr != offsetItrEnd; ++offsetItr) {

    CellInds neighInPlane = ciInPlane + *offsetItr;

    if (lattice.getInt(neighInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT) < kDepAtom) {
      lattice.setInt(neighInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT, kDepAtom);
    }
  }

  lattice.setInt(ciInPlane, BDIntVal::ACTIVE_ZONE_HEIGHT, kDepAtom + 1);

}

void ColorMixPropensity::operator()(const KMCThinFilm::CellNeighProbe & cnp,
                                    std::vector<double> & propensityVec) const {

  if (cnp.getInt(cnp.getCellToProbe(MIX_OFFSET::SELF), BDIntVal::IS_OCCUPIED)) {

    int numNeighs = 0;

    // Visiting four lateral neighbors
    for (int whichOffset = 1; whichOffset <= 4; ++whichOffset) {

      if (cnp.getInt(cnp.getCellToProbe(whichOffset), BDIntVal::IS_OCCUPIED)) {
        ++numNeighs;
      }

    }

    CellToProbe downCell = cnp.getCellToProbe(MIX_OFFSET::DOWN);
    if (cnp.belowLatticeBottom(downCell) || cnp.getInt(downCell, BDIntVal::IS_OCCUPIED)) {
      ++numNeighs;
    }

    CellToProbe upCell = cnp.getCellToProbe(MIX_OFFSET::UP);
    if ((!cnp.exceedsLatticeHeight(upCell)) && cnp.getInt(upCell, BDIntVal::IS_OCCUPIED)) {
      ++numNeighs;
    }

    propensityVec[CellCenteredEvents::COLOR_MIXING] = mixPropPerNeighbor_*numNeighs;

  }
}

ColorMixExecute::ColorMixExecute(RandNumGenSharedPtr rng,
                                 const CellNeighOffsets * mixCNO,
                                 int * numMixes)
  : rng_(rng),
    mixCNO_(mixCNO),
    numMixes_(numMixes) {

  *numMixes_ = 0;

}

void ColorMixExecute::operator()(const KMCThinFilm::CellInds & ci,
                                 const KMCThinFilm::SimulationState & simState,
                                 KMCThinFilm::Lattice & lattice) {

  double color = lattice.getFloat(ci, BDFloatVal::COLOR);
  int numColors = 1;

  // Visiting four lateral neighbors
  for (int whichOffset = 1; whichOffset <= 4; ++whichOffset) {

    CellInds ciNeigh = ci + mixCNO_->getOffset(whichOffset);

    if (lattice.getInt(ciNeigh, BDIntVal::IS_OCCUPIED)) {
      color += lattice.getFloat(ciNeigh, BDFloatVal::COLOR);
      ++numColors;
    }

  }

  CellInds ciDown = ci +  mixCNO_->getOffset(MIX_OFFSET::DOWN);

  bool belowLatticeBottom = (ciDown.k < 0);
  bool ciDownIsOccupied = (belowLatticeBottom || lattice.getInt(ciDown, BDIntVal::IS_OCCUPIED));

  if (ciDownIsOccupied) {
    color += (belowLatticeBottom ?
              (rng_->getNumInOpenIntervalFrom0To1()) :
              lattice.getFloat(ciDown, BDFloatVal::COLOR));
    ++numColors;
  }

  CellInds ciUp = ci +  mixCNO_->getOffset(MIX_OFFSET::UP);

  if ((ciUp.k < lattice.currHeight()) && lattice.getInt(ciUp, BDIntVal::IS_OCCUPIED)) {

    color += lattice.getFloat(ciUp, BDFloatVal::COLOR);
    ++numColors;

  }

  lattice.setFloat(ci, BDFloatVal::COLOR, color/numColors);

  ++(*numMixes_);

}

void PrintPoint3D::operator()(const SimulationState & simState, Lattice & lattice) {
  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".3D";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  outFile << "x y z value\n";

  CellInds ci;
  int kMaxP1 = lattice.currHeight();

  for (ci.k = 0; ci.k < kMaxP1; ++(ci.k)) {
    for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
      for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {

        if (lattice.getInt(ci, BDIntVal::IS_OCCUPIED) > 0) {
          outFile << ci.i << " " << ci.j << " "
                  << ci.k << " " << lattice.getFloat(ci, BDFloatVal::COLOR) << "\n";
        }
      }
    }
  }

  outFile.close();
}
```

8.11 testBallisticDep1/InitLattice.hpp

Header file for an example implementation of a ballistic deposition model that defines a custom method for initializing empty cells of a lattice.

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “First implementation.”

```
#ifndef INIT_LATTICE_HPP
#define INIT_LATTICE_HPP

#include <KMCThinFilm/Lattice.hpp>
#include <KMCThinFilm/RandNumGen.hpp>

class SetEmptyCellWithRandColor {
public:
  SetEmptyCellWithRandColor(KMCThinFilm::RandNumGenSharedPtr rng)
    : rng_(rng)
  {}

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<int> & emptyIntVals,
                  std::vector<double> & emptyFloatVals);
private:
  KMCThinFilm::RandNumGenSharedPtr rng_;
};

#endif /* INIT_LATTICE_HPP */
```

8.12 testBallisticDep1/InitLattice.cpp

An implementation file for an example implementation of a ballistic deposition model that defines a custom method for initializing empty cells of a lattice.

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “First implementation.”

```
#include "InitLattice.hpp"
#include "EventsAndActions.hpp"

using namespace KMCThinFilm;

void SetEmptyCellWithRandColor::operator()(const CellInds & ci,
                                           const Lattice & lattice,
                                           std::vector<int> & emptyIntVals,
                                           std::vector<double> & emptyFloatVals) {

  emptyFloatVals[BDFloatVal::COLOR] = rng_->getNumInOpenIntervalFrom0To1();

}
```

8.13 testBallisticDep2/testBallisticDep.cpp

Driver file for an example implementation of a ballistic deposition model

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “Second implementation.”

```
#include <iostream>

#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>
```

```
#include <KMCThinFilm/MakeEnum.hpp>

#include "EventsAndActions.hpp"
#include "InitLattice.hpp"

using namespace KMCThinFilm;

int main() {

  // Parameters used in the simulation
  double F = 1, maxCoverage = 4;
  int domainSize = 100;
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  double approxDepTime = maxCoverage/F;

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  LatticeParams latParams;
  latParams.numIntsPerCell = BDIntVal::SIZE;
  latParams.numFloatsPerCell = BDFloatVal::SIZE;
  latParams.globalPlanarDims[0] = latParams.globalPlanarDims[1] = domainSize;
  latParams.setEmptyCellVals = SetEmptyCellWithRandColor(rng);
  latParams.numPlanesToReserve = 100;

  Simulation sim(latParams);

  sim.setSolver(sId);

  sim.setRNG(rng);

  KMCThinFilm::LatticePlanarBBox planarBBox;
  sim.getLatticeGlobalPlanarBBox(planarBBox);

  std::vector<CellNeighOffsets> tmpExecCNO;
  tmpExecCNO.reserve(1);
  tmpExecCNO.push_back(CellNeighOffsets(1));

  sim.reserveOverLatticeEvents(OverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(OverLatticeEvents::DEPOSITION,
                          F, DepositionExecute(planarBBox), tmpExecCNO);

  CellNeighOffsets mixCNO(MIX_OFFSET::SIZE);
  mixCNO.addOffset(MIX_OFFSET::NORTH, CellIndsOffset(+1, 0, 0));
  mixCNO.addOffset(MIX_OFFSET::SOUTH, CellIndsOffset(-1, 0, 0));
  mixCNO.addOffset(MIX_OFFSET::WEST,  CellIndsOffset( 0,-1, 0));
  mixCNO.addOffset(MIX_OFFSET::EAST,  CellIndsOffset( 0,+1, 0));
  mixCNO.addOffset(MIX_OFFSET::UP,    CellIndsOffset( 0, 0,+1));
  mixCNO.addOffset(MIX_OFFSET::DOWN,  CellIndsOffset( 0, 0,-1));

  int numMixes;

  EventExecutorGroup mixExec(CellCenteredEvents::SIZE);

  mixExec.addEventExecutor(CellCenteredEvents::COLOR_MIXING,
                           ColorMixExecute(rng, &mixCNO, &numMixes),
                           tmpExecCNO);

  sim.reserveCellCenteredEventGroups(1, CellCenteredEvents::SIZE);
  sim.addCellCenteredEventGroup(1, mixCNO,
                                ColorMixPropensity(10*F),
                                mixExec);

  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintPoint3D("snapshot"),
                            0.05*approxDepTime, true);

  sim.run(approxDepTime);

  std::cout « "Number of color mixes = " « numMixes « "\n";

  return 0;
}
```

8.14 testBallisticDep2/EventsAndActions.hpp

Header file for an example implementation of a ballistic deposition model

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “Second implementation.”

```
#ifndef EVENTS_AND_ACTIONS_HPP
#define EVENTS_AND_ACTIONS_HPP

#include <KMCThinFilm/CellNeighOffsets.hpp>
#include <KMCThinFilm/CellCenteredGroupPropensities.hpp>
#include <KMCThinFilm/EventExecutor.hpp>
#include <KMCThinFilm/MakeEnum.hpp>
#include <KMCThinFilm/RandNumGen.hpp>

#include <vector>
#include <string>

#include <boost/multi_array.hpp>
#include <boost/shared_ptr.hpp>

typedef boost::multi_array<int, 2> IntArray2D;
typedef boost::shared_ptr<IntArray2D> IntArray2DSharedPtr;

KMC_MAKE_ID_ENUM(OverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(CellCenteredEvents,
                 COLOR_MIXING);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);

KMC_MAKE_LATTICE_INTVAL_ENUM(BD, IS_OCCUPIED, ACTIVE_ZONE_HEIGHT);

KMC_MAKE_LATTICE_FLOATVAL_ENUM(BD, COLOR);

KMC_MAKE_OFFSET_ENUM(MIX_OFFSET,
                     NORTH, SOUTH, WEST, EAST /* First four neighbors are lateral */,
                     UP, DOWN);

class DepositionExecute {
public:
  DepositionExecute(const KMCThinFilm::LatticePlanarBBox & planarBBox);

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec);
private:
  IntArray2DSharedPtr activeZoneHeights_;

  std::vector<KMCThinFilm::CellIndsOffset> neighOffsets_;
};

class ColorMixPropensity {
public:
  ColorMixPropensity(double mixPropPerNeighbor)
    : mixPropPerNeighbor_(mixPropPerNeighbor)
  {}

  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double mixPropPerNeighbor_;
};

class ColorMixExecute {
public:
  ColorMixExecute(KMCThinFilm::RandNumGenSharedPtr rng_,
                  const KMCThinFilm::CellNeighOffsets * mixCNO,
                  int * numMixes);

  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec);
private:
  KMCThinFilm::RandNumGenSharedPtr rng_;
  const KMCThinFilm::CellNeighOffsets * mixCNO_;
  int * numMixes_;
};

class PrintPoint3D {
public:
  PrintPoint3D(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};

#endif /* EVENTS_AND_ACTIONS_HPP */
```

8.15 testBallisticDep2/EventsAndActions.cpp

Primary implementation file for an example implementation of a ballistic deposition model

This example is discussed in detail in the section “Example: Implementations of a ballistic deposition model,” subsection “Second implementation.”

```
#include "EventsAndActions.hpp"

#include <fstream>

#include <boost/lexical_cast.hpp>

using namespace KMCThinFilm;

DepositionExecute::DepositionExecute(const LatticePlanarBBox & planarBBox)
: activeZoneHeights_(new IntArray2D) {

  // The resize() member function will initialize the elements of this
  // array to zero.
  activeZoneHeights_->resize(boost::extents[planarBBox.imaxP1 - planarBBox.imin]
                             [planarBBox.jmaxP1 - planarBBox.jmin]);

  neighOffsets_.reserve(4);
  neighOffsets_.push_back(CellIndsOffset( 0,-1));
  neighOffsets_.push_back(CellIndsOffset( 0,+1));
  neighOffsets_.push_back(CellIndsOffset(-1, 0));
  neighOffsets_.push_back(CellIndsOffset(+1, 0));

}

void DepositionExecute::operator()(const KMCThinFilm::CellInds & ci,
                                   const KMCThinFilm::SimulationState & simState,
                                   const KMCThinFilm::Lattice & lattice,
                                   std::vector<KMCThinFilm::CellsToChange> & ctcVec) {

  /* Ballistic deposition algorithm for cubic lattices from Meakin and
     Krug, Physical Review A, vol. 46, num. 6, pp. 3390-3399 (1992).*/

  int kDepAtom = (*activeZoneHeights_)[ci.i][ci.j];

  CellInds ciTo(ci.i, ci.j, kDepAtom);

  CellsToChange & ctc = ctcVec[0];

  ctc.setCenter(ciTo);

  ctc.addLatticePlanes(ciTo.k - ci.k);
  ctc.setInt(0, BDIntVal::IS_OCCUPIED, 1);

  for (std::vector<CellIndsOffset>::const_iterator offsetItr = neighOffsets_.begin(),
         offsetItrEnd = neighOffsets_.end(); offsetItr != offsetItrEnd; ++offsetItr) {

    CellInds neigh = ciTo + *offsetItr;

    int i = lattice.wrapI(neigh);
    int j = lattice.wrapJ(neigh);

    if ((*activeZoneHeights_)[i][j] < kDepAtom) {
      (*activeZoneHeights_)[i][j] = kDepAtom;
    }
  }

  ++((*activeZoneHeights_)[ci.i][ci.j]);
}

void ColorMixPropensity::operator()(const KMCThinFilm::CellNeighProbe & cnp,
                                    std::vector<double> & propensityVec) const {

  if (cnp.getInt(cnp.getCellToProbe(MIX_OFFSET::SELF), BDIntVal::IS_OCCUPIED)) {

    int numNeighs = 0;

    // Visiting four lateral neighbors
    for (int whichOffset = 1; whichOffset <= 4; ++whichOffset) {

      if (cnp.getInt(cnp.getCellToProbe(whichOffset), BDIntVal::IS_OCCUPIED)) {
        ++numNeighs;
      }

    }

    CellToProbe downCell = cnp.getCellToProbe(MIX_OFFSET::DOWN);
    if (cnp.belowLatticeBottom(downCell) || cnp.getInt(downCell, BDIntVal::IS_OCCUPIED)) {
      ++numNeighs;
    }

    CellToProbe upCell = cnp.getCellToProbe(MIX_OFFSET::UP);
    if ((!cnp.exceedsLatticeHeight(upCell)) && cnp.getInt(upCell, BDIntVal::IS_OCCUPIED)) {
      ++numNeighs;
    }

    propensityVec[CellCenteredEvents::COLOR_MIXING] = mixPropPerNeighbor_*numNeighs;

  }
}

ColorMixExecute::ColorMixExecute(RandNumGenSharedPtr rng,
                                 const CellNeighOffsets * mixCNO,
                                 int * numMixes)
  : rng_(rng),
    mixCNO_(mixCNO),
    numMixes_(numMixes) {

  *numMixes_ = 0;

}

void ColorMixExecute::operator()(const KMCThinFilm::CellInds & ci,
                                 const KMCThinFilm::SimulationState & simState,
                                 const KMCThinFilm::Lattice & lattice,
                                 std::vector<KMCThinFilm::CellsToChange> & ctcVec) {

  double color = lattice.getFloat(ci, BDFloatVal::COLOR);
  int numColors = 1;

  // Visiting four lateral neighbors
  for (int whichOffset = 1; whichOffset <= 4; ++whichOffset) {

    CellInds ciNeigh = ci + mixCNO_->getOffset(whichOffset);

    if (lattice.getInt(ciNeigh, BDIntVal::IS_OCCUPIED)) {
      color += lattice.getFloat(ciNeigh, BDFloatVal::COLOR);
      ++numColors;
    }

  }

  CellInds ciDown = ci +  mixCNO_->getOffset(MIX_OFFSET::DOWN);

  bool belowLatticeBottom = (ciDown.k < 0);
  bool ciDownIsOccupied = (belowLatticeBottom || lattice.getInt(ciDown, BDIntVal::IS_OCCUPIED));

  if (ciDownIsOccupied) {
    color += (belowLatticeBottom ?
              (rng_->getNumInOpenIntervalFrom0To1()) :
              lattice.getFloat(ciDown, BDFloatVal::COLOR));
    ++numColors;
  }


  CellInds ciUp = ci +  mixCNO_->getOffset(MIX_OFFSET::UP);

  if ((ciUp.k < lattice.currHeight()) && lattice.getInt(ciUp, BDIntVal::IS_OCCUPIED)) {

    color += lattice.getFloat(ciUp, BDFloatVal::COLOR);
    ++numColors;

  }

  CellsToChange & ctc = ctcVec[0];

  ctc.setCenter(ci);
  ctc.setFloat(0, BDFloatVal::COLOR, color/numColors);

  ++(*numMixes_);

}

void PrintPoint3D::operator()(const SimulationState & simState, Lattice & lattice) {
  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".3D";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  outFile « "x y z value\n";

  CellInds ci;
  int kMaxP1 = lattice.currHeight();

  for (ci.k = 0; ci.k < kMaxP1; ++(ci.k)) {
    for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
      for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {

        if (lattice.getInt(ci, BDIntVal::IS_OCCUPIED) > 0) {
          outFile « ci.i « " " « ci.j « " "
                  « ci.k « " " « lattice.getFloat(ci, BDFloatVal::COLOR) « "\n";
        }
      }
    }
  }

  outFile.close();
}
```

8.16 testPatternedSurface1/testPatternedSurface.cpp

Driver file for an example implementation of a model with a patterned substrate surface

This example is discussed in detail in the section “Example: Implementations of a patterned substrate model,” subsection “First implementation.”

```
/*
   Any comments in this code of the form "//! [...]" are used to
   assist Doxygen in documenting this file.  */

#include <string>
#include <fstream>

#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>
#include <KMCThinFilm/ErrorHandling.hpp>

#include "EventsAndActions.hpp"
```

```
#include "InitLattice.hpp"

using namespace KMCThinFilm;

int main() {

  double F = 0.0033, maxCoverage = 0.15, E_n = 0.18, T = 390;
  std::string patternFName("../tiled16x16Domain.dat");
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  double approxDepTime = maxCoverage/F;

  LatticeParams latParams;
  latParams.numIntsPerCell = PSIntVal::SIZE;
  latParams.numFloatsPerCell = PSFloatVal::SIZE;

  std::ifstream patternFile(patternFName.c_str());

  if (patternFile) {
    patternFile » latParams.globalPlanarDims[0] » latParams.globalPlanarDims[1];
  }
  else {
    exitWithMsg("Cannot access " + patternFName +
                ". Check if your are executing this program from the correct directory.");
  }

  patternFile.close();

  latParams.latInit = InitLatticeFromFile(patternFName);

  Simulation sim(latParams);

  sim.setSolver(sId);

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  sim.setRNG(rng);

  sim.reserveOverLatticeEvents(PSOverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(PSOverLatticeEvents::DEPOSITION,
                          F, DepositionExecute);

  CellNeighOffsets hopCNO(HopOffset::SIZE);

  hopCNO.addOffset(HopOffset::UP,    CellIndsOffset(0,+1,0));
  hopCNO.addOffset(HopOffset::DOWN,  CellIndsOffset(0,-1,0));
  hopCNO.addOffset(HopOffset::LEFT,  CellIndsOffset(-1,0,0));
  hopCNO.addOffset(HopOffset::RIGHT, CellIndsOffset(+1,0,0));

  sim.reserveCellCenteredEventGroups(1, PSCellCenteredEvents::SIZE);

  EventExecutorGroup hopExecs(PSCellCenteredEvents::SIZE);

  std::vector<CellNeighOffsets> tmpExecCNO;
  tmpExecCNO.reserve(1);
  tmpExecCNO.push_back(CellNeighOffsets(2));
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::LEFT));

  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_LEFT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::RIGHT));
  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_RIGHT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::UP));
  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_UP,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::DOWN));
  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_DOWN,
                            HoppingExecute(), tmpExecCNO);

  sim.addCellCenteredEventGroup(1, hopCNO, HoppingPropensity(E_n, T), hopExecs);


  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintASCII("snapshot"),
                            0.05*approxDepTime, true);

  sim.run(approxDepTime);
  sim.removeOverLatticeEvent(PSOverLatticeEvents::DEPOSITION);
  sim.run(0.1*approxDepTime);

  return 0;
}
```

8.17 testPatternedSurface1/EventsAndActions.hpp

Header file for an example implementation of a model with a patterned substrate surface

This example is discussed in detail in the section "Example: Implementations of a patterned substrate model," subsection "First implementation."

```
#ifndef EVENTS_AND_ACTIONS_HPP
#define EVENTS_AND_ACTIONS_HPP

/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include <string>

#include <KMCThinFilm/CellCenteredGroupPropensities.hpp>
#include <KMCThinFilm/EventExecutor.hpp>
#include <KMCThinFilm/MakeEnum.hpp>

#include "PhysicalConstants.hpp"

KMC_MAKE_ID_ENUM(PSOverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(PSCellCenteredEvents,
                 HOP_UP,
                 HOP_DOWN,
                 HOP_LEFT,
                 HOP_RIGHT);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);

KMC_MAKE_LATTICE_INTVAL_ENUM(PS, HEIGHT);
KMC_MAKE_LATTICE_FLOATVAL_ENUM(PS, E_s);

KMC_MAKE_OFFSET_ENUM(HopOffset,
                     UP, DOWN, LEFT, RIGHT);

void DepositionExecute(const KMCThinFilm::CellInds & ci,
                       const KMCThinFilm::SimulationState & simState,
                       KMCThinFilm::Lattice & lattice);

class HoppingPropensity {
public:
  HoppingPropensity(double E_n, double T)
    : E_n_(E_n), kBT_(PhysConst::kB*T), k_(kBT_/PhysConst::h)
  {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  double E_n_, kBT_, k_;
};

class HoppingExecute {
public:
  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec) const;
};

class PrintASCII {
public:
  PrintASCII(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};

#endif /* EVENTS_AND_ACTIONS_HPP */
```

8.18 testPatternedSurface1/EventsAndActions.cpp

Primary implementation file for an example implementation of a model with a patterned substrate surface

This example is discussed in detail in the section "Example: Implementations of a patterned substrate model," subsection "First implementation."

```
/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include "EventsAndActions.hpp"

#include <vector>
#include <fstream>
#include <cmath>

#include <boost/lexical_cast.hpp>

#include <KMCThinFilm/ErrorHandling.hpp>

using namespace KMCThinFilm;

void DepositionExecute(const CellInds & ci,
                       const SimulationState & simState,
                       Lattice & lattice) {

  int currVal = lattice.getInt(ci, PSIntVal::HEIGHT);
  lattice.setInt(ci, PSIntVal::HEIGHT, currVal + 1);

}

void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  KMCThinFilm::CellToProbe ctpSelf = cnp.getCellToProbe(HopOffset::SELF);

  int currHeight = cnp.getInt(ctpSelf, PSIntVal::HEIGHT);

  if (currHeight > 0) {

    int n = 0;

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::UP), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    double E = cnp.getFloat(ctpSelf, PSFloatVal::E_s) + n*E_n_;

    double p = k_*std::exp(-E/kBT_);

    for (int i = 0; i < PSCellCenteredEvents::SIZE; ++i) {
      propensityVec[i] = p;
    }

  }
}

void HoppingExecute::operator()(const CellInds & ci,
                                const SimulationState & simState,
                                const Lattice & lattice,
                                std::vector<CellsToChange> & ctcVec) const {

  CellsToChange & ctc = ctcVec[0];
  ctc.setCenter(ci);

  int currFrom = lattice.getInt(ci, PSIntVal::HEIGHT);
  int currTo = ctc.getInt(1, PSIntVal::HEIGHT);

  ctc.setInt(0, PSIntVal::HEIGHT, currFrom - 1);
  ctc.setInt(1, PSIntVal::HEIGHT, currTo + 1);
}

void PrintASCII::operator()(const SimulationState & simState, Lattice & lattice) {

  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".dat";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int iminGlobal = localPlanarBBox.imin;
  int jminGlobal = localPlanarBBox.jmin;

  // "P1" here is short for "Plus 1".
  int imaxP1Global = localPlanarBBox.imaxP1;
  int jmaxP1Global = localPlanarBBox.jmaxP1;

  outFile << "# " << iminGlobal << " " << imaxP1Global << " " << jminGlobal << " " << jmaxP1Global
          << " time:" << simState.elapsedTime() << "\n";

  CellInds ci; ci.k = 0;
  for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
    for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {
      outFile << ci.i << " " << ci.j << " "
              << lattice.getInt(ci, PSIntVal::HEIGHT) << "\n";
    }
  }

  outFile.close();
}
```

8.19 testPatternedSurface1/InitLattice.hpp

Header file for an example implementation of a model with a patterned substrate surface that defines a custom method for initializing a lattice.

This example is discussed in detail in the section "Example: Implementations of a patterned substrate model," subsection "First implementation."

```
#ifndef INIT_LATTICE_HPP
#define INIT_LATTICE_HPP

#include <string>

#include <KMCThinFilm/Lattice.hpp>
#include <KMCThinFilm/ErrorHandling.hpp>

class InitLatticeFromFile {
```

```
public:
  InitLatticeFromFile(const std::string & inpFName)
    : inpFName_(inpFName)
  {}

  void operator()(KMCThinFilm::Lattice & lattice) const;
private:
  std::string inpFName_;
};

#endif /* INIT_LATTICE_HPP */
```

8.20 testPatternedSurface1/InitLattice.cpp

An implementation file for an example implementation of a model with a patterned substrate surface that defines a custom method for initializing a lattice.

This example is discussed in detail in the section “Example: Implementations of a patterned substrate model,” subsection “First implementation.”

```
#include "InitLattice.hpp"
#include "EventsAndActions.hpp"

#include <fstream>

#include <boost/array.hpp>
#include <boost/lexical_cast.hpp>

using namespace KMCThinFilm;

void InitLatticeFromFile::operator()(Lattice & lattice) const {

  std::ifstream inpFile(inpFName_.c_str());

  boost::array<int,2> globalPlanarDims;

  inpFile >> globalPlanarDims[0] >> globalPlanarDims[1];

  LatticePlanarBBox globalPlanarBBox;
  lattice.getGlobalPlanarBBox(globalPlanarBBox);

  exitOnCondition((globalPlanarDims[0] != (globalPlanarBBox.imaxP1 - globalPlanarBBox.imin)) ||
                  (globalPlanarDims[1] != (globalPlanarBBox.jmaxP1 - globalPlanarBBox.jmin)),
                  "Mismatch in lattice dimensions and dimensions of strain eng. den. array.");

  lattice.addPlanes(1);

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int maxLineNumP1 = (globalPlanarDims[0])*(globalPlanarDims[1]);

  CellInds ci; ci.k = 0;
  for (int lineNum = 0; lineNum < maxLineNumP1; ++lineNum) {

    int i, j;
    double E_s;

    inpFile >> i >> j >> E_s;

    if ((i >= localPlanarBBox.imin) && (i < localPlanarBBox.imaxP1) &&
        (j >= localPlanarBBox.jmin) && (j < localPlanarBBox.jmaxP1)) {

      ci.i = i;
      ci.j = j;

      lattice.setFloat(ci, PSFloatVal::E_s, E_s);
    }

  }

  inpFile.close();

}
```

8.21 testPatternedSurface1/PhysicalConstants.hpp

Header file defining some physical constants for a model with a patterned substrate surface.

```
#ifndef PHYSICAL_CONSTANTS_HPP
#define PHYSICAL_CONSTANTS_HPP

namespace PhysConst {
  const double kB = 8.6173324e-5; /* Boltzmann constant eV/Kelvin */
  const double h = 4.135667516e-15; /* Planck's constant eV*seconds */
}

#endif /* PHYSICAL_CONSTANTS_HPP */
```

8.22 testPatternedSurface2/testPatternedSurface.cpp

Driver file for another example implementation of a model with a patterned substrate surface

This example is discussed in detail in the section "Example: Implementations of a patterned substrate model," subsection "First implementation."

```
/*
   Any comments in this code of the form "//! [...]" are used to
   assist Doxygen in documenting this file.  */

#include <string>
#include <fstream>

#include <boost/array.hpp>

#include <KMCThinFilm/Simulation.hpp>
#include <KMCThinFilm/RandNumGenMT19937.hpp>
#include <KMCThinFilm/ErrorHandling.hpp>

#include "EventsAndActions.hpp"

using namespace KMCThinFilm;

int main() {

  double F = 0.0033, maxCoverage = 0.15, E_n = 0.18, T = 390;
  int numDomains = 16;
  unsigned int seed = 42;
  SolverId::Type sId = SolverId::DYNAMIC_SCHULZE;

  DblArray2D E_s;

  std::ifstream patternFile("../singleDomain.dat");
  boost::array<int,2> E_s_extents;

  patternFile >> E_s_extents[0] >> E_s_extents[1];
  E_s.resize(E_s_extents);

  int E_s_i, E_s_j;
  double E_s_val;

  while (patternFile >> E_s_i >> E_s_j >> E_s_val) {
    E_s[E_s_i][E_s_j] = E_s_val;
  }

  double approxDepTime = maxCoverage/F;

  LatticeParams latParams;
  latParams.numIntsPerCell = PSIntVal::SIZE;
  latParams.numFloatsPerCell = PSFloatVal::SIZE;
  latParams.globalPlanarDims[0] = numDomains*(E_s.shape()[0]);
  latParams.globalPlanarDims[1] = numDomains*(E_s.shape()[1]);

  Simulation sim(latParams);

  sim.setSolver(sId);

  RandNumGenSharedPtr rng(new RandNumGenMT19937(seed));

  sim.setRNG(rng);

  sim.reserveOverLatticeEvents(PSOverLatticeEvents::SIZE);
  sim.addOverLatticeEvent(PSOverLatticeEvents::DEPOSITION,
                          F, DepositionExecute);

  CellNeighOffsets hopCNO(HopOffset::SIZE);

  hopCNO.addOffset(HopOffset::UP,    CellIndsOffset(0,+1,0));
  hopCNO.addOffset(HopOffset::DOWN,  CellIndsOffset(0,-1,0));
  hopCNO.addOffset(HopOffset::LEFT,  CellIndsOffset(-1,0,0));
  hopCNO.addOffset(HopOffset::RIGHT, CellIndsOffset(+1,0,0));

  sim.reserveCellCenteredEventGroups(1, PSCellCenteredEvents::SIZE);

  EventExecutorGroup hopExecs(PSCellCenteredEvents::SIZE);

  std::vector<CellNeighOffsets> tmpExecCNO;
  tmpExecCNO.reserve(1);
  tmpExecCNO.push_back(CellNeighOffsets(2));
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::LEFT));

  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_LEFT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::RIGHT));
  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_RIGHT,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::UP));
  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_UP,
                            HoppingExecute(), tmpExecCNO);

  tmpExecCNO.back().resetOffsets(2);
  tmpExecCNO.back().addOffset(1, hopCNO.getOffset(HopOffset::DOWN));
  hopExecs.addEventExecutor(PSCellCenteredEvents::HOP_DOWN,
                            HoppingExecute(), tmpExecCNO);

  sim.addCellCenteredEventGroup(1, hopCNO, HoppingPropensity(&E_s, E_n, T), hopExecs);


  sim.reserveTimePeriodicActions(PAction::SIZE);
  sim.addTimePeriodicAction(PAction::PRINT,
                            PrintASCII("snapshot"),
                            0.05*approxDepTime, true);

  sim.run(approxDepTime);
  sim.removeOverLatticeEvent(PSOverLatticeEvents::DEPOSITION);
  sim.run(0.1*approxDepTime);

  return 0;
}
```

8.23 testPatternedSurface2/EventsAndActions.hpp

Header file for another example implementation of a model with a patterned substrate surface

This example is discussed in detail in the section “Example: Implementations of a patterned substrate model,” subsection “First implementation.”

```
#ifndef EVENTS_AND_ACTIONS_HPP
#define EVENTS_AND_ACTIONS_HPP

/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
*/

#include <string>

#include <KMCThinFilm/CellCenteredGroupPropensities.hpp>
#include <KMCThinFilm/EventExecutor.hpp>
#include <KMCThinFilm/MakeEnum.hpp>

#include <boost/multi_array.hpp>
```

```
typedef boost::multi_array<double,2> DblArray2D;

#include "PhysicalConstants.hpp"

KMC_MAKE_ID_ENUM(PSOverLatticeEvents,
                 DEPOSITION);

KMC_MAKE_ID_ENUM(PSCellCenteredEvents,
                 HOP_UP,
                 HOP_DOWN,
                 HOP_LEFT,
                 HOP_RIGHT);

KMC_MAKE_ID_ENUM(PAction,
                 PRINT);

KMC_MAKE_LATTICE_INTVAL_ENUM(PS, HEIGHT);
KMC_MAKE_LATTICE_FLOATVAL_ENUM(PS, E_s);

KMC_MAKE_OFFSET_ENUM(HopOffset,
                     UP, DOWN, LEFT, RIGHT);

void DepositionExecute(const KMCThinFilm::CellInds & ci,
                       const KMCThinFilm::SimulationState & simState,
                       KMCThinFilm::Lattice & lattice);

class HoppingPropensity {
public:
  HoppingPropensity(const DblArray2D * E_s, double E_n, double T)
    : E_s_(E_s), E_n_(E_n), kBT_(PhysConst::kB*T), k_(kBT_/PhysConst::h)
  {}
  void operator()(const KMCThinFilm::CellNeighProbe & cnp,
                  std::vector<double> & propensityVec) const;
private:
  const DblArray2D * E_s_;
  double E_n_, kBT_, k_;
};

class HoppingExecute {
public:
  void operator()(const KMCThinFilm::CellInds & ci,
                  const KMCThinFilm::SimulationState & simState,
                  const KMCThinFilm::Lattice & lattice,
                  std::vector<KMCThinFilm::CellsToChange> & ctcVec) const;
};

class PrintASCII {
public:
  PrintASCII(const std::string & fNameRoot)
    : fNameRoot_(fNameRoot),
      snapShotCntr_(0)
  {}

  void operator()(const KMCThinFilm::SimulationState & simState,
                  KMCThinFilm::Lattice & lattice);

private:
  std::string fNameRoot_;
  int snapShotCntr_;
};

#endif /* EVENTS_AND_ACTIONS_HPP */
```

8.24 testPatternedSurface2/EventsAndActions.cpp

Implementation file for another example implementation of a model with a patterned substrate surface

This example is discussed in detail in the section "Example: Implementations of a patterned substrate model," subsection "First implementation."

```
/*
   Comments in this code of the form "//! [...]" are used to assist
   Doxygen in documenting this file.
```

```
*/

#include "EventsAndActions.hpp"

#include <vector>
#include <fstream>
#include <cmath>

#include <boost/lexical_cast.hpp>

#include <KMCThinFilm/ErrorHandling.hpp>

using namespace KMCThinFilm;

void DepositionExecute(const CellInds & ci,
                       const SimulationState & simState,
                       Lattice & lattice) {

  int currVal = lattice.getInt(ci, PSIntVal::HEIGHT);
  lattice.setInt(ci, PSIntVal::HEIGHT, currVal + 1);

}

void HoppingPropensity::operator()(const CellNeighProbe & cnp,
                                   std::vector<double> & propensityVec) const {

  KMCThinFilm::CellToProbe ctpSelf = cnp.getCellToProbe(HopOffset::SELF);

  int currHeight = cnp.getInt(ctpSelf, PSIntVal::HEIGHT);

  if (currHeight > 0) {

    int n = 0;

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::UP), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::DOWN), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::LEFT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    if (cnp.getInt(cnp.getCellToProbe(HopOffset::RIGHT), PSIntVal::HEIGHT) >= currHeight) {
      ++n;
    }

    const CellInds & ciSelf = ctpSelf.inds();

    int coord_i_in_domain = ciSelf.i % (E_s_->shape()[0]);
    int coord_j_in_domain = ciSelf.j % (E_s_->shape()[1]);

    double E = (*E_s_)[coord_i_in_domain][coord_j_in_domain] + n*E_n_;

    double p = k_*std::exp(-E/kBT_);

    for (int i = 0; i < PSCellCenteredEvents::SIZE; ++i) {
      propensityVec[i] = p;
    }

  }
}

void HoppingExecute::operator()(const CellInds & ci,
                                const SimulationState & simState,
                                const Lattice & lattice,
                                std::vector<CellsToChange> & ctcVec) const {

  CellsToChange & ctc = ctcVec[0];
  ctc.setCenter(ci);

  int currFrom = lattice.getInt(ci, PSIntVal::HEIGHT);
  int currTo = ctc.getInt(1, PSIntVal::HEIGHT);

  ctc.setInt(0, PSIntVal::HEIGHT, currFrom - 1);
  ctc.setInt(1, PSIntVal::HEIGHT, currTo + 1);
}

void PrintASCII::operator()(const SimulationState & simState, Lattice & lattice) {

  ++snapShotCntr_;

  std::string fName = fNameRoot_ + boost::lexical_cast<std::string>(snapShotCntr_) + ".dat";

  std::ofstream outFile(fName.c_str());

  LatticePlanarBBox localPlanarBBox;
  lattice.getLocalPlanarBBox(false, localPlanarBBox);

  int iminGlobal = localPlanarBBox.imin;
  int jminGlobal = localPlanarBBox.jmin;

  // "P1" here is short for "Plus 1".
  int imaxP1Global = localPlanarBBox.imaxP1;
  int jmaxP1Global = localPlanarBBox.jmaxP1;

  outFile « "# " « iminGlobal « " " « imaxP1Global « " " « jminGlobal « " " « jmaxP1Global
          « " time:" « simState.elapsedTime() « "\n";

  CellInds ci; ci.k = 0;
  for (ci.i = localPlanarBBox.imin; ci.i < localPlanarBBox.imaxP1; ++(ci.i)) {
    for (ci.j = localPlanarBBox.jmin; ci.j < localPlanarBBox.jmaxP1; ++(ci.j)) {
      outFile « ci.i « " " « ci.j « " "
              « lattice.getInt(ci, PSIntVal::HEIGHT) « "\n";
    }
  }

  outFile.close();
}
```

Bibliography

[1] James L. Blue, Isabel Beichl, and Francis Sullivan. Faster Monte Carlo simulations. *Physical Review E*, 51:R867–R868, February 1995. 6

[2] A. B. Bortz, M. H. Kalos, and J. L. Lebowitz. A new algorithm for Monte Carlo simulation of Ising spin systems. *Journal of Computational Physics*, 17(1):10–18, 1975. 5

[3] J. Cho, S. G. Terry, R. LeSar, and C. G. Levi. A kinetic Monte Carlo simulation of film growth by physical vapor deposition on rotating substrates. *Materials Science and Engineering: A*, 391(1):390–401, 2005. 16

[4] Kristen A. Fichthorn and W. H. Weinberg. Theoretical foundations of dynamical Monte Carlo simulations. *The Journal of Chemical Physics*, 95(2):1090–1096, July 1991. 5

[5] A. Kuronen, L. Nurminen, and K. Kaski. Computer simulation of nucleation on patterned surfaces. *MRS Online Proceedings Library*, 584(1):239–244, December 1999. 33, 37

[6] Paul Meakin and Joachim Krug. Three-dimensional ballistic deposition at oblique incidence. *Physical Review A*, 46:3390–3399, September 1992. 25

[7] John A. Mitchell, Fadi Abdeljawad, Corbett Battaile, Cristina Garcia-Cardona, Elizabeth A. Holm, Eric R. Homer, Jon Madison, Theron M. Rodgers, Aidan P. Thompson, Veena Tikare, Ed Webb, and Steven J. Plimpton. Parallel simulation via SPPARKS of on-lattice kinetic and Metropolis Monte Carlo models for materials processing. *Modelling and Simulation in Materials Science and Engineering*, 31(5):055001, May 2023. 9, 10, 25

[8] T. P. Schulze. Kinetic Monte Carlo simulations with minimal searching. *Physical Review E*, 65:036704, February 2002. 6

[9] Feng Shi, Yunsic Shim, and Jacques G. Amar. Parallel kinetic Monte Carlo simulations of two-dimensional island coarsening. *Physical Review E*, 76:031607, September 2007. 10

[10] Yunsic Shim and Jacques G. Amar. Semirigorous synchronous sublattice algorithm for parallel kinetic Monte Carlo simulations of thin film growth. *Physical Review B*, 71:125432, March 2005. 8, 10, 11

Index