Monte Carlo Simulation of Crystal Growth

Kevin Juan (kj89)

November 14th 2018

The Numerical Methods and Algorithms

The objective of this assignment was to implement Monte Carlo methods to simulate the growth of a crystal lattice. Broadly speaking, Monte Carlo methods are any methods that use a random number generator. Monte Carlo methods are well-known for being useful for high dimensional integration problems where other methods like the trapezoid method perform worse.

In this assignment, we use pseudo-random number generators to predict the crystal growth by randomly controlling atom deposition or diffusion. Pseudo-RNGs start with a seed, and create a reproducible sequence of random numbers. Starting with a 32-bit (or 64-bit) integer seed, x_0, the sequence of random numbers $x_1, x_2, ..., x_\infty$ is given by the following equation:

$$x_{i+1} = (ax_i + c) \bmod m \tag{1}$$

The numbers a, b, and c are the so-called magic numbers, and arbitrary values for those numbers do not work. For this assignment, the method `Ranq1.doub()` from Press et al[1] was used.

Some properties of the sequence are that each starting seed yields a different sequence, the same seed yields the same sequence, and the random numbers are uniformly distributed between 0 and $m-1$. The particular RNG used in this assignment produces numbers between 0 and 1. More on RNGs can be found in §7.0-7.3 in Press et al[1]. Monte Carlo methods can be read about further in chapter 5 in Landau et al[2]. More on Monte Carlo integration can be found in §7.7 in Press et al[1] or §6.5-6.9 in Landau et al[2].

Crystal Growth in 3D

Monte Carlo methods and RNGs were used to predict crystal growth on a surface that had a pre-existing crystal surface deposited on top by vapor deposition. For this application,

we started with a 2D crystal surface that was $N_x \times N_y$ in dimension where $N_x = 50$ and $N_y = 25$. A 3D lattice array, $L(x, y, z)$ can be instantiated with dimensions N_x, N_y, and N_z where $N_z = 25$. This array stores integer values of either 1 or 0 where 1 indicates a coordinate occupied by an atom and 0 indicates an unoccupied space. For large lattices, `char` values for 1 and 0 may be more appropriate to save memory. We note that the x and y axes are periodic while the z axis is not.

We initialized the surface $z = 0$ to be completely filled to represent the pre-existing surface. For every time step, we identify perimeter sites, which are sites that are unoccupied, but at least one of its 26 nearest neighbors in the lattice is occupied. The coordinates of the perimeter sites can be stored in arrays to be accessed later. Using an RNG, we can determine if this time step will result in deposition of a new atom or diffusion of a pre-existing atom in the lattice to a new nearby vacant point. We thus want to check the condition $P_{Dep} > R$ to determine if we will deposit or diffuse. In this case, P_{Dep} is the probability of diffusion, which was set to 0.5, and R is a randomly generated number between 0 and 1. If this step is a deposition step, we randomly choose a perimeter site from the arrays to deposit an atom into. We then check its probability of sticking using the equation below:

$$P_{Total} = \sum_{i,j,k=-1}^{+1} L(x+i, y+j, z+k) \frac{p_x}{P_{Max}\sqrt{i^2+j^2+k^2}} \tag{2}$$

If we are to deposit, $p_x = p_0 = 0.01$ for sticking. P_{Max} is the value when all neighbors are occupied, which is 19.1041 and helps to normalize the probability. If we are to diffuse an atom, we choose one perimeter site at random and one of its nearest neighbors at random. If the neighbor is occupied, we move the atom to the perimeter site if it has a higher P_{Total} or if $P_{Total} > R$ where R is yet another random number between 0 and 1. It's important to note that the initial surface cannot be moved. This is repeated until enough atoms have been deposited, which in this application is 10 monolayers of $N_x \times N_y \times 10$ atoms.

At each time step, we also calculate the average height and roughness (width) or the surface. The height is calculated as follows:

$$h = \frac{1}{N_x N_y} \sum_{xy} h_{x,y} \tag{3}$$

$$w^2 = \frac{1}{N_x N_y} \sum_{xy} (h_{x,y} - h)^2 = \frac{1}{N_x N_y} \sum_{xy} h_{x,y}^2 - \left(\frac{1}{N_x N_y} \sum_{xy} h_{x,y}^2 \right) \tag{4}$$

We see that the width is just the standard deviation of the height

Implementation and Results

Using the RNG `Ranq1.doub()` with a seed of `1`, tests were done to ensure that it properly worked. A histogram was made to show the distribution of random numbers.

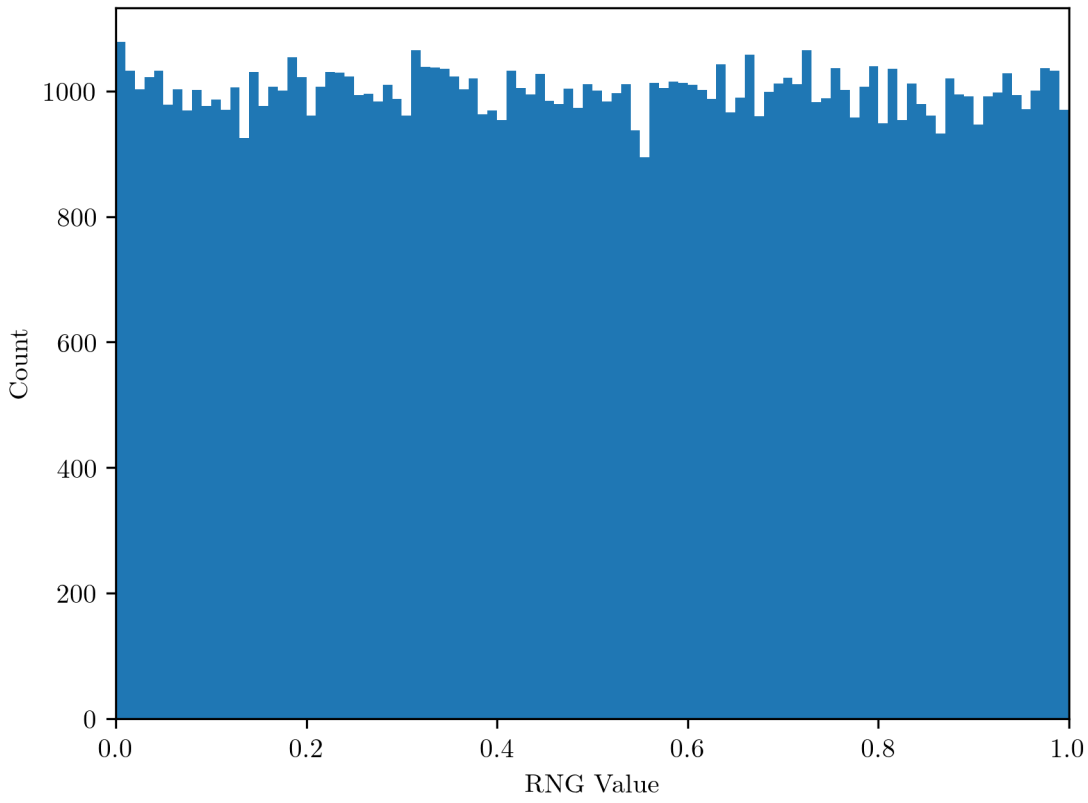

Figure 1: Histogram of random numbers produced by `Ranq1.doub()` using seed `1`.

As expected, the histogram shows a roughly uniform distribution. A plot of x_{i+1} vs. x_i was also produced to show the distribution of random pairs as small dots.

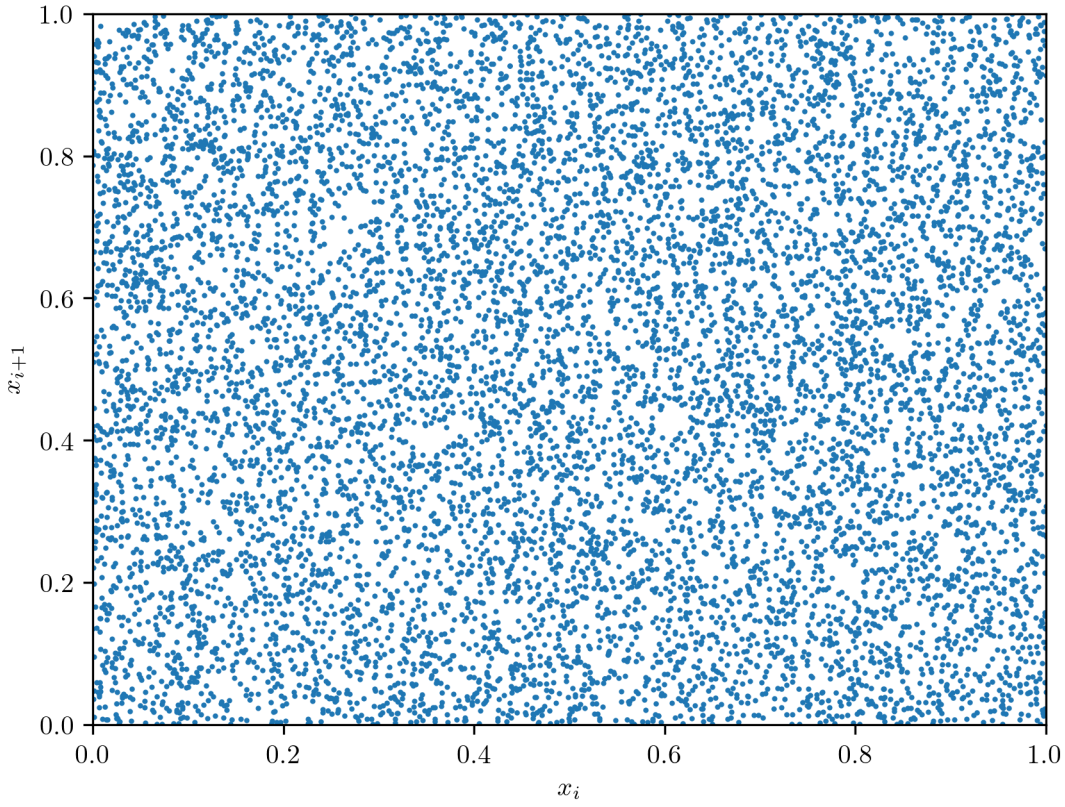

Figure 2: Plot of x_{i+1} vs. x_i produced by `Ranq1.doub()` using seed `1`.

As expected, the distribution of pairs shows no glaring trends or patterns of bias.

A program was then written to simulate the deposition and growth of a crystal using the tested RNG. The following plots show a 3D image of the crystal structure with spheres resembling atoms and a cross-sectional view of the crystal taken at the middle y coordinate with circles resembling atoms.

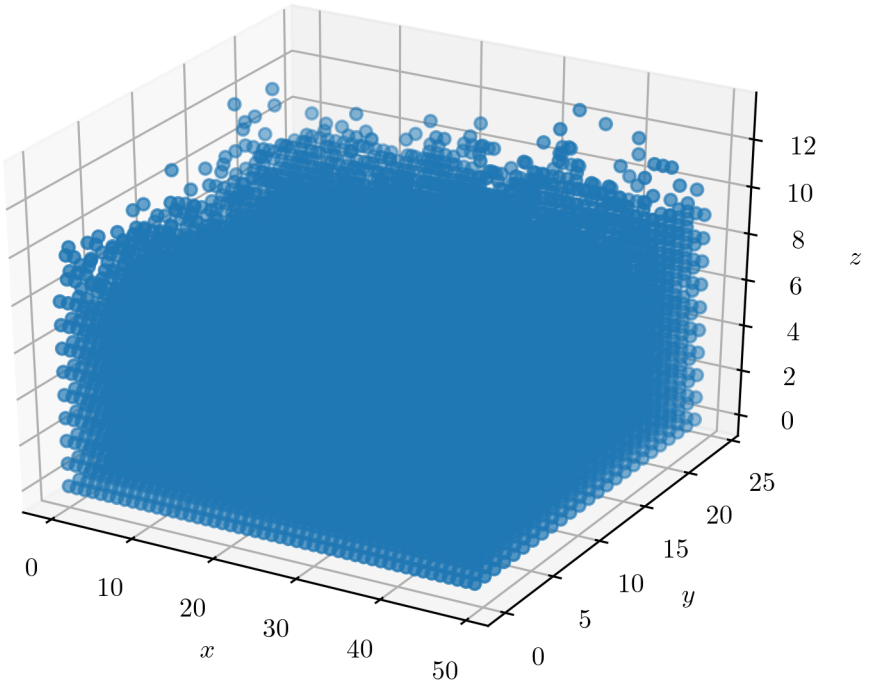

Figure 3: 3D image of the crystal structure.

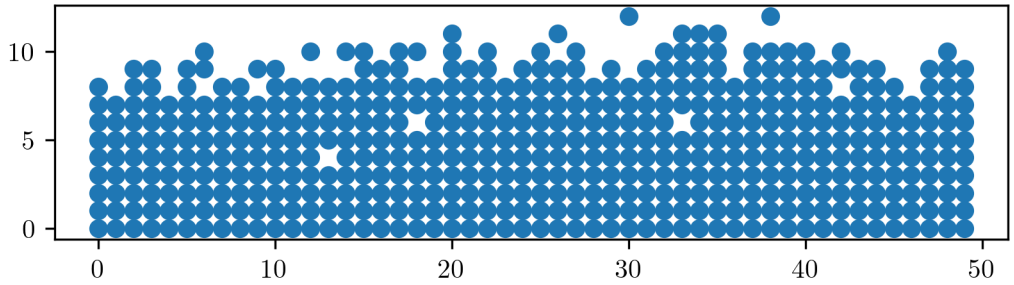

Figure 4: Cross-sectional view of the crystal at the middle y-value.

At the end of each time step, a calculation was done for the average height and the crystal width. A plot showing how these two values changed throughout the simulation is shown below:

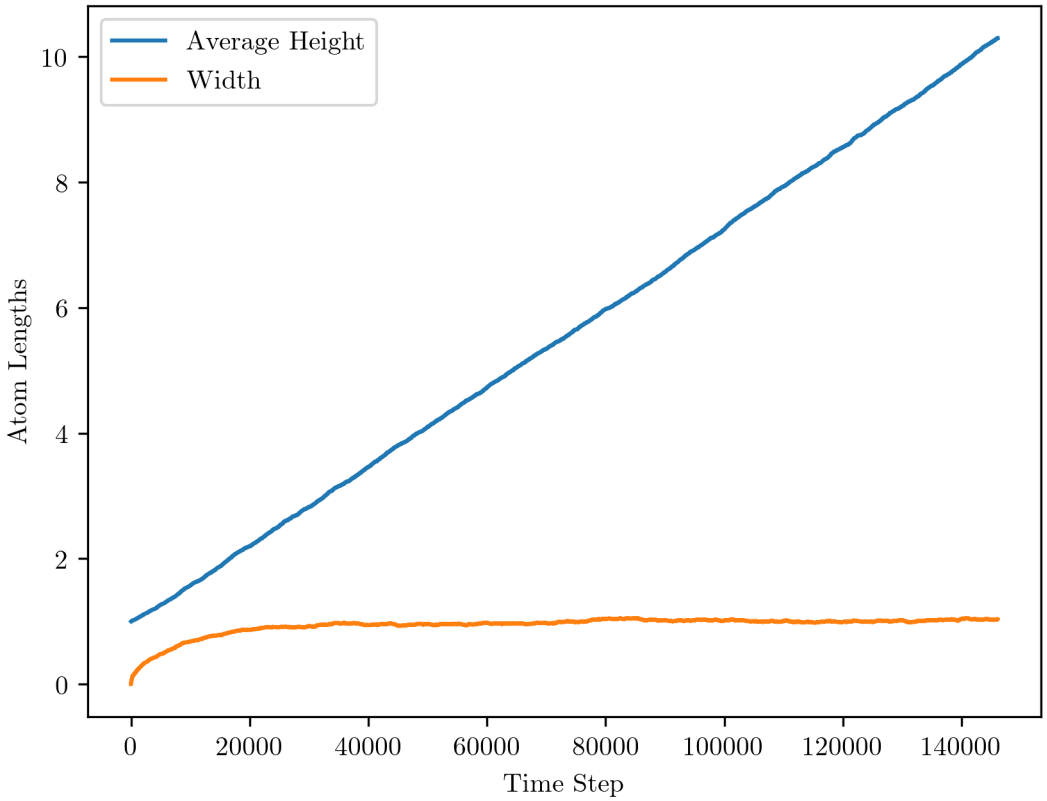

Figure 5: Plot showing how the average height and width of the crystal changed throughout the simulation.

Source Code

```
/* AEP 4380 Homework #9

Monte Carlo method tested on crystal growth

Run on a core i7 using clang 1000.11.45.2 on macOS Mojave

Kevin Juan 14 November 2018
*/

#include "nr3.h"
#include "ran.h"
#include <cmath>
```

```
#include <cstdlib>
#include <ctime>
#include <fstream>  //  stream file IO
#include <iomanip>  //  to format the output
#include <iostream> //  stream IO
#define ARRAYT_BOUNDS_CHECK
#include "bigarrayt.hpp"

using namespace std;

double height(arrayt<int> &L) {
    double h = 0;
    int i, j, k, Nx = L.n1(), Ny = L.n2(), Nz = L.n3();
    int lastZ = 1;

    for (i = 0; i < Nx; i++) {
        for (j = 0; j < Ny; j++) {
            for (k = 0; k < Nz; k++) {
                if (L(i, j, k) == 1 && k + 1 > lastZ) {
                    lastZ = k + 1;
                }
            }
            h += (double)lastZ;
            lastZ = 1;
        }
    }
    return h / (Nx * Ny);
}

double width(arrayt<int> &L, double h) {
    double wSq, hSq = 0;
    int i, j, k, Nx = L.n1(), Ny = L.n2(), Nz = L.n3();
    int lastZ = 1;

    for (i = 0; i < Nx; i++) {
        for (j = 0; j < Ny; j++) {
            for (k = 0; k < Nz; k++) {
                if (L(i, j, k) == 1 && k + 1 > lastZ) {
                    lastZ = k + 1;
                }
            }
            hSq += (double)(lastZ * lastZ);
            lastZ = 1;
        }
    }
```

```
    hSq /= (double)(Nx * Ny);
    wSq = hSq - h * h;
    return sqrt(wSq);
}

inline int periodic(int i, int n) { return (i + n) % n; }

double PTotal(arrayt<int> &L, bool dep, int x, int y, int z) {
    const static double PMax = 19.1041;
    int i, j, k, ix, jy;
    double PTotal = 0.0, px, p0 = 0.5, p1 = 0.5;

    if (dep == true) {
        px = p0;
    } else {
        px = p1;
    }
    for (i = -1; i <= 1; i++) {
        ix = periodic(x + i, L.n1());
        for (j = -1; j <= 1; j++) {
            jy = periodic(y + j, L.n2());
            for (k = -1; k <= 1; k++) {
                if (i != 0 || j != 0 || k != 0) {
                    PTotal += L(ix, jy, z + k) * px /
                              (PMax * sqrt(i * i + j * j + k * k));
                }
            }
        }
    }
    return PTotal;
}

// return true if point is a perimeter, false otherwise
bool isPerimeter(arrayt<int> &L, int x, int y, int z) {
    int i, j, k, ix, jy;
    bool perimeter = false;

    if (L(x, y, z) == 0) {
        for (i = -1; i <= 1; i += 2) {
            ix = periodic(x + i, L.n1());
            for (j = -1; j <= 1; j += 2) {
                jy = periodic(y + j, L.n2());
                for (k = -1; k <= 1; k += 2) {
                    if (L(ix, jy, z + k) == 1) {
                        perimeter = true;
```

```
                    }
                }
            }
        }
    }
    return perimeter;
}

// return 1 or -1 to get random neighbor for x, y, or z
int randNN(double randVal) {
    if (randVal > 0.5) {
        return 1;
    } else {
        return -1;
    }
}

int main() {
    unsigned int seed1, seed2;
    const static int Nx = 50, Ny = 25, Nz = 25, Na = 10;
    const static double PDep = 0.5;
    bool dep;
    double h, w;
    int i, j, k, tIter = 0, numDep = Nx * Ny, count;
    arrayt<int> x(Nx * Ny * Nz); // initialize array for x perimeters
    arrayt<int> y(Nx * Ny * Nz); // initialize array for y perimeters
    arrayt<int> z(Nx * Ny * Nz); // initialize array for z perimeters

    seed1 = 1;
    seed2 = time(NULL);
    struct Ranq1 randNum1(seed1);
    struct Ranq1 randNum2(seed2);

    // output RNG data
    ofstream fpRNG;
    fpRNG.open("RNG_data.dat");
    if (fpRNG.fail()) {
        cout << "cannot open file" << endl;
        return (EXIT_SUCCESS);
    }
    // initialize seed
    i = 0;
    // generate new random number from different seed
    while (i < 100000) {
        fpRNG << randNum1.doub() << endl;
```

```
    i++;
}

// initialize lattice points
arrayt<int> L(Nx, Ny, Nz);
for (i = 0; i < Nx; i++) {
    for (j = 0; j < Ny; j++) {
        for (k = 0; k < Nz; k++) {
            if (k == 0) {
                L(i, j, k) = 1;
            } else {
                L(i, j, k) = 0;
            }
        }
    }
}

// output height and width data
ofstream fpHW;
fpHW.open("heightwidth_data.dat");
if (fpHW.fail()) {
    cout << "cannot open file" << endl;
    return (EXIT_SUCCESS);
}

// initial height and width
h = height(L);
w = width(L, h);
fpHW << setw(10) << tIter << setw(10) << h << setw(10) << w << endl;

while (numDep <= Nx * Ny * Na) {
    double R1 = randNum2.doub();
    double R2 = randNum2.doub();
    double R3 = randNum2.doub();
    double randVal = randNum2.doub();
    double xRand = randNum2.doub(); // random number for x neighbor
    double yRand = randNum2.doub(); // random number for y neighbor
    double zRand = randNum2.doub(); // random number for z neighbor
    int xVal, yVal, zVal, xInc, yInc, zInc;
    int xNbr, yNbr, zNbr;

    count = 0;
    // determine perimeter sites
    for (i = 0; i < Nx; i++) {
        for (j = 0; j < Ny; j++) {
```

```
        for (k = 1; k < Nz - 1; k++) {
            if (isPerimeter(L, i, j, k) == true) {
                x(count) = i;
                y(count) = j;
                z(count) = k;
                count++;
            }
        }
    }
}

// randomize deposition or diffusion
if (R1 < PDep) {
    dep = true;
} else {
    dep = false;
}
int position = randVal * count;
xVal = x(position);
yVal = y(position);
zVal = z(position);
xInc = randNN(xRand);
yInc = randNN(yRand);
zInc = randNN(zRand);
double perimPTotal = PTotal(L, dep, xVal, yVal, zVal);

// deposition
if (dep == true) {
    if (perimPTotal > R2) {
        L(xVal, yVal, zVal) = 1;
        numDep++;
    }
} else {
    xNbr = periodic(xVal + xInc, Nx);
    yNbr = periodic(yVal + yInc, Ny);
    zNbr = zVal + zInc;

    // diffusion
    if (L(xNbr, yNbr, zNbr) == 1 && zNbr != 0) {
        double neighborPTotal = PTotal(L, dep, xNbr, yNbr, zNbr);
        if (neighborPTotal < perimPTotal || perimPTotal > R3) {
            L(xNbr, yNbr, zNbr) = 0;
            L(xVal, yVal, zVal) = 1;

            // move back if PTotal is 0 at new site
```

```
                    if (PTotal(L, dep, xVal, yVal, zVal) == 0.0) {
                        L(xNbr, yNbr, zNbr) = 1;
                        L(xVal, yVal, zVal) = 0;
                    }
                }
            }
        }
        h = height(L);
        w = width(L, h);
        tIter++;
        fpHW << setw(10) << tIter << setw(10) << h << setw(10) << w << endl;
    }
    fpHW.close();

    // output crystal data
    ofstream fp;
    fp.open("crystal_data.dat");
    if (fp.fail()) {
        cout << "cannot open file" << endl;
        return (EXIT_SUCCESS);
    }
    for (i = 0; i < Nx; i++) {
        for (j = 0; j < Ny; j++) {
            for (k = 0; k < Nz; k++) {
                if (L(i, j, k) == 1) {
                    fp << setw(5) << i << setw(5) << j << setw(5) << k << endl;
                }
            }
        }
    }
    fp.close();
}
```

References

[1] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery, editors. *Numerical Recipes: The Art of Scientific Computing*. Cambridge University Press, Cambridge, UK ; New York, 3rd ed edition, 2007. OCLC: ocn123285342.

[2] Rubin H. Landau, Jose Paez, and Christian C. Bordeianu. *A Survey of Computational Physics: Introductory Computational Science*. Princeton University Press, Princeton, 2008.