# The Metropolis Algorithm and Nanometer-Scale Pattern Formation

New Mexico
Supercomputing Challenge
Final Report
April 7, 2010

Team 5
Albuquerque Academy

Team Members:
Michael Wang
Jack Ingalls

Teacher:
Jim Mims

Project Mentor:
David Dunlap, Ph.D

# Table of Contents

Executive Summary Page 2

A Brief Summary of the Past Project Page 3

Mathematical Model Page 5

Boltzmann's Distribution Page 7

Simple Two Energy System Page 7

*N* Energy System Page 8

The Metropolis Algorithm Page 10

Algorithm Page 10

Entropy Page 10

Verifying the Metropolis Algorithm for a Two Energy System Page 11

Energies Used in Our System Page 12

Architecture of the Program Page 14

Results and Discussion Page 15

Future Plans Page 17

Acknowledgements Page 19

References Page 20

Appendices Page 21

Appendix A: Parallel Metropolis Algorithm Page 21

Appendix B: Screenshots of program Page 22

## Executive Summary

When chemicals are layered on a surface, they begin to form patterns in order to reduce energy. This phenomenon is known as nanometer-scale pattern formation. This phenomenon plays a huge role in nanotechnology. If understood completely, it can be used to create devices at the nanometer scale. In addition, it would allow for cheap mass production.

In our previous project, we wrote a program that could simulate these patterns. However, we ran into several problems. We had to solve a set of differential equations that described the pattern formation process. Solving the equations required intense calculations, which slowed down our program significantly. In many cases, simulations had to be done overnight to obtain any useful results. In addition, we noticed that somehow mass was not conserved. We believe that the main culprit is numeric error.

In light of these issues, we seek to find a new way to simulate this phenomenon. In this project, we present a Monte Carlo method known as the Metropolis Algorithm that has successfully simulated the patterning phenomenon. This method of solving the problem provides us, and hopefully future users, with a short simulation time and a great amount of flexibility to allow us to study systems under a wide variety of conditions. In the previous project, extending the project to include many conditions might prove to be impossible, as one might not be able to derive the equations. On the other hand, with this project, including many conditions doesn't require more than a simple change in the code, thus making this program far more flexible.

## A Brief Summary of the Past Project

When deposited on a solid surface (substrate), some chemicals rearrange to form patterns. The main factor that drives this pattern formation is free energy. In order to reach any sort equilibrium, a system will try to minimize its total free energy. In the case of pattern formation, the system will separate into multiple phases, that is, multiple regions of different concentration. Each concentration corresponds to a minimum (a trough) in the free energy function as shown in the following figures (Suo and Lu, Forces that drive nanoscale self-assembly on solid surfaces, 2000).

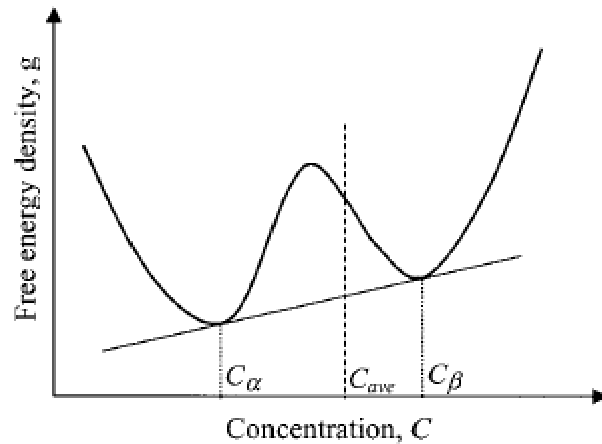



**Figure 1: This is the free energy as a function of concentration. $C_\alpha$ and $C_\beta$ correspond to the phases (below) α and β, respectively.**

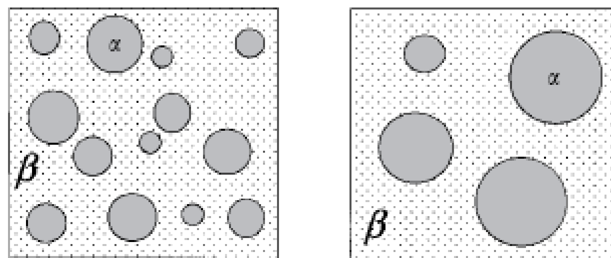



**Figure 2: Regions *α* and *β* are the two phases the system separates into.**

There are actually two other factors: surface strain energy and interface energy. These factors however, do not affect the separation process. Instead, they are involved in size selection. Surface strain energy is the energy due to an elastic deformation in the substrate, which occurs when there is a heterogeneous pattern. This is in some way analogous to having blocks of different masses resting on a bed of springs. The heaver blocks will tend to compress the strings more than the lighter ones. So, different concentrations will have different effects on the substrate. Interface energy, also known as phase boundary energy, is the energy associated with a difference in concentration, or the chemical gradient. It is like saying that a particle with zero neighbors will have an energy (likely higher) different from that of a particle with more than zero neighbors. So, the longer a boundary is, the more interface energy there will be.

Now, how do these affect size selection? It should be clear that interface energy increases with the sum of the lengths of all the boundaries. Thus the system will try to reduce the total boundary length. It turns out that the way to do this is by clumping like regions together. In Figure 2, we see that the smaller regions group together into larger regions. Will this continue until all that is left is one large region? The answer would be yes if surface strain energy were not included. As the size of each region increases, the larger the "heavier blocks" become and the more deformed the substrate becomes. To reduce the strain energy, the size of the regions must decrease. As you can probably guess, these two factors will eventually balance and the system will settle down in equilibrium.

Mathematical Model

A diffusion equation has been developed by Lu and Kim to describe the process above.

$$\frac{\partial C}{\partial t} = \nabla^2\left(\ln\frac{C}{1-C} + \Omega(1-2C) - 2\nabla^2 C - \frac{Q}{\pi}\iint \frac{(x_1-\xi_1)\frac{\partial C}{\partial \xi_1} + (x_2-\xi_2)\frac{\partial C}{\partial \xi_2}}{\left((x_1-\xi_1)^2 + (x_1-\xi_1)^2\right)^{3/2}} d\xi_1 d\xi_2\right)$$

This equation is solved by using the Fourier Transform and a semi-implicit difference method. The equation simplifies greatly to $\frac{\partial C}{\partial t} = -k^2\hat{P} - 2\left(k^4 - k^3Q\right)\hat{C}$, where $\hat{P}$ is the Fourier Transform of $\ln\frac{C}{1-C} + \Omega(1-2C)$. Applying the semi-implicit method, we end up with $\hat{C}_{n+1} = \frac{\hat{C}_n - k^2\hat{P}_n\Delta t}{1 + 2\left(k^4 - k^3Q\right)\Delta t}$. Because we can't intuitively "see" Fourier space, we must somehow transform this equation back into real space. This is done with the Fast Fourier Transform (FFT). Also, there are no known transforms for *P*. Therefore, we must calculate *P* in real space and then transform it to Fourier space.

This procedure produces interesting patterns as well as helping us understand the mysterious nanoworld. However, there are two issues that we must address. The first issue deals with time. This procedure is very computationally intensive due to calling the FFT multiple times every time step. In order to get interesting and useful results, we sometimes are required to run the simulation for several hours. Though this isn't an especially long period of time, it can still pose problems.

The second issue deals with stability. It is clear from the numeric methods that errors can build up over time. We have noticed that mass is not conserved. Mass seems to appear out of nowhere. We believe that there are two possible causes for this: numeric

error or just something that wasn't taken into account in the equations. Whatever the cause, the simulations cannot give any useful results if mass is not conserved.

In light of these issues, we sought another simulation method, in particular, a Monte Carlo method. Professor David Dunlap suggested to us the Metropolis-Hastings Sampling Algorithm.

## Boltzmann's Distribution

The Boltzmann's distribution describes the probability that a system is in some particular state. For example, Boltzmann's distribution is frequently used to describe the distribution of velocities of particles in a gas (also known as the Maxwell-Boltzmann's distribution). In a more general case (not just kinetic energy), the Boltzmann's distribution either describes the probability that a particles has a specific energy or roughly how many particles have one specific energy. In this section, we show some brief simplified derivations of Boltzmann's distribution in a two energy system and a *n* energy system.

Deriving Boltzmann's Distribution for a Simple Two Energy System

Let $E_1$ and $E_2$ be the two possible energies a particle can have. Suppose our system is made up of $N$ particles and $M$ lattice sites on a grid. $M_1$ of the sites have energy $E_1$ while $M_2$ have energy $E_2$. We want to determine how many particles have energy particles have energy $E_1$ and how many, energy $E_2$. We assume that there are $N_1$ and $N_2$ particles having energies $E_1$ and $E_2$, respectively. To find $N_1$ and $N_2$, will minimize the free energy of the system. The free energy $F$ is defined as $E - kT \ln W$, where $E$ is the energy, $k$ is Boltzmann's constant ($1.3806505 \times 10^{-23} \, J/K$), $T$ is the temperature, and $W$ is the number of possible microstates or configurations. The number of microstates is the number of ways $N_1$ particles can be placed into $M_1$ locations times the number of ways $N_2$ particles can be placed into $M_2$ locations, or $W = \binom{M_1}{N_1}\binom{M_2}{N_2}$. Substituting this in with

$E = E_1N_1 + E_2N_2$, we have $F = E_1N_1 + E_2N_2 - kT\ln\left(\frac{M_1!}{N_1!(M_1 - N_1)!}\frac{M_2!}{N_2!(M_2 - N_2)!}\right)$ .

We now wish to minimize *F*.

Rewriting using logarithm rules, we obtain

$$F = E_1N_1 + E_2N_2 - kT\left(\ln(M_1!) - \left(\ln(N_1!) + \ln((M_1 - N_1)!)\right) + \ln(M_2!) - \left(\ln(N_2!) + \ln((M_2 - N_2)!)\right)\right)$$
.

Using Stirling's approximation, $\ln(K!) \approx K\ln K - K$ for large *K*,

$$F = E_1N_1 + E_2N_2 - kT\big(M_1\ln M_1 - M_1 - \left(N_1\ln N_1 - N_1 + (M_1 - N_1)\ln(M_1 - N_1) - (M_1 - N_1)\right) + M_2\ln M_2 - M_2 - \left(N_2\ln N_2 - N_2 + (M_2 - N_2)\ln(M_2 - N_2) - (M_2 - N_2)\right)\big)$$

Now we can minimize *F* using $N_2 = N - N_1$.

$$\frac{dF}{dN_1} = E_1 - E_2 - kT\left(\ln(M_1 - N_1) - \ln N_1 - \ln(M_2 - N_2) + \ln N_2\right),\text{ or}$$

$$\frac{dF}{dN_1} = E_1 - E_2 - kT\ln\frac{(M_1 - N_1)N_2}{(M_2 - N_2)N_1} = 0.$$

Knowing that $N = N_1 + N_2$, we can readily solve for $N_1$ and $N_2$.

<u>Deriving Boltzmann's Distribution for a *n* Energy System</u>

This is almost identical to the two energy system, only now we have *n* different energies. *F* now is defined as $F = \sum_{i=0}^{n} E_iN_i - kT\ln\prod_{i=0}^{n}\binom{M_i}{N_i}$, or equivalently

$$F = \sum_{i=0}^{n} E_iN_i - kT\sum_{i=0}^{n}\ln\frac{M_i!}{N_i!(M_i - N_i)!}.$$

Simplifying and using Stirling's approximation, we have

$$F = \sum_{i=0}^{n} E_iN_i - kT\sum_{i=0}^{n}\left(M_i\ln M_i - M_i - \left(N_i\ln N_i - N_i + (M_i - N_i)\ln(M_i - N_i) - (M_i - N_i)\right)\right).$$

Using Lagrange multipliers with the constraint $\sum N_i = N$, we find that $\frac{\partial F}{\partial N_i} = \frac{\partial F}{\partial N_j}$ for all $i$ and $j$. Therefore, knowing $\frac{\partial F}{\partial N_i} = E_i - kT \ln \frac{M_i - N_i}{N_i}$, we obtain

$$E_i - kT \ln \frac{M_i - N_i}{N_i} = E_j - kT \ln \frac{M_j - N_j}{N_j}.$$

Assuming the system is "large" (i.e. $M_i >> N_i$) and using the same approximation, we have $N_j = \frac{M_j}{M_i} N_i e^{\frac{E_i - E_j}{kT}}$. Summing over all $j$, we get $\sum_{j=0}^{n} N_j = N = \sum_{j=0}^{n} \frac{M_j}{M_i} N_i e^{\frac{E_i - E_j}{kT}}$. So,

$N_i = \frac{N}{\sum_{j=0}^{n} \frac{M_j}{M_i} e^{\frac{E_i - E_j}{kT}}}$. However, if we do not assume that $M_i >> N_i$, then the equation becomes significantly harder to solve. We will not show it here.

## Metropolis Algorithm

Boltzmann's distribution can be useful when figuring how particles are distributed. Other examples include the distribution of velocity of gas particles (just let $E$ be kinetic energy). Analytically, the above process is how we would derive the distribution. How would we do it computationally? This is where the Metropolis Algorithm comes in (there are other ways). In the Metropolis Algorithm, we essentially check whether or not a particle will move based on the energy change of that move. The rules are incredibly simple.

1. If the energy change is negative, accept the move.
2. If the energy change is positive, generate a random number between 0 and 1. If that random number is less than $e^{-\frac{\text{energy change}}{kT}}$, accept the move. Otherwise, reject it.

Entropy in the Metropolis Algorithm

The Metropolis algorithm at first sight seems to not include entropy, the randomness of a system. After all, it is based solely on energy reduction. The entropy, however, is actually really subtle. There are two ways to think it. The first way is to examine how particles are moved in the algorithm. The direction of movement is random. In addition, it is a clear that a particle can only move to an unoccupied location. Therefore, if that particle is in an organized group, the only direction it can move is away from that group, thus slightly disturbing the order in the system. The second way to understand entropy is to examine the energies. In many cases, the energies themselves secretly encode entropy. In our case, that energy is the surface strain energy. If particles begin clumping together, the strain induced on the surface increases, which in turn increase the energy. That increase in energy due to a high concentration of particles in

one region causes those particles to spread out, once again disturbing some sort of order. So, entropy is in fact included in the algorithm.

Verifying the Metropolis Algorithm

To determine whether we coded the algorithm rules correctly, we tested our code on one distribution. That distribution is based on Boltzmann's distribution in a two energy system. For this simulation, the simulated region is a box having side length 200, meaning that there are a total of 40000 (=$M$) sites on the lattice. This box is divided in to two rectangles, one with 10000 (=$M_1$) sites and the other with 30000 (=$M_2$) sites. The rectangle containing 10000 sites is assigned a unit-less energy of 3 while the other rectangle is assigned an energy of 0 (we let Boltmann's constant $k$ and $T$ be 1 for simplicity). In our simulations, we placed down 20000 particles.

To test our code, we look at the ratio of the number of particles in one region to the number of particles in the other region. From our derivation of Boltzmann's distribution for a two energy system, we find that $\frac{N_2}{N_1} = 0.0423$, where $N_2$ is the number of particles having energy 1 and $N_1$ is the number of particles having energy 0.

| Number of moves | Ratio $N_2/N_1$ |
|---|---|
| 100000 | 0.33129202 |
| 1000000 | 0.32231404 |
| 2000000 | 0.31648235 |
| 10000000 | 0.28924128 |
| 20000000 | 0.27048660 |
| 100000000 | 0.20141767 |
| 200000000 | 0.15667110 |
| 300000000 | 0.12549240 |
| 400000000 | 0.10674561 |
| 500000000 | 0.09481060 |
| 600000000 | 0.08483402 |
| 700000000 | 0.07874865 |
| 1000000000 | 0.05982725 |

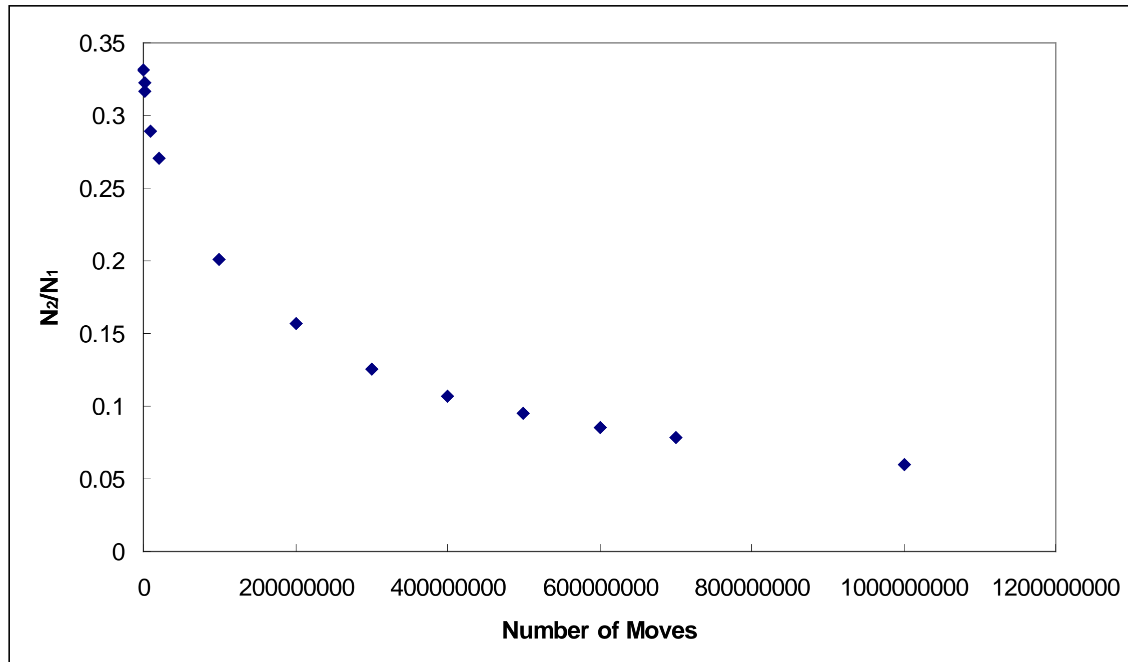



**Figure 3: Graph of N2/N1 as a function on the number of moves. The curve formed by the points approach ~0.04-0.05**

We did not run the simulation long enough to obtain points around 0.42. However, based on the table and graph, it is clear that 2000000000 moves, the ratio should begin to hover around 0.42.

<u>Energies in Our System</u>

There are two energies defined in our system: interaction energy and the misfit strain energy. The interaction energy is like the boundary energy in our previous project while the misfit strain energy is related to the surface strain energy in out previous project.

The interaction energy essentially models bonding and is determined by the number of neighbors. In our specific system, the maximum number of neighbors a particle can have is four. In our program, the energy is a function of the number of neighbors. In many cases, as that number increases, the energy decreases. If a particle

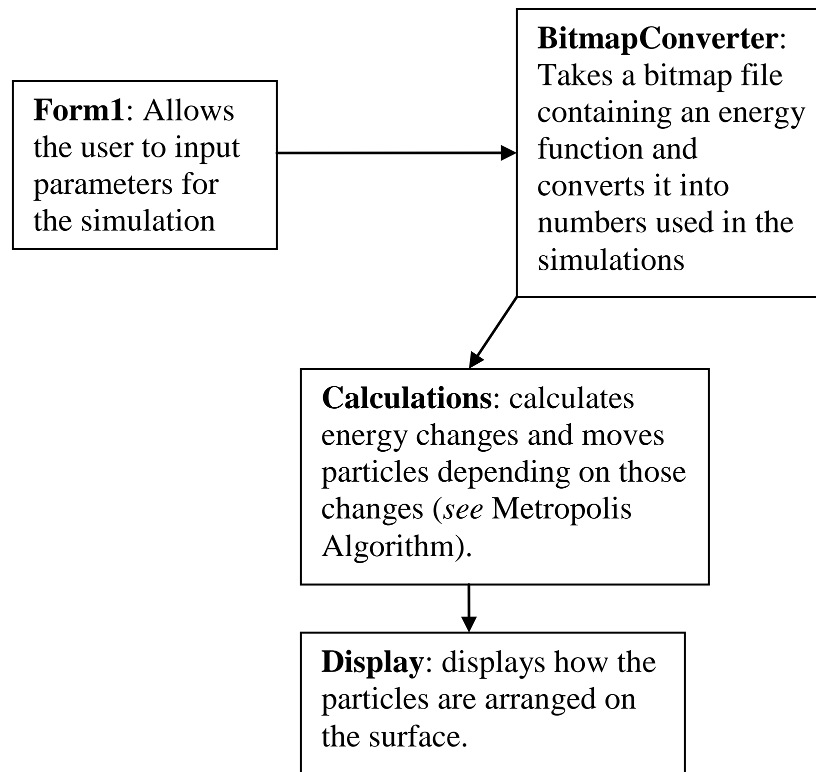
loves to “bond” or interact with other particles, then its energy will drop significantly for each addition particle it finds.

The misfit strain energy essentially models how much strain a layer of particles induces on the surface. The lattices constants for the particles and the surface are usually different, which creates misfit. Particles typically want to align themselves with the surface to reduce energy. However, due to the size difference, many particles will not be able to align and will instead become offset. This creates strain and potential energy in the surface. Below is a side view of the particles and surface.

```
Film:         |     |     |     |     |     |     |     |     |     |
Lattice:      |    |    |    |    |    |    |    |    |    |    |    |
```

To calculate the misfit energy, we look at the distances between the midpoints of the film and the lattice. One can view each line above as one particle. In our program, we select two adjacent particles in the film. That gives us an interval between those two particles. We then find all of the midpoints that lie on that interval. In most cases, there is only one such midpoint. Once we find all of the corresponding midpoints, we take the average of the distances between the midpoint of the two particles and the corresponding midpoints in the lattice/surface. A particle that is aligned with the surface will have an energy of 0 because the midpoints will line up. However, if the particle is offset by a small amount, the midpoints will not line up, thus generating energy. This energy we describe is a function of the distance between the midpoints. For simplicity and flexibility, we let that function be a polynomial. Users can include as many terms in the polynomial as they wish, depending on the shape of the function.

## Architecture of the Program




The structure of the program is straightforward. The program is written in C#, which is very similar to Java. In Form1, the user first inputs values for the parameters and starts the simulation. The calculations are performed in Calculations and then the results are displayed in Display. The calculations can be run over and over until satisfactory results are obtained.

The BitmapConverter reads the hexadecimal numbers contained in the bitmap file and assigns energy levels to those numbers, or colors if one looks at the picture (e.g. black = 1 and white = 0).

## Results and Discussion

The three most important results that we must reproduce with our program are quantum dots (patches of particles), serpentine stripes (particles arranged in snake-like shapes), and quantum pits (patches or "holes" where there are no particles). At lower concentrations, we should be getting quantum dots. As we increase the concentration or the number of particles, the results should transition from quantum dots to serpentine strips and finally from serpentine stripes to quantum pits. Physically, this makes since. When the number of particles is low, serpentine stripes cannot form because the energy can still be lowered by breaking the serpentine stripes into quantum dots. As the number of particles increases, it becomes difficult to form quantum dots because the limited room would create rather large patches, which actually would have a higher energy than serpentine stripes. Once the particles cover a majority of the surface, quantum dots become impractical (they would be enormous). There would not be enough room to for m serpentine stripes. They would be so close together that they would begin to merge. Thus, quantum pits form.

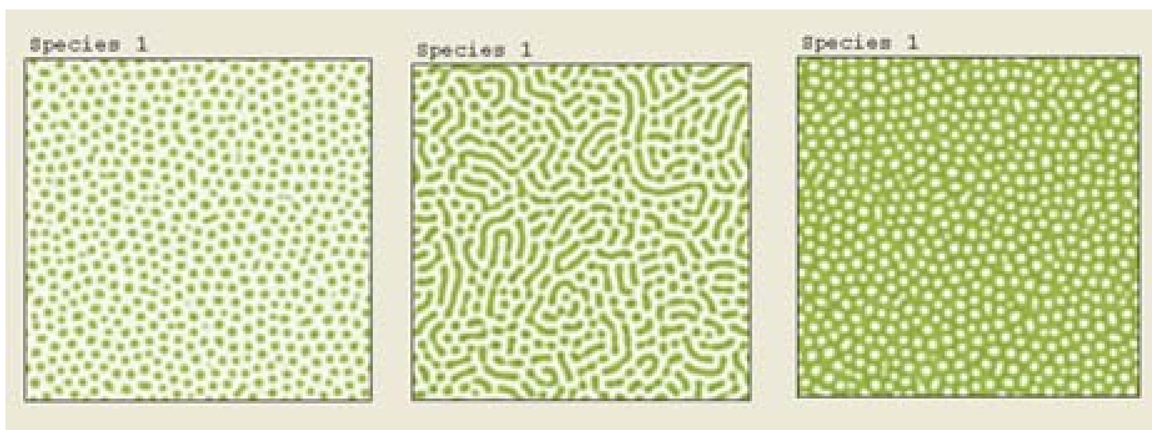



**Figure 4: Simulations from the past project. In the image on the far left, we see quantum dots (low concentration). In the middle image, we see serpentine stripes (medium concentration). In the image on the far left, we see quantum pits (high concentration).**

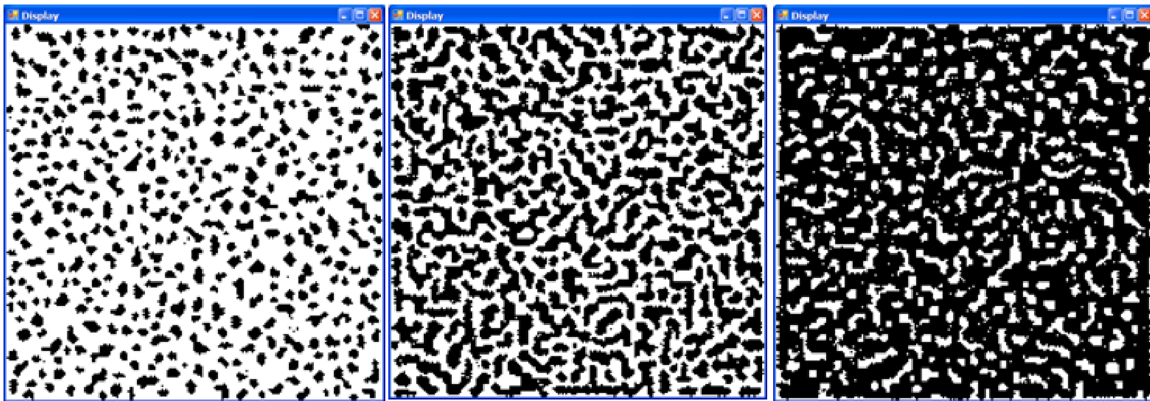



**Figure 5: As in Figure 4, we see the transition from quantum dots to serpentine stripes and from serpentine stripes to quantum pits. Note that these simulations are more microscopic than those in Figure 4.**

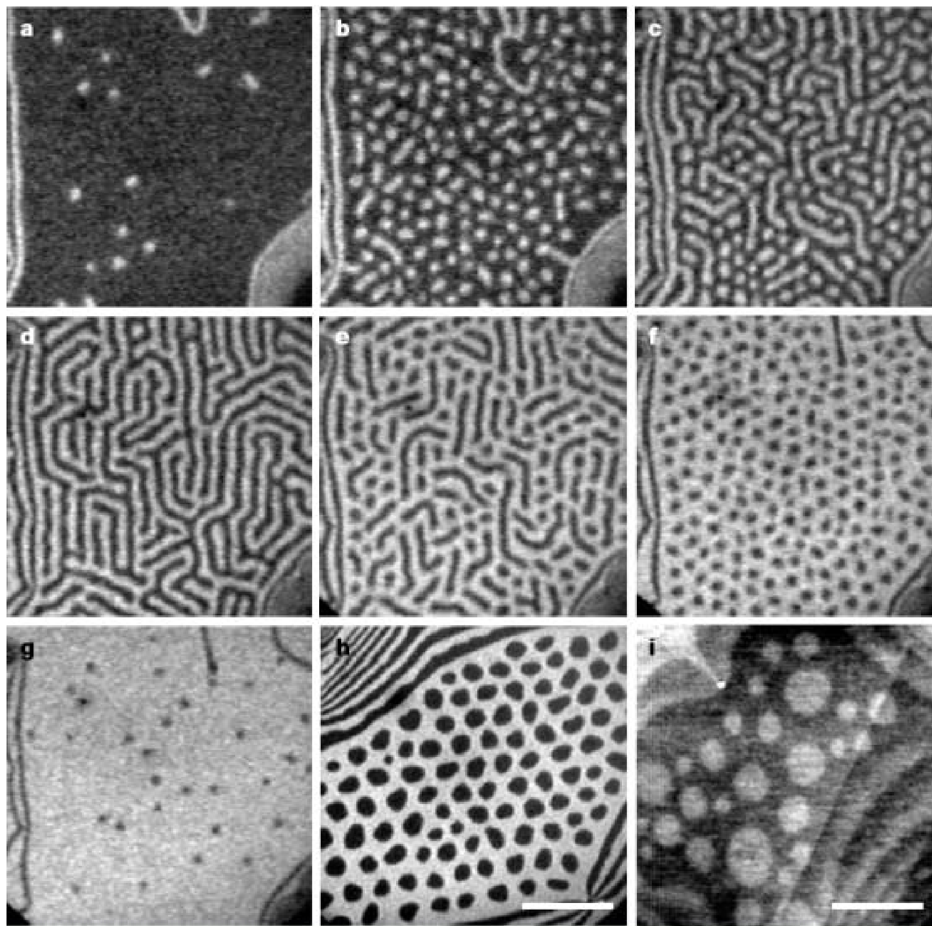



**Figure 6: Experimental observations of the nanoscale self-assembly of Pb (lead) on Cu (copper) (Plass et al., 2001). A transition from quantum dots through serpentine stripes to quantum pits can be clearly seen in b-f.**

The final simulation we perform shows the effect of misfit strain energy. As predicted, misfit strain energy (also known as surface strain energy) should cause the system to form finer patterns. For example, quantum dots should be smaller. Below, we compare a simulation using small, almost negligible strain to one that uses a much larger strain.

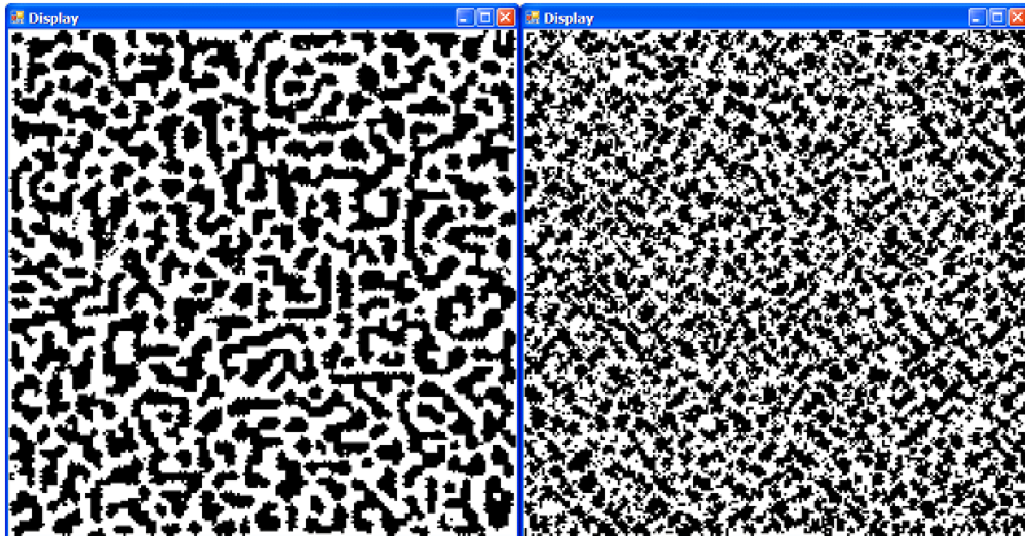



**Figure 7: This camparison show the effect of strain energy. The figure on the right didn't not use any strain energy while the figure on the lefet used $10x+10x^2$ (x is the distance between midpoints) as the strain energy function, which is significantly larger. Due to the strain energy, the patterns in the system on the left become much finer because the system is more sensitive to concentration.**

Based on the simulations above and the numerous reruns, our program has successfully simulated the qualitative features of our system. We successfully reproduced quantum dots, serpentine stripes, and quantum pits. These results match accurately with our old program and experimental results. In addition, we successfully included the effect of strain on the patterns.

<u>Future Plans</u>

There were many things that we could not include in our finished product. We finished the most essential parts of the program. There are actually three improvements

that we plan on making. The first has to do with parallel computing. The Metropolis algorithm can be run on multiple cores. Not surprisingly, the new algorithm is called the Parallel Metropolis algorithm (*see* Appendix A for a brief description). We would also like to expand our system more. In other words, generalize it. Now that we have the basic components finished, we can start adding more energy so that we can apply our program to many other systems. For example, we can study how dipoles might arrange on the surface or how an electric or magnetic field might effect pattern formation, which could be useful to know if nanoscale circuits are used. All of these additions are extremely easy to include. Unlike the past project, we will not have to derive, if possible, any differential equations when we make the system more complicated. It might be impossible to derive equations for such complicated systems. Instead, we simply determine what new energies to include. Finally, we would like to include heterogeneous initial distributions, that is, the user can put whatever initial distribution (e.g. a circuit) and see how it changes under certain conditions.

**Acknowledgements**

We would like to thank our mentor, Professor David Dunlap, for his invaluable help in demystifying the Metropolis Algorithm and giving us a brief introduction to thermodynamics. We would also like to tank Jim Mims for supporting us along the way.

## References


Chen, L.-Q. and Shen, J., Applications of semi-implicit Fourier spectral method to phase field equations, Comp. Phys. Commun. 108 (1998) 147–158.

Hu, Shaowen, Girish Nathan, Fazle Hussain, Donald J. Kouri, Pradeep Sharma, and Gemunu H. Gunaratne. "On Stability of Self-Assembled Nanoscale Patterns." Journal of the Mechanics and Physics of Solids 55 (2007): 1357-1384. ScienceDirect. Elsevier. 28 Nov. 2007 <http://www.sciencedirect.com/>.

Johnson, K. L. "Point Loading of an Elastic Half-Space." Contact Mechanics. Cambridge, U.K.: Cambridge University Press, 1985. 45-83.

Kaganovskii, Yu. S., L. N. Paritskaya, and V. V. Bogdanov. "Kinetics and Mechanisms of Intermetallic Growth By Surface Interdiffusion." Mat. Res. Soc. Symp. Proc. 527 (1998): 303-307.

Lu, Wei. "Theory and Simulation of Nanoscale Self-Assembly on Substrates." Journal of Computational and Theoretical Nanoscience 3.3 (2006): 342-361.

Lu, Wei, and Dongchoul Kim. "Dynamics of Nanoscale Self-Assembly of Ternary Epilayers." Microelectronic Engineering 75 (2004): 78-84. ScienceDirect. 26 Feb. 2004. Elsevier. 28 Nov. 2007 <http://www.sciencedirect.com/>.

- - -. "Patterning nanoscale Structures by Surface Chemistry." Nano Letters 4.2 (2004): 313-316.

- - -. "Simulation on Nanoscale Self-Assembly of Ternary-Epilayers." Computational Materials Science 32 (2005): 20-30. ScienceDirect. Elsevier. 8 Dec. 2007 <http://www.sciencedirect.com/>.

Plass, Richard, Julie A. Last, N. C. Bartelt, and G. L. Kellogg. "Self-Assembled Domain Patterns." Nature 412 (Aug. 2001): 875. 25 Mar. 2008 <http://www.nature.com>.

Ratsch, C. and Venables J. A., Nucleation Theory and the early stages of thin film growth, J. Vac. Sci. Technol. A, Vol. 21, No. 5 (2003) S96-S109

Roduner, Emil. Nanoscopic Material: Size-Dependent Phenomena. Cambridge: The Royal Society of Chemistry, 2006.

“Metropolis Algorithm Statistical System and Simulated Annealing.” N.d. *Physics 170*. Web. 7 Apr. 2010. <http://kossi.physics.hmc.edu/courses/p170/Metropolis.pdf>.

## Appendix A: Brief Description of the Parallel Metropolis Algorithm

As its name suggests, the Parallel Metropolis algorithm is a parallel version of the Metropolis algorithm. To run the parallel version, the simulation grid is divided into sections, each sections is handled by one processor.

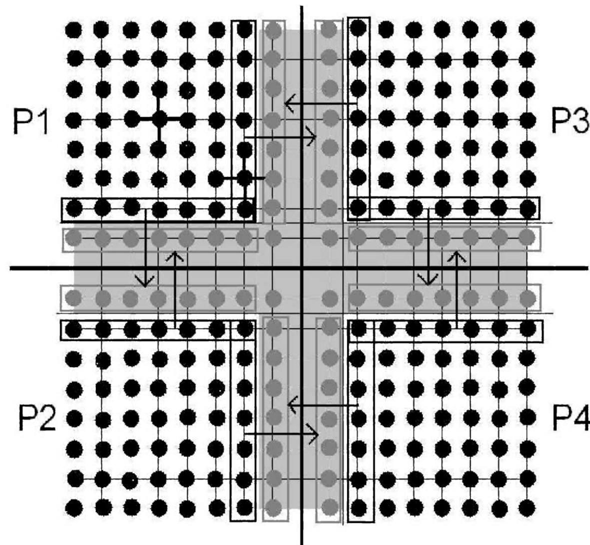



**Figure 8: The schematics for the Parallel Metropolis Algorithm. Gray cells represent ghost cells. http://www.fysik.uu.se/cmt/berg/node31.html**

In Figure 8, the grid is divided into four quadrants. Each quadrant contains ghost cells (gray areas) that hold information about the adjacent quadrant. As the program ones, the ghost cells are continuously updated. Currently, our program runs significantly faster than the program in our past project. However, we noticed that as the system becomes larger (more than a 200x200 grid), the program gets slower and slower. By implementing the Parallel Metropolis Algorithm, we hope to further reduce the simulation time, which would allow us to simulate much larger and more interesting systems.

## Appendix B: Screenshots of the Program

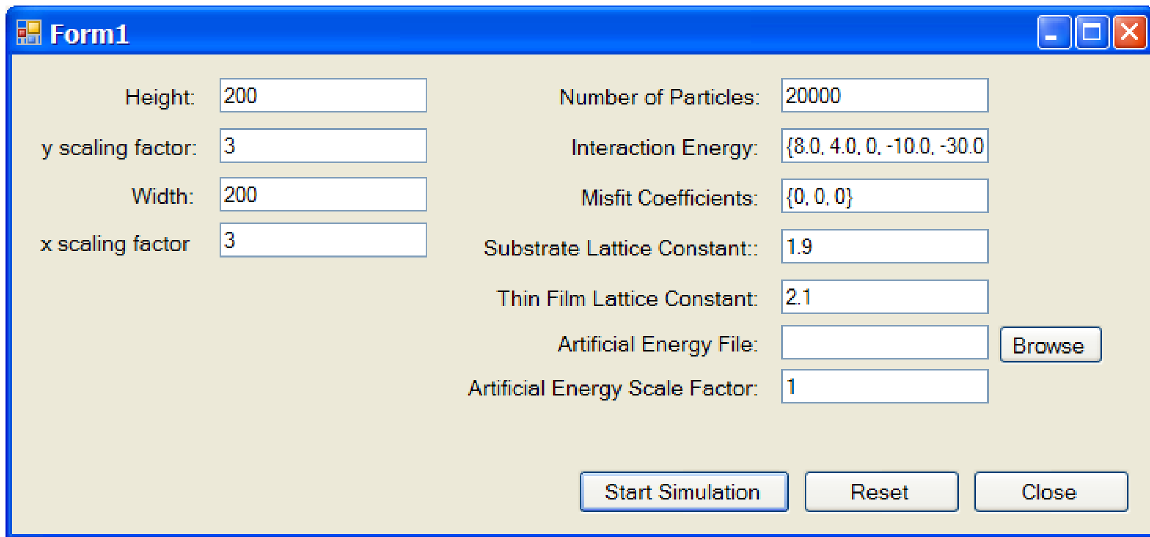



**Figure 9: Form 1. Here, users can input parameter values. The x and y scaling factors just change the pixiel size of each particle.**

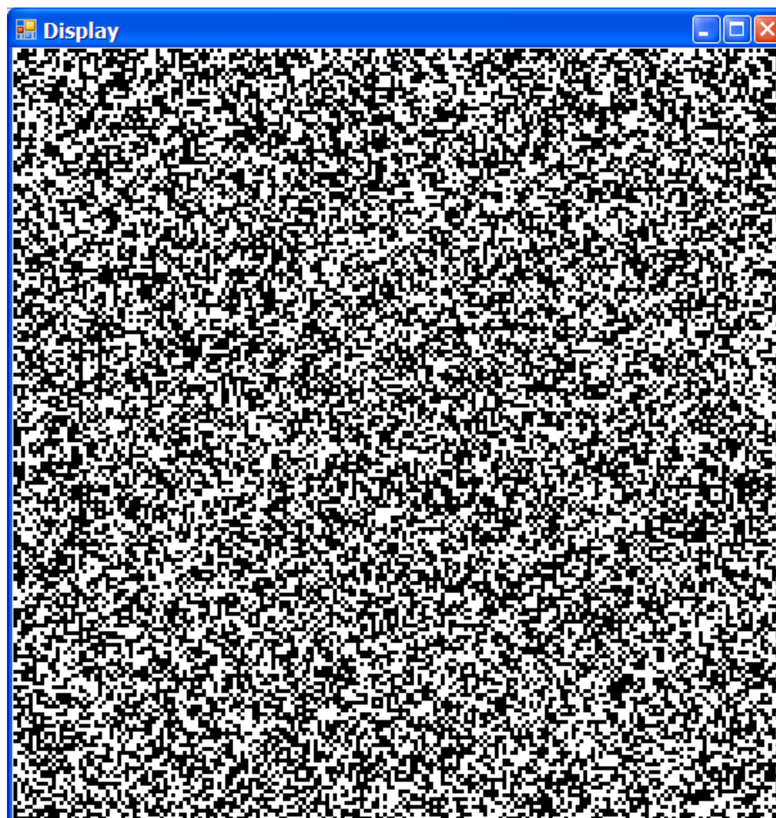



**Figure 10: Display screen. This is where the result is displayed. The current image is one of a random initial distribution. Attached to this screen is a little control device on which the user can continue the calculations or exit the program.**

# Appendix C: The Code

**Form1**

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MetropolisNanoSim
{
    public partial class Form1 : Form
    {
        OpenFileDialog ofd = new OpenFileDialog();

        //-----------------------------------------------------------------
----------------------------
        /**
         * Constructor
         */
        public Form1()
        {
            InitializeComponent();
        }

        //-----------------------------------------------------------------
----------------------------
        /**
         * Start the simulation
         */
        private void bt_start_Click(object sender, EventArgs e)
        {
            int height = int.Parse(tb_height.Text); //height (in pixels)
of the simulated region
            int yScale = int.Parse(tb_yscale.Text);  //height of an
individual grid point
            int width = int.Parse(tb_width.Text);  //width (int pixels)
of the simulated region
            int xScale = int.Parse(tb_xscale.Text);  //width of an
individual grid point
            int numP = int.Parse(tb_numparticles.Text);  //number of
particles
            double[] ie = toArray(tb_interaction.Text);   //interaction
energy
            double latConst = double.Parse(tb_lattice.Text); //lattice
constant of the substrate
            double filmConst = double.Parse(tb_film.Text);  //lattice
constant of the film
            double[] coef = toArray(tb_misfit.Text); //strain energy
polynomial coefficients
            String AEFileLocation = tb_AEFileLoc.Text;
```

```
            double AEScaleFactor = double.Parse(tb_AEScaleFactor.Text);

            if (AEFileLocation != "")
            {
                BitmapConverter bmpc = new BitmapConverter(height,
width, numP, ie, latConst, filmConst, coef, xScale, yScale,
AEFileLocation, AEScaleFactor, true);
            }
            else
            {
                BitmapConverter bmpc = new BitmapConverter(height,
width, numP, ie, latConst, filmConst, coef, xScale, yScale,
AEFileLocation, AEScaleFactor, false);
            }
        }

        //--------------------------------------------------------------
----------------------------
        /**
         * Convert the string "{value1, value2, ...}" into an array
containing value1, value2,...
         * Precondition: string is in the correct format
         */
        private double[] toArray(String s)
        {
            List<double> vals = new List<double>();
            bool searching = false;
            string sub;
            string val = "";

            for (int i = 0; i < s.Length; i++)
            {
                sub = s.Substring(i, 1);
                //if 'sub' is a (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, -, .),
append it to 'val' until the
                //end of the number is reached
                if (!sub.Equals(",") && !sub.Equals(" ")
&& !sub.Equals("{") && !sub.Equals("}"))
                {
                    val += sub;
                    searching = true;
                }
                //once at the end of the number, convert it to a double
and add to the list
                //don't do this again until another number is found
                else if (searching)
                {
                    vals.Add(double.Parse(val));
                    val = "";
                    searching = false;
                }
            }

            //convert 'vals' and return the array
            return vals.ToArray();
        }
```

```
        //---------------------------------------------------------------
----------------------------
        /**
         * Close the program
         */
        private void bt_close_Click(object sender, EventArgs e)
        {
            this.Dispose();
        }

        private void bt_reset_Click(object sender, EventArgs e)
        {
            tb_height.Text = "200";
            tb_yscale.Text = "3";
            tb_width.Text = "200";
            tb_xscale.Text = "3";
            tb_numparticles.Text = "20000";
            tb_interaction.Text = "{8, 4, 0, -10, -32}";
            tb_lattice.Text = "1.9";
            tb_film.Text = "2.1";
            tb_misfit.Text = "{0, 0, 0}";
            tb_AEFileLoc.Text = "";
            tb_AEScaleFactor.Text = "1";
        }

        private void bt_browse_Click(object sender, EventArgs e)
        {
            ofd.Filter = "Bitmap Image|*.bmp";
            ofd.Title = "Open an Image File";
            ofd.FileName = "";
            ofd.ShowDialog();

            tb_AEFileLoc.Text = ofd.FileName;
        }
    }
}
```

**BitmapConverter**

```
using System;
using System.IO;
using System.Collections.Generic;
using System.Text;

namespace MetropolisNanoSim
{
    class BitmapConverter
    {
        /**
         * constructor
         */
        public BitmapConverter(int sim_height, int sim_width, int
numParticles, double[] intEnergy,
            double latConst, double filmConst, double[] misfit_coefs,
int xScl, int yScl, String FileLocation, double scaleFactor, bool
fileNameExists)
```

```
        {
            Calculations c = new Calculations(sim_height, sim_width,
numParticles, intEnergy, latConst, filmConst, misfit_coefs,
colorToEnergyLevel(FileLocation, sim_height, sim_width, scaleFactor,
fileNameExists));
            Display disp = new Display(c, sim_height, sim_width, xScl,
yScl);
        }

        //------------------------------------------------------------
------
        /**
         * This returns the concentrations corresponding to the colors
in
         * the bmp file
         */
        private Dictionary<int, double> colorToEnergyLevel(String
filename, int height, int width, double AEScale, bool nameExists)
        {
            Dictionary<int, double> AEMat = new Dictionary<int,
double>();
            byte hex1, hex2, hex3;
            int pixelColor, hash;
            int yVal = 1;
            int xVal = 1;
            hash = width + 1;

            if (nameExists)
            {
                FileStream file = new FileStream(filename,
FileMode.Open, FileAccess.Read);
                BinaryReader reader = new BinaryReader(file);

                // read past the 54 byte header
                for (int i = 0; i < 54; i++)
                {
                    reader.ReadByte();
                }

                for (int i = 1; i < height * width + 1; i++)
                {
                    hex1 = reader.ReadByte();
                    hex2 = reader.ReadByte();
                    hex3 = reader.ReadByte();
                    pixelColor = (hex1 << 16) + (hex2 << 8) + hex3;
                    switch (pixelColor)
                    {
                        case 0x000000://black
                            AEMat.Add(xVal + hash * yVal, 0.99 *
AEScale);

                            break;
                        case 0x061310:
                            AEMat.Add(xVal + hash * yVal, 0.95 *
AEScale);

                            break;
                        case 0x0E2520:
```

```
                                AEMat.Add(xVal + hash * yVal, 0.9 *
AEScale);
                                break;
                            case 0x143830:
                                AEMat.Add(xVal + hash * yVal, 0.85 *
AEScale);
                                break;
                            case 0x1D4940:
                                AEMat.Add(xVal + hash * yVal, 0.8 *
AEScale);
                                break;
                            case 0x245B4F:
                                AEMat.Add(xVal + hash * yVal, 0.75 *
AEScale);
                                break;
                            case 0x2B6F60:
                                AEMat.Add(xVal + hash * yVal, 0.70 *
AEScale);
                                break;
                            case 0x32816F:
                                AEMat.Add(xVal + hash * yVal, 0.65 *
AEScale);
                                break;
                            case 0x37957D:
                                AEMat.Add(xVal + hash * yVal, 0.6 *
AEScale);
                                break;
                            case 0x3EA88D:
                                AEMat.Add(xVal + hash * yVal, 0.55 *
AEScale);
                                break;
                            case 0x48B798:
                                AEMat.Add(xVal + hash * yVal, 0.5 *
AEScale);
                                break;
                            case 0x59BFA8:
                                AEMat.Add(xVal + hash * yVal, 0.45 *
AEScale);
                                break;
                            case 0x6CC6B4:
                                AEMat.Add(xVal + hash * yVal, 0.4 *
AEScale);
                                break;
                            case 0x7ECDBD:
                                AEMat.Add(xVal + hash * yVal, 0.35 *
AEScale);
                                break;
                            case 0x93D2C5:
                                AEMat.Add(xVal + hash * yVal, 0.30 *
AEScale);
                                break;
                            case 0xA4DBCE:
                                AEMat.Add(xVal + hash * yVal, 0.25 *
AEScale);
                                break;
                            case 0xB6E2D9:
```

```
                            AEMat.Add(xVal + hash * yVal, 0.20 *
AEScale);
                            break;
                        case 0xC8EAE3:
                            AEMat.Add(xVal + hash * yVal, 0.15 *
AEScale);
                            break;
                        case 0xDAF1ED:
                            AEMat.Add(xVal + hash * yVal, 0.1 *
AEScale);
                            break;
                        case 0xEDF8F5:
                            AEMat.Add(xVal + hash * yVal, 0.05 *
AEScale);
                            break;
                        default:
                            AEMat.Add(xVal + hash * yVal, 0.05 *
AEScale);
                            break;
                    }

                    xVal++;
                    if (xVal == width + 1)
                    {
                        yVal++;
                        xVal = 1;
                    }
                }

                reader.Close();
                file.Close();

                return AEMat;
            }
            else
            {
                for (int i = 1; i < height + 1; i++)
                {
                    for (int j = 1; j < width + 1; j++)
                    {
                        AEMat.Add(i + hash * j, 0);
                    }
                }

                return AEMat;
            }
        }
    }
}
```

## Calculations

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Text;
```

```
namespace MetropolisNanoSim
{
    public class Calculations
    {
        int height; //height of the simulated region
        int width; //width of the simulated region
        int numP; //number of particles
        double totalE; //energy of the system at a particular state
        double T; //temperature
        double al; //lattice constant of the substrate
        double af; //lattice constant of the film
        Dictionary<int, Particle> particle; //collection of particles
        Dictionary<int, double> artificialE;
        List<int> locs; //collection of locations
        Random r; //random number
        //const double kb = 1.3806505e-23; //Boltzmann's constant
        const double kb = 1; //Boltzmann's constant
        int[,] directions = { { 0, -1 }, { 1, 0 }, { 0, 1 }, { -1,
0 } }; //possible move directions
        double[] interaction; //interaction energy
        double[] coef; //misfit energy polynomial coefficients
        int hash; //number used to create the hashing function

        //------------------How the hashing function works-------------
----//
        /**
         * h(x,y) = x + (width + 1) * y
         * As long as x is in the grid, h(x,y) will be unique for all y.
         *
         * Brief Proof (if it isn't already intuitive):
         * Suppose h(x1,y1) = h(x2,y2), or x1 + hash * y1 = x2 + hash *
y2.
         * Rewrite this as y2 = (x1 - x2) / hash + y1.  Since y2 is an
         * integer, (x1 - x2) / hash must also be an integer.
Therefore,
         * x1 - x2 = hash * integer.  But this implies that x1 or x2
must
         * be off the grid.  So this h(x,y) works!
         *
         * Modifications to the code:
         * In some of the methods, there is a temporary variable.
Instead
         * of calculating the x and y coordinates and THEN calculating
the
         * adjacent location variable, we simply the calculation by
expanding
         * the adjacent location variable.  In other words...
         * (x + xChange) + hash * (y + yChange) -->
         * x + hash * y + xChange + hash * yChange -->
         * location variable + temporary variable (= xChange + hash *
yChange)
         *
         * In some cases, this will reduce the number of operations.
         */
```

```
        //--------------------------------------------------------------
----------------------------
        /**
         * Constructor
         */
        public Calculations(int simulation_height, int simulation_width,
int number_of_particles, double[] interaction_energy,
            double lattice_constant, double film_lattice_constant,
double[] misfit_coefficients, Dictionary<int, double> artificial_energy)
        {
            height = simulation_height;
            width = simulation_width;
            numP = number_of_particles;
            interaction = interaction_energy;
            al = lattice_constant;
            af = film_lattice_constant;
            hash = width + 1;

            interaction = interaction_energy;
            coef = misfit_coefficients;

            particle = new Dictionary<int, Particle>();
            artificialE = artificial_energy;
            locs = new List<int>();
            r = new Random();

            //distribute the particles randomly over the grid
            distribute_particles();
            //calculate the energy of the inital state
            initial_energy();
        }

        //--------------------------------------------------------------
----------------------------
        /**
         * Return the occupied locations
         */
        public List<int> getOccupiedLocations()
        {
            return locs;
        }

        //--------------------------------------------------------------
----------------------------
        /**
         * Distribute numP particles randomly over the grid
         * Precondition: there are more grid spaces as there are
particles
         */
        private void distribute_particles()
        {
            int tempNum = height + 1; //0<=y<=height
            int loc; //a location
            int numUnOccLocs; //number of unoccupied locations

            //loop through all locations on the grid and apply the
hashing function
```

```
            for (int x = 1; x < hash; x++)
            {
                for (int y = 1; y < tempNum; y++)
                {
                    locs.Add(x + hash * y);
                }
            }
            numUnOccLocs = locs.Count;

            //loop through each particle
            for (int i = 0; i < numP; i++)
            {
                //random location
                loc = locs[r.Next(numUnOccLocs)];

                //place a particle at 'loc'
                particle.Add(loc, update(loc));

                //'loc' is not occupied
                locs.Remove(loc);
                numUnOccLocs--;
            }

            //fill 'locs' with all occupied locations
            locs = new List<int>(particle.Keys);
        }

        //--------------------------------------------------------------
----------------------------
        /**
         * Update all relevant particles involved in placing a particle
at 'location'
         */
        private Particle update(int location)
        {
            int tempVar; //see top of class for explanation
            int counter;
            int addNumInDir; //number of new particles in a specific
directions
            int adjLoc; //adjacent location
            Particle p = new Particle();

            for (int i = 0; i < 4; i++)
            {
                tempVar = directions[i, 0] + hash * directions[i, 1];
                counter = 1;

                //adjLoc = (x + directions[i, 0]) + hash * (y +
directions[i, 1])
                adjLoc = location + tempVar;
                if (particle.ContainsKey(adjLoc))
                {
                    p.NN++; //increment number of nearest neighbors
                    p.numInDir[i] = particle[adjLoc].numInDir[i] + 1;
//update number of particles in direction i
                    particle[adjLoc].NN++; //increment number of
nearest neighbors for particles around 'location'
```

```
                    addNumInDir = p.numInDir[i] + 1;
                }
                else
                {
                    p.numInDir[i] = 0;
                    addNumInDir = 1;
                }

                //adjLoc = (x - directions[i, 0]) + hash * (y - 
directions[i, 1])
                //update all other particles in x and y directions
                adjLoc = location - tempVar;
                while (particle.ContainsKey(adjLoc))
                {
                    particle[adjLoc].numInDir[i] += addNumInDir;
                    counter++;
                    adjLoc = location - counter * tempVar;
                }
            }

            return p;
        }

        //-----------------------------------------------------------
----------------------------
        /**
         * Update all relevant particles involved in moving one 
particle from 'old_location' to
         * 'new_location'
         */
        private Particle update(int old_location, int new_location)
        {
            int tempVar, adjLoc;
            int counter;
            int subtractNumInDir; //number of old particles in a 
specific direction

            for (int i = 0; i < 4; i++)
            {
                tempVar = directions[i, 0] + hash * directions[i, 1];
                counter = 1;

                adjLoc = old_location + tempVar;
                if (particle.ContainsKey(adjLoc))
                {
                    subtractNumInDir = particle[adjLoc].numInDir[i] + 2;
                    particle[adjLoc].NN--;
                }
                else
                {
                    subtractNumInDir = 1;
                }

                adjLoc = old_location - tempVar;
                while (particle.ContainsKey(adjLoc))
                {
                    particle[adjLoc].numInDir[i] -= subtractNumInDir;
```

```
                counter++;
                adjLoc = old_location - counter * tempVar;
            }
        }

        return update(new_location);
    }

    //-----------------------------------------------------------------
----------------------------
    /**
     * Calculate the initial energy of the system
     */
    private void initial_energy()
    {
        Particle p;

        foreach (int loc in locs)
        {
            p = particle[loc];

            //calculate misfit energy
            totalE += misfit_energy(p.numInDir[1] + p.numInDir[3] +
1);
            totalE += misfit_energy(p.numInDir[0] + p.numInDir[2] +
1);

            //calculate
            totalE += interaction[p.NN];
        }
    }

    //-----------------------------------------------------------------
----------------------------
    /**
     * Move the particles
     */
    public void next(int number_of_moves, double temperature)
    {
        T = temperature;
        int numlocs = locs.Count;
        int rDir, tempVar, rLoc, moveLoc;

        for (int i = 0; i < number_of_moves; i++)
        {
            //select a random particle and a random direction
            rLoc = locs[r.Next(numlocs)];
            rDir = r.Next(4);

            tempVar = directions[rDir, 0] + hash * directions[rDir,
1];
            moveLoc = rLoc + tempVar;

            //check if move is valid
            if (isValidMove(moveLoc))
            {
                //check the energy change
```

```
                    if (metropolis(energy_change(rLoc, moveLoc)))
                    {
                        //remove from old location
                        particle.Remove(rLoc);
                        locs.Remove(rLoc);

                        //add to new location
                        particle.Add(moveLoc, update(rLoc, moveLoc));
                        locs.Add(moveLoc);
                    }
                }
            }
        }

        //-----------------------------------------------------------------
----------------------------
        /**
         * Check if 'location' is a valid location.  'location' cannot
be occupied or outside the grid
         */
        private bool isValidMove(int location)
        {
            int x = location % hash;
            int y = (location - x) / hash;

            //return false if the location is occupied or it is outside
the grid
            if (x == 0 || y == 0 || y > height ||
particle.ContainsKey(location))
            {
                return false;
            }
            else
            {
                return true;
            }
        }

        //-----------------------------------------------------------------
----------------------------
        /**
         * Calculate the energy change associated with moving a
particle from 'old_location' to
         * 'new_location'
         */
        private double energy_change(int old_location, int new_location)
        {
            double Ei = 0;
            double Ef = 0;
            int tempVar, adjLoc;
            int NNNew = 0;
            int[] length = { 0, 0 };
            Particle p;

            Ei += interaction[particle[old_location].NN];
            Ei += misfit_energy(particle[old_location].numInDir[1] +
particle[old_location].numInDir[3] + 1);
```

```
            Ei += misfit_energy(particle[old_location].numInDir[0] +
particle[old_location].numInDir[2] + 1);
            Ei += artificialE[old_location];
            for (int i = 0; i < 4; i++)
            {
                tempVar = directions[i, 0] + hash * directions[i, 1];

                adjLoc = old_location + tempVar;
                if (particle.ContainsKey(adjLoc))
                {
                    p = particle[adjLoc];

                    Ei += interaction[p.NN];
                    Ef += interaction[p.NN - 1];

                    Ef += misfit_energy(p.numInDir[i] + 1);
                }
                adjLoc = new_location + tempVar;
                if (particle.ContainsKey(adjLoc))
                {
                    p = particle[adjLoc];

                    Ei += interaction[p.NN];
                    Ef += interaction[p.NN + 1];

                    Ei += misfit_energy(p.numInDir[i] + 1);
                    length[i % 2] += p.numInDir[i] + 1;

                    NNNew++;
                }
            }
            Ef += interaction[NNNew];
            Ef += misfit_energy(length[0] + 1);
            Ef += misfit_energy(length[1] + 1);
            Ef += artificialE[new_location];

            return (Ef - Ei);
        }

        //------------------------------------------------------------
----------------------------
        /**
         * Apply the metropolis algorithm given a change in energy
         */
        private bool metropolis(double difference)
        {
            if (difference <= 0)
            {
                //if energy change is 0, hte particle has a 50% chance
of moving
                if (difference == 0 && r.NextDouble() > 0.5)
                {
                    return false;
                }
                totalE += difference;
                return true;
            }
```

```
            else if (r.NextDouble() < Math.Pow(Math.E, -difference /
(kb * T)))
            {
                totalE += difference;
                return true;
            }
            else
            {
                return false;
            }
        }

        //-----------------------------------------------------------
----------------------------
        /**
         * Calculate the misfit energy
         */
        private double misfit_energy(int length)
        {
            List<short> j = new List<short>();
            int ith = 0;
            short jth = 1;
            double average;
            double sum = 0;
            int upperLim = length / 2 + 1;
            double sumOdd = 0;
            bool isOdd;

            if (length % 2 != 0)
            {
                isOdd = true;
            }
            else
            {
                isOdd = false;
            }

            do
            {
                ith++;
                average = 0;
                j.Clear();
                j.TrimExcess();

                while ((ith - 1) * af <= (jth - 1 / 2) * al && (jth - 1
/ 2) * al <= ith * af)
                {
                    j.Add(jth);
                    jth++;
                }

                if ((jth - 1 / 2) * al == ith * af)
                {
                    jth--;
                }

                if (j.Count == 0)
```

```
                {
                    j.Add(jth);
                    j.Add((short)(jth - 1));
                }

                foreach (short js in j)
                {
                    average += Math.Abs((ith - 1 / 2) * af - (js - 1 /
2) * al);
                }
                average /= j.Count;
                sum += polynomial(coef, average);

                if (isOdd && ith == upperLim - 1)
                {
                    sumOdd = sum;
                }

            } while (ith < upperLim);

            if (isOdd)
            {
                return (sum + sumOdd);
            }
            else
            {
                return (2 * sum);
            }
        }

        private double polynomial(double[] c, double value)
        {
            int n = c.Length - 1;
            double result = c[n];

            for (int i = n - 1; i >= 0; i--)
            {
                result = result * value + c[i];
            }

            return result;
        }

        //------------------------------------------------------------
----------------------------
        /**
         * Calculate the ratio.  Simply used to testing.
         */
        /*public double calcRatio()
        {
            double ratio;
            int x, y;
            double N1 = 0;
            double N2 = 0;

            for (int i = 0; i < numP; i++)
            {
```

```
                x = locs[i] % hash;
                //y = (location - x) / hash;

                //this part is hard coded just for testing
                if (x > 150)
                {
                    N2++;
                }
                else
                {
                    N1++;
                }
            }
            ratio = N2 / N1;

            return ratio;
        }*/
    }
}
```

## Controls

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MetropolisNanoSim
{
    public partial class Controls : Form
    {
        Display disp;
        Calculations calc;
        int totalNumMoves;

        //------------------------------------------------------------
----------------------------
        /**
         * Constructor
         */
        public Controls(Calculations calculations, Display display)
        {
            calc = calculations;
            disp = display;
            totalNumMoves = 0;
            InitializeComponent();
        }

        //------------------------------------------------------------
----------------------------
        /**
         * Move particles
         */
```

```
        private void bt_makemoves_Click(object sender, EventArgs e)
        {
            int numMoves = int.Parse(tb_nummoves.Text);
            double temp = double.Parse(tb_temperature.Text);
            totalNumMoves += numMoves;
            tb_totalmoves.Text = totalNumMoves + "";

            calc.next(numMoves, temp);
            //tb_totalmoves.Text += ", " + calc.calcRatio();

            disp.display_state(sender);
        }

        //------------------------------------------------------------
----------------------------
        /**
         * Close the simulation window
         */
        private void bt_close_Click(object sender, EventArgs e)
        {
            this.Dispose();
            disp.Dispose();
        }
    }
}
```

## Display

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Text;
using System.Windows.Forms;

namespace MetropolisNanoSim
{
    public partial class Display : Form
    {
        Calculations calc;
        List<int> newLocs;
        int hash, xScaleFactor, yScaleFactor;

        //------------------------------------------------------------
----------------------------
        /**
         * Constructor
         */
        public Display(Calculations calculations, int height, int width,
int xscale, int yscale)
        {
            calc = calculations;
            hash = width + 1;
            xScaleFactor = xscale;
            yScaleFactor = yscale;
```

```
            InitializeComponent();

            p_disp.Size = new Size(xscale * width, yscale * height);
            this.MaximumSize = new Size(xscale * width + 10, yscale *
height + 36);
            this.MinimumSize = new Size(xscale * width + 10, yscale *
height + 36);

            //create Controls form
            Controls ctrls = new Controls(calculations, this);
            this.Show();
            ctrls.Show();
        }

        //------------------------------------------------------------
----------------------------
        /**
         * Draw the current state of the system
         */
        private void p_disp_Paint(object sender, PaintEventArgs e)
        {
            //clear the screen
            e.Graphics.Clear(Color.White);

            //display the new configuration
            newLocs = calc.getOccupiedLocations();

            SolidBrush blackBrush = new SolidBrush(Color.Black);
            int x, y;

            foreach (long loc in newLocs)
            {
                x = (int)(loc % hash);
                y = (int)((loc - x) / hash);
                //NOTE: points in the panel lie in the grid
[0,width]x[0,height], so x and y needed to be decremented
                e.Graphics.FillRectangle(blackBrush, xScaleFactor * (x
- 1), yScaleFactor * (y - 1), xScaleFactor, yScaleFactor);
            }
        }

        //------------------------------------------------------------
----------------------------
        /**
         * Call p_disp_Paint
         */
        public void display_state(object sender)
        {
            Graphics g = p_disp.CreateGraphics();
            Rectangle r = new Rectangle();
            PaintEventArgs e = new PaintEventArgs(g, r);

            p_disp_Paint(sender, e);
        }
    }
}
```