Physics 750: Final Examination

to be submitted by Thursday, December 7 at 3:00 pm

For this final examination, you are to attempt all the steps enumerated below. The work is to be completed out of class time, on your own schedule. Please package your work as a bitbucket repository and send a share invitation to kevin.beach@gmail.com by the due date. You are responsible for providing correct, properly documented C++ code and a makefile. If you feel that additional explanation is required to answer the questions, please include a standalone document; I suggest providing a pdf file, LaTeXsource code, or a simple plain-text README (or README.md file in markdown). For any graphs that are requested, please provide a columnar data file, a gnuplot script, and the output in pdf format. Don't forget to label the axes.

I have confidence that you will conduct yourself with the highest standards of academic honesty. The work you submit should be carried out independently, and I expect that you will not collaborate with other students on your solution. While I understand that you may need to consult C++ language references, I ask you not to search online for other people's solutions to similar computational problems.

1. Use the `curl` command to download from the class website everything you'll need for this examination.

```
$ WEBPATH=http://www.phy.olemiss.edu/~kbeach/
$ curl $WEBPATH/courses/fall2017/phys750/src/exam.tgz -O
$ tar xzf exam.tgz
$ cd exam
```

Part I

In the Deposition, Diffusion, and Aggregation (DDA) model [see P. Jensen *et al.*, Phys. Rev. B **50**, 15316 (1994)], we imagine a steady flux of incoming atoms incident on a flat surface. Because of thermal effects, the atoms randomly diffuse on the surface until they encounter other atoms, which they bond with to form heavy, immobile structures. In other words, adjacent atoms nucleate static islands of atoms.

The behaviour of the system depends on the deposition rate and the rate of diffusion on the surface. When the incident flux is low and the surface diffusion high, each deposited atom travels a long distance before it encounters another atom. In the opposite limit, each atoms travels only a short distance before it bonds to another atom and stops moving. The two regimes lead to very different patterns of nucleation.

2. The program `diffusion.cpp` (naively) implements the DDA model. Look at the source code and figure out how it works. Explore its behaviour over the range of input arguments.

```
$ cd part1
$ make diffusion
g++ -o diffusion diffusion.cpp -O2 -Wall -ansi -pedantic ...
$ ./diffusion
diffusion (0.0 < initial population probability < 1.0) (0.0 < deposition rate)
$ ./diffusion 0.0 2.5
```

3. Develop a stopping condition for the algorithm.

4. Compute the effective dimension—possibly fractal—of the resulting configuration in the limit of slow and fast deposition, in both cases starting with no initial population on the surface.

5. Check out how slowly the program runs when the linear dimension of the lattice is increased:

```
const size_t L = 100; // change the value on line 26 to 200 or 400, say
```

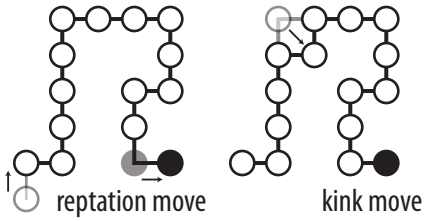
The problem is that the implementation is poor. Modify the code to get a comparable efficiency gain as seen in `perc_cluster2.cpp` with respect to `perc_cluster.cpp` in Exercise 4.

6. Using the `bitmap.hpp` header from Exercise 1, create a new version of the program that dumps its final configuration to `output.png` rather than to the screen via OpenGL. You'll have to modify the `makefile` accordingly.

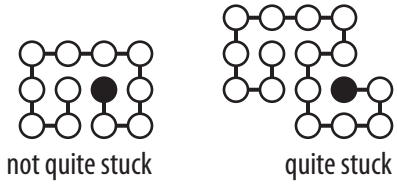
Part II

Enumeration techniques can be used to compute the properties of the self-avoiding walk (SAW) for walk lengths only up to $N \approx 20$. For as N gets large, the number of possible configurations grows exponentially and exhaustive counting eventually becomes unfeasible. In situations such as this, where the sheer size of the phase space is an obstacle, progress can often be made using Monte Carlo methods. That's the plan here. You will simulate very long SAWs by stochastically sampling some representative fraction of the total number of configurations.

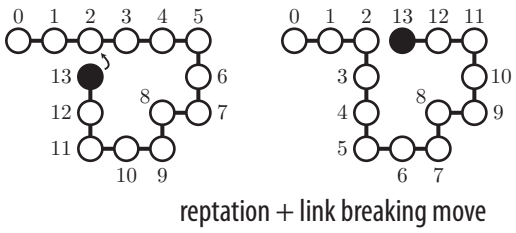
Our previous discussions have focused on the step-by-step process ($x_N \to x_{N+1}$) of growing a random walk. We now take a different point of view and consider all possible walks of fixed length. The trick is to find a set of Monte Carlo updates or *moves* that sample the entire configuration space in an unbiased way. (To make the connection to physics more concrete, we will use the polymer vocabulary: a walk of N steps corresponds to a chain of $N + 1$ monomers, each of which is chemically bonded to two neighbours.) Two possibilities are the *reptation* and *kink* moves illustrated below.



Repation causes the polymer to progress like a slithering snake by removing one monomer from the tail and adding another to the head. It's important here that the head advance with equal probability in one of the three non-backtracking directions. If the move is forbidden by self-avoidance, it is rejected. The kink move introduces a wiggle in the chain by attempting to move a monomer with 90° bonds diagonally across the square plaquet. If the destination site is occupied, the move is rejected. Note that these two moves in combination are not ergodic. Some points of the phase space are disconnected and can never be reached.



Consider another move that is reptation-like, but modified so that on self-intersection an adjacent link is removed in compensation; that link is chosen in such a way that the monomers continue to form a single, contiguous chain. This motion is sometimes called a *worm* update.



Suppose that the 14 monomers in the example above are stored in an array with indices $0, 1, \ldots, 13$ with the understanding that sequentially ordered monomers are linked. Then, the move shown, in which the monomer labelled 13 attempts to advance on the site occupied by monomer 2, amounts to the swapping of monomers in the array: $3 \leftrightarrow 13$, $4 \leftrightarrow 12$, $\ldots$, $7 \leftrightarrow 9$. This move is ergodic, and no part of the phase space is unreachable. It's also very efficient, since it's never rejected. As you will discover, however, the worm move is not unbiased.

7. `cd` into the `part2` directory and compile the provided code by invoking `make`. The programs `MonteCarlo` and `MonteCarlo_openGL` simulate an ordinary random walk (ORW) via reptation; the latter provides graphical output. Both programs take a single command line argument representing the length of the walk. For example, the following simulates an ORW of length 300.

   ```
   $ cd ../part2
   $ make
   $ ./MonteCarlo_openGL 300
   ```

   The nongraphical version computes the squared displacement (end-to-end distance of the polymer), bins these measurements, and writes the completed bins to a file `r2.bins.dat` in three-column format: i N $(r_N^2)_i$, where $i = 1, 2, \ldots$ is the bin number and N is the number of walk steps. The flag `std::ios::app` ensures that the file is never overwritten. New data is simply appended to the existing file each time the program is run.

   Write a program `gather` that accumulates the various measurements and organizes them in three-column format: N $\langle r_N^2 \rangle$ ϵ, where $\epsilon = \sqrt{\mathrm{var}(r^2)/N_{\mathrm{bins}}}$. Recall that the mean and variance are given by

   $$\langle A \rangle = \frac{1}{N_{\mathrm{bins}}} \sum_{i=1}^{N_{\mathrm{bins}}} A_i, \quad \mathrm{var}(A) = \frac{1}{N_{\mathrm{bins}}} \sum_{i=1}^{N_{\mathrm{bins}}} (A_i - \langle A \rangle)^2 = \langle A^2 \rangle - \langle A \rangle^2.$$

   Run the Monte Carlo for $N = 3, 4, \ldots, 20$ (using the `runsA.bash` script provided). Compare the results to the $\langle r_N^2 \rangle = N$ diffusion law. If you write `gather` so that it reads from `cin` and writes to `cout`, then you can use redirection as follows.

   ```
   $ ./runsA.bash
   $ ./gather < r2.bins.dat > r2.orw.dat
   $ gnuplot
   gnuplot> plot "r2.orw.dat" using 1:2:3 with errorbars
   ```

8. The header file `polymer.hpp` defines `monomer` to be a coordinate pair and defines `polymer` to be a `deque` (double-ended queue) of `monomers`. Add methods to the `polymer` class that implement forward and backward reptation. One case amounts to pushing a new monomer onto the front (`push_front`) and popping an old one off the back (`pop_back`). The other is just the reverse. You should also add a method for the kink move.

   The prototypes for these methods should be as follows.

   ```
   template <typename Func>
   bool reptation_saw_forward(Func &Rand)
   template <typename Func>
   bool reptation_saw_backward(Func &Rand)
   template <typename Func>
   void kink_saw(Func &Rand)
   ```

The template is there so that a function-object random number generator can be passed to the method. The return value should be `true` if the reptation move is accepted and `false` if it is rejected (when the excluded volume constraint is violated). The kink move needn't return anything. Now, in the file `routines.cpp`, comment out the code block marked `question 7`, uncomment the block marked `question 8`, and recompile.

What does this `question 8` code block actually do? Also, how do the simulated values for $N = 3, 4, \ldots, 20$ compare with the exact results from enumeration?

```
$ ./runsA.bash
$ ./gather < r2.bins.dat > r2.saw.dat
$ gnuplot
gnuplot> plot "r2.saw.dat" using 1:2:3 with errorbars, \
              "exact.saw.dat" using 1:3 with points
```

9. Add a method to `polymer.hpp` that implements the worm move. Use the following function protoype.

   ```
   template <typename Func>
   void worm_saw(Func &Rand)
   ```

   In the file `routines.cpp`, comment out the code block marked `question 8` and uncomment the block marked `question 9`. Compile and run the modified code. How do the simulated values for $N = 3, 4, \ldots, 20$ compare with the exact results from enumeration and with those from the reptation-and-kink simulation in the previous question?

10. Using the Monte Carlo methods described in questions 8 and 9, compute the mean squared displacement for polymer chains of length $N = 20, 40, 80, \ldots, 1280$. You can automate this with the `runsB.bash` script. In each case, fit the data to the form $\langle r_N^2 \rangle \sim N^{2\nu}$ and extract a value for the exponent ν.