Nanoscale Self-Assembly

New Mexico
Supercomputing Challenge
Final Report
April 2, 2008

Team 2
Albuquerque Academy

Team Members:
Michael Wang
Ari Shaw-Faber

Teacher:
Jim Mims

Project Mentor:
Yifeng Wang, Ph.D

Table of Contents

Executive Summary

When deposited over a solid surface, some chemicals form patterns at the nanoscale. Two major factors cause this pattern formation. The minimums in Gibb's free energy drive the phase separation of the chemical components. This separation increases the amount surface free energy. To minimize its total energy, the system reacts by reducing the number of phase boundaries. On the other hand, the surface stress produced by concentration variations tends to create finer patterns by increasing the number of phase boundaries. These two opposing factors cause the system to reach equilibrium and form a stable pattern.

This pattern formation is described by a set of nonlinear integral-differential diffusion equations that couple the concentrations and the surface stress. These equations are simplified using the Fourier Transformation, which converts the integral-differential equations into simpler partial differential equations in Fourier space. The Fast Fourier Transform is used to transform values between real and Fourier space.

We successfully wrote a program in C# to simulate this self-assembly process. We modified the equations to include temperature fluctuations. Our simulations agree qualitatively with the experimental results reported in literature and what we expect. We have successfully simulated the transitions between quantum dots, serpentine stripes, and quantum pits. We have shown that heterogeneous pattern formations can be guided by preexisting patterns. We have also shown that temperature can be used to control the size of patterns. This software can be used to understand and design nano pattern formation on solid surfaces. Future work will be focused on improving the numerical method and including other mechanisms for controlling these pattern formations.

Introduction

The self-assembly of nano patterns is very important because it can potentially allow us to create circuits and thus devices at the nanoscale. For example, the smaller we can make circuits, the more "things" we can fit on a single computer chip. This will allow us to create faster computers because data will only need to be transferred over extremely short distances. In addition, the cost and energy required to operate such devices will be greatly reduced. Therefore, nanotechnology can revolutionize our existing industry.

The self-assembly of nano patterns has been observed in many systems (*see* Figure 1). Those patterns are formed when some chemicals are deposited over a surface. Two major factors cause this pattern formation. The minimums in Gibb's free energy drive the phase separation of the chemical components, which increases the amount of surface free energy. To minimize its total energy, the system reacts by reducing the number of phase boundaries. On the other hand, the surface stress produced by concentration variations tends to create finer patterns by increasing the number of phase boundaries. These two opposing factors cause the system to reach equilibrium and form a stable pattern. A computer model simulating such processes can be used for pattern design.

Because pattern formation happens spontaneously, the same self-assembly process could affect the life of pre-made nano devices. Even though we might achieve making nano devices, the patterns on these devices could at any point reassemble into something else, which would cause it to malfunction. Thus, a computer code that

simulates how patterns evolve over time under certain conditions (e.g. temperature fluctuations) could be useful for predicting device lifetime.

The self-assembly of nanoscale patterns is not only important because of their applications, but also because they are so mysterious and fascinating. Many of these patterns are so stunning and beautiful that we would never expect. In this project, we develop computer software to simulate these patterns under constant and changing temperature. Using simulations, we investigate the effects of prescribing an initial pattern and show that initial patterns can be used to control the pattern formations. We also investigate the effects of temperature changes on these patterns.

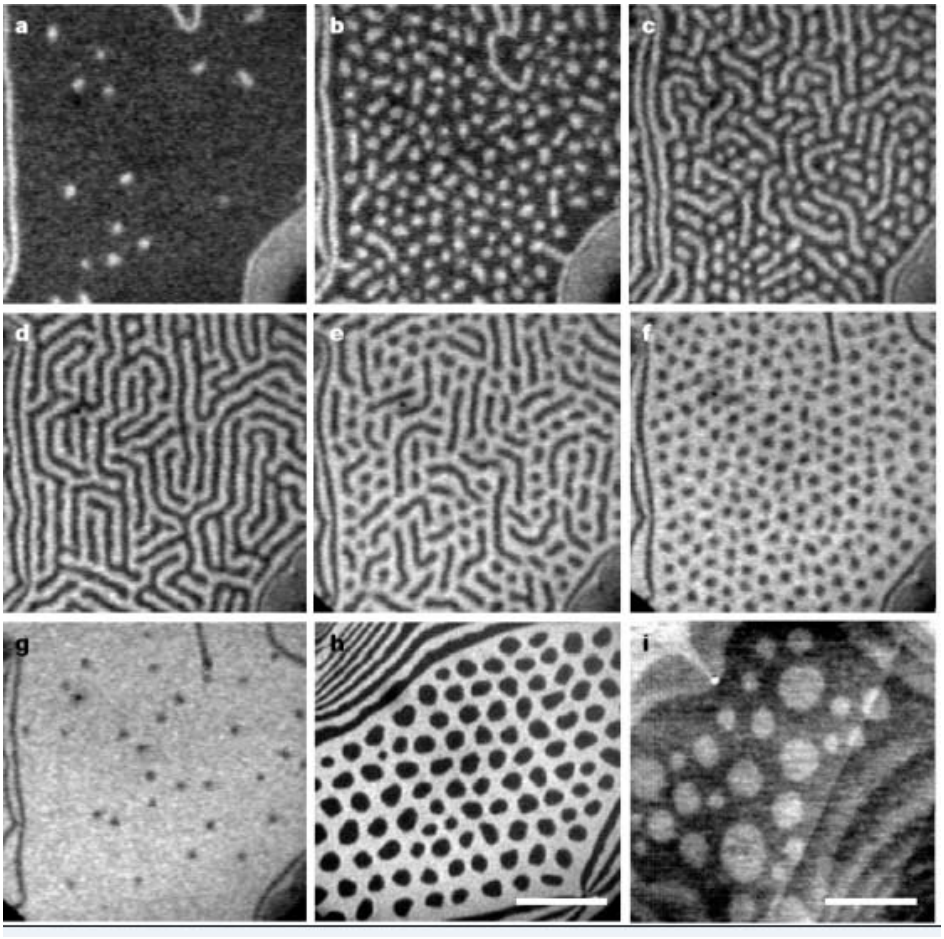


Figure 1: Experimental observations of the nanoscale self-assembly of Pb (lead) on Cu (copper) (Plass et al., 2001). A transition from quantum dots through serpentine stripes to quantum pits can be clearly seen in b-f.

Mathematical Model

The model described below is based on the work of Lu (2006) and Lu and Kim (2005) with our own understanding and some modifications. The mathematical model can be derived using the atomic flux and the driving force of diffusion. The atomic flux vector, $\vec{J} = J_1\vec{i} + J_2\vec{j}$, is defined as the amount of chemical species passing over unit length per unit time.

Deriving the Equations (One Dimensional Version):

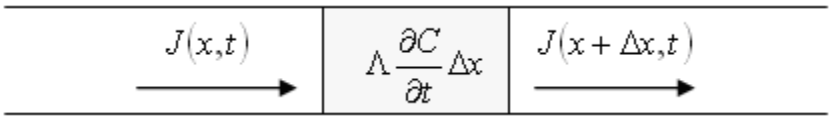


Figure 2: Derivation of one dimensional model

$J(x,t)$ is the amount of chemical species entering the shaded region per unit time and $J(x+\Delta x,t)$ is the amount of chemical species leaving the region per unit time. $\Lambda\frac{\partial C}{\partial t}\Delta x$ is the change in the number of moles of species accumulating (change in concentration) in this region per unit time, where Λ is the number of species per unit length. Thus, the amount of species entering the region minus the amount of species leaving the region is equal to the change in concentration per unit time. In mathematical terms, $J(x,t)-J(x+\Delta x,t)=\Lambda\frac{\partial C}{\partial t}\Delta x$. Bringing the Δx to the left side and taking the limit as $\Delta x \to 0$, we obtain $-\lim_{\Delta x\to 0}\frac{J(x+\Delta x,t)-J(x,t)}{\Delta x}=-\frac{\partial J}{\partial x}=\Lambda\frac{\partial C}{\partial t}$. Using Fick's diffusion law (Pelesko and Bernstein, 2003), which states that the atomic flux is

negatively proportional the gradient of chemical potential (energy stored per mol of a species) per unit length, we can write the above equation as

$\Lambda \frac{\partial C}{\partial t} = -\frac{\partial}{\partial x}\left(-M\frac{\partial \mu}{\partial x}\right) = M\frac{\partial^2 \mu}{\partial x^2}$, where M is the proportionality constant known as the diffusion coefficient, and μ is the chemical potential. This is the one dimensional model of our system and can be extended to two dimensions, which we are studying. The two dimensional derivation is similar but slightly more difficult, and we will not show it here. In the two dimensional model, $M\frac{\partial^2 \mu}{\partial x^2}$ becomes $M\nabla^2\mu$ where $\nabla^2 = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$. As demonstrated by Lu and Kim (2005), μ is defined as $\frac{1}{\Lambda}\left(\frac{\partial \bar{g}}{\partial C} + \frac{\partial f}{\partial C}\varepsilon_{\beta\beta} - 2h\nabla^2 C\right)$, where $\bar{g}$ is the excess energy created from the mixing of the chemical components (*see* Eq. 3), f $(= \psi + \phi C_1 + \eta C_2)$ is the surface stress (surface energy per unit of strain in the surface) assumed to be proportional to concentrations, $\varepsilon_{\beta\beta}$ is the strain in the surface, and h is a constant characterizing the contribution of chemical potential from phase boundaries. Since we are studying the patterns formed by two chemicals, the final set of equations is

$$\frac{\partial C_1}{\partial t} = \frac{M_1}{\Lambda^2}\nabla^2\mu_1 = \frac{M_1}{\Lambda^2}\nabla^2\left(\frac{\partial \bar{g}}{\partial C_1} + \frac{\partial f}{\partial C_1}\varepsilon_{\beta\beta} - 2h_1\nabla^2 C_1\right) \tag{1}$$

$$\frac{\partial C_2}{\partial t} = \frac{M_2}{\Lambda^2}\nabla^2\mu_2 = \frac{M_2}{\Lambda^2}\nabla^2\left(\frac{\partial \bar{g}}{\partial C_2} + \frac{\partial f}{\partial C_2}\varepsilon_{\beta\beta} - 2h_2\nabla^2 C_2\right) \tag{2}$$

where (Lu and Kim, 2005)

$$\begin{aligned}\bar{g}(C_1, C_2) = \Lambda k_B T\{&C_1 \ln C_1 + C_2 \ln C_2 + (1 - C_1 - C_2)\ln(1 - C_1 - C_2) + \\ &C_1 C_2\left[\Omega_{12}^0 + \Omega_{12}^1(C_1 - C_2)\right] + C_1(1 - C_1 - C_2)\left[\Omega_{13}^0 + \Omega_{13}^1(2C_1 + C_2 - 1)\right] + \\ &C_2(1 - C_1 - C_2)\left[\Omega_{23}^0 + \Omega_{23}^1(C_1 + 2C_2 - 1)\right]\}\end{aligned} \tag{3}$$

$$\varepsilon_{\beta\beta} = -\frac{(1-\nu^2)\phi}{\pi E}\iint \frac{(x_1-\xi_1)\frac{\partial C_1}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C_1}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 - \frac{(1-\nu^2)\eta}{\pi E}\iint \frac{(x_1-\xi_1)\frac{\partial C_2}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C_2}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 \tag{4}$$

C_1 — concentration of chemical component 1
C_2 — concentration of chemical component 2
μ_1 — chemical potential of component 1
μ_2 — chemical potential of component 2
M — diffusion coefficient
Λ — moles of component per area
$\bar{g}$ — excess energy created from the mixing of chemicals
k_B — Boltzmann's constant
Ω — bonding strength (subscript such as 12 means component 1 to component 2)
T — absolute temperature
f — surface stress due to concentration variations
$\varepsilon_{\beta\beta}$ — strain in the surface
h_1 and h_2 — constants characterizing chemical potential from phase boundaries
E — Young's modulus (stiffness of substrate)
ν — Poisson's ratio of the substrate
ϕ — surface stress per mole of component 1
η — surface stress per mole of component 2

Scaled Equations:

We now scale the equations to reduce model parameters. The scaled equations are

$$\frac{\partial C_1}{\partial \tau} = \nabla^2\left[P_1(C_1,C_2) - 2\nabla^2 C_1 + \varepsilon^*{}_1\right] \tag{5}$$

$$\frac{\partial C_2}{\partial \tau} = S\nabla^2\left[P_2(C_1,C_2) - 2H\nabla^2 C_2 + \varepsilon^*{}_2\right] \tag{6}$$

where

$$\varepsilon^*{}_1 = -\frac{Q_1}{\pi}\iint \frac{(x_1-\xi_1)\frac{\partial C_1}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C_1}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 - \frac{Q_2}{\pi}\iint \frac{(x_1-\xi_1)\frac{\partial C_2}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C_2}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 \tag{7}$$

$$\varepsilon^*_2 = -\frac{Q_2}{\pi}\iint \frac{(x_1-\xi_1)\frac{\partial C_1}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C_1}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 - \frac{Q_3}{\pi}\iint \frac{(x_1-\xi_1)\frac{\partial C_2}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C_2}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 \tag{8}$$

$$P_1(C_1,C_2) = \frac{1}{\Lambda k_B T}\frac{\partial \overline{g}}{\partial C_1} = \ln\left(\frac{C_1}{1-C_1-C_2}\right) + C_2\left[\Omega_{12}^0 + \Omega_{12}^1(2C_1 - C_2)\right] - C_2\left[\Omega_{23}^0 + \Omega_{23}^1(2C_1+3C_2-2)\right] + \Omega_{13}^0(1-2C_1-C_2) + \Omega_{13}^1\left(6C_1+2C_2-6C_1^2-C_2^2-6C_1C_2-1\right) \tag{9}$$

$$P_2(C_1,C_2) = \frac{1}{\Lambda k_B T}\frac{\partial \overline{g}}{\partial C_2} = \ln\left(\frac{C_2}{1-C_1-C_2}\right) + C_1\left[\Omega_{12}^0 + \Omega_{12}^1(C_1 - 2C_2)\right] - C_1\left[\Omega_{13}^0 + \Omega_{13}^1(3C_1+2C_2-2)\right] + \Omega_{23}^0(1-C_1-2C_2) + \Omega_{13}^1\left(2C_1+6C_2-C_1^2-6C_2^2-6C_1C_2-1\right) \tag{10}$$

$$Q_1 = \frac{b}{l_1}, \qquad Q_2 = \frac{b}{l_2}, \qquad Q_3 = \frac{b}{l_3} \tag{11}$$

$$S = \frac{M_2}{M_1}, \qquad H = \frac{h_2}{h_1} \tag{12}$$

$$l_1 = \frac{Eh_1}{(1-\nu^2)\phi^2}, \qquad l_2 = \frac{Eh_1}{(1-\nu^2)\phi\eta}, \qquad l_3 = \frac{Eh_1}{(1-\nu^2)\eta^2} \tag{13}$$

$$b = \sqrt{\frac{h_1}{\Lambda k_B T}} \tag{14}$$

$$\tau = \frac{h_1}{M_1(k_B T)^2} \tag{15}$$

Only the *Q*'s, *S*, and *H* after scaling need to be assigned values for simulations. Putting in additional parameters such as *E* will be unnecessary because we will eventually calculate *Q* anyways.

Adding Temperature Effects:

Eq. 5 and Eq. 6 assume a constant temperature. However, it is obvious that temperature fluctuations cause chemicals to behave differently. This is an important aspect that we plan to model, and these equations have to be modified to include temperature changes during a simulation. The terms we add in are based on experimental data and observations. According to Anderson and Crerar (1993), $\overline{g}$ (*see* Eq. 3) is a linear function of temperature. First we change the equations slightly. In Eq. 3, instead of having T multiply to the entire equation, we only multiply it to the ideal mixing terms (the logarithmic terms). A new T_0 is introduced and is multiplied to the rest of the equation (the non-ideal mixing terms). We then multiply the terms containing Ω_{ab}^0 and Ω_{ab}^1 by $1+\alpha_{ab}(T_0-T)$, where α_{ab} is a constant. For example, $C_1C_2\left[\Omega_{12}^0+\Omega_{12}^1(C_1-C_2)\right]$ becomes $C_1C_2\left[\Omega_{12}^0+\Omega_{12}^1(C_1-C_2)\right]\left(1+\alpha_{12}(T_0-T)\right)$. The new $\overline{g}$ that incorporates temperature is

$$\begin{aligned}\overline{g}(C_1,C_2)=\Lambda k_B T_0\Bigg\{&\frac{T}{T_0}\left[C_1\ln C_1+C_2\ln C_2+(1-C_1-C_2)\ln(1-C_1-C_2)\right]+\\&C_1C_2\left[\Omega_{12}^0+\Omega_{12}^1(C_1-C_2)\right]\left(1+\alpha_{12}(T_0-T)\right)+\\&C_1(1-C_1-C_2)\left[\Omega_{13}^0+\Omega_{13}^1(2C_1+C_2-1)\right]\left(1+\alpha_{13}(T_0-T)\right)+\\&C_2(1-C_1-C_2)\left[\Omega_{23}^0+\Omega_{23}^1(C_1+2C_2-1)\right]\left(1+\alpha_{23}(T_0-T)\right)\Bigg\}\end{aligned}\qquad(16)$$

Using this new equation, P_1 and P_2 (Eq. 9 and Eq. 10) become

$$P_1(C_1,C_2)=\frac{1}{\Lambda k_B T_0}\frac{\partial \overline{g}}{\partial C_1}$$
$$=\frac{T}{T_0}\ln\left(\frac{C_1}{1-C_1-C_2}\right)+C_2\left[\Omega_{12}^0+\Omega_{12}^1(2C_1-C_2)\right]\left(1+\alpha_{12}(T_0-T)\right)-$$
$$C_2\left[\Omega_{23}^0+\Omega_{23}^1(2C_1+3C_2-2)\right]\left(1+\alpha_{23}(T_0-T)\right)+$$
$$\left[\Omega_{13}^0(1-2C_1-C_2)+\Omega_{13}^1\left(6C_1+2C_2-6C_1^2-C_2^2-6C_1C_2-1\right)\right]\left(1+\alpha_{13}(T_0-T)\right) \quad (17)$$

$$P_2(C_1,C_2)=\frac{1}{\Lambda k_B T_0}\frac{\partial \overline{g}}{\partial C_2}$$
$$=\frac{T}{T_0}\ln\left(\frac{C_2}{1-C_1-C_2}\right)+C_1\left[\Omega_{12}^0+\Omega_{12}^1(C_1-2C_2)\right]\left(1+\alpha_{12}(T_0-T)\right)-$$
$$C_1\left[\Omega_{13}^0+\Omega_{13}^1(3C_1+2C_2-2)\right]\left(1+\alpha_{13}(T_0-T)\right)+$$
$$\left[\Omega_{23}^0(1-C_1-2C_2)+\Omega_{13}^1\left(2C_1+6C_2-C_1^2-6C_2^2-6C_1C_2-1\right)\right]\left(1+\alpha_{13}(T_0-T)\right) \quad (18)$$

Temperature changes also affect the rate at which the chemicals diffuse; thus, M_1 and M_2 must be a function of temperature. According to experimental results (Kaganovskii et al., 1998), this rate increases exponentially with temperature. To capture this effect this, we multiply M_1 and M_2 by $e^{-\frac{\Delta E}{R}\left(\frac{1}{T}-\frac{1}{T_0}\right)}$, where ΔE is the activation energy (kilojoules per mole) and R is the ideal gas constant (8.314 joules per mole Kelvin). The final modification we make is to Young's modulus E, the stiffness. The experimental results of Jeong et al. (2003) show that Young's modulus decreases linearly as temperature increases. So we multiply it by $1+\beta(T_0-T)$. This constant divides all of the Q's after scaling. In general, after scaling and transforming, the temperature constants will remain unchanged.

Numerical Solution

The set of integral-differential equations Eq. 5 and Eq. 6 are impossible to solve analytically. However, we can use the Fourier Transform to simplify them enough so that they can be solved numerically using a semi-implicit method. First, initial and boundary conditions must be given in order to solve these equations. The initial condition is the beginning pattern created by the user. Two possible initial conditions are considered: homogeneous and heterogeneous (e.g. certain areas have higher concentrations). For boundary condition, we let both concentrations to be zero at infinity, that is, $\begin{matrix} C(-\infty, x_2, t) = 0, & C(\infty, x_2, t) = 0 \\ C(x_1, -\infty, t) = 0, & C(x_1, \infty, t) = 0 \end{matrix}$. This convention is useful when we transform the equations. In addition to these, we also let the successive derivatives (up to the third order) to be zero at infinity[1]. Again, these help in the Fourier transformations.

Let the Fourier Transformation be defined as

$$\widehat{C}(k_1, k_2, t) = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} C(x_1, x_2, t) e^{-i(k_1 x_1 + k_2 x_2)} dx_1 dx_2 \tag{19}$$

The $\frac{1}{2\pi}$ can be included however it is unnecessary because it will eventually drop out and will only act as minor scaling factor if we do include it.

---

[1] $\begin{matrix} \frac{\partial^n C}{\partial x_1^n}(-\infty, x_2, t) = 0, & \frac{\partial^n C}{\partial x_1^n}(\infty, x_2, t) = 0 \\ \frac{\partial^n C}{\partial x_2^n}(x_1, -\infty, t) = 0, & \frac{\partial^n C}{\partial x_2^n}(x_1, \infty, t) = 0 \end{matrix}$
$n = 1,2,3$

Transformation of the Time Derivative:

$$\int_{-\infty-\infty}^{\infty}\int^{\infty}\frac{\partial C}{\partial t}e^{-i(k_1x_1+k_2x_2)}dx_1dx_2=\frac{\partial}{\partial t}\int_{-\infty-\infty}^{\infty}\int^{\infty}C(x_1,x_2,t)e^{-i(k_1x_1+k_2x_2)}dx_1dx_2=\frac{\partial\widehat{C}}{\partial t} \tag{20}$$

Transformation of the Laplacian:

$$\nabla^2C=\frac{\partial^2C}{\partial x_1^2}+\frac{\partial^2C}{\partial x_2^2} \tag{21}$$

We here only show the transformation of the first term since the transformation of the second one follows the same procedure.

$$\begin{aligned}\int_{-\infty-\infty}^{\infty}\int^{\infty}\frac{\partial^2C}{\partial x_1^2}e^{-i(k_1x_1+k_2x_2)}dx_1dx_2&=\int_{-\infty}^{\infty}\left[e^{-i(k_1x_1+k_2x_2)}\frac{\partial C}{\partial x_1}+ik_1\int\frac{\partial C}{\partial x_1}e^{-i(k_1x_1+k_2x_2)}dx_1\right]_{-\infty}^{\infty}dx_2\\&=ik_1\int_{-\infty-\infty}^{\infty}\int^{\infty}\frac{\partial C}{\partial x_1}e^{-i(k_1x_1+k_2x_2)}dx_1dx_2\\&=ik_1\int_{-\infty}^{\infty}\left[e^{-i(k_1x_1+k_2x_2)}C(x_1,x_2,t)+ik_1\int C(x_1,x_2,t)e^{-i(k_1x_1+k_2x_2)}dx_1\right]_{-\infty}^{\infty}dx_2\\&=-k_1^2\int_{-\infty-\infty}^{\infty}\int^{\infty}C(x_1,x_2,t)e^{-i(k_1x_1+k_2x_2)}dx_1dx_2\\&=-k_1^2\widehat{C}(k_1,k_2,t)\end{aligned} \tag{22}$$

Here we use integration by parts[2] twice and the boundary conditions, which are $e^{-i(k_1x_1+k_2x_2)}\frac{\partial C}{\partial x_1}\Big|_{-\infty}^{\infty}=0$ and $e^{-i(k_1x_1+k_2x_2)}C(x_1,x_2,t)\Big|_{-\infty}^{\infty}=0$. Using a similar procedure, we can obtain the transformation of the second term, which is $-k_2^2\widehat{C}(k_1,k_2,t)$. Adding these two transformations together, we have $\int_{-\infty-\infty}^{\infty}\int^{\infty}\nabla^2Ce^{-i(k_1x_1+k_2x_2)}dx_1dx_2=-k^2\widehat{C}(k_1,k_2,t)$, where $k=\sqrt{k_1^2+k_2^2}$. Note that this procedure also applies to $P(x_1,x_2,t)$, which becomes

[2] $\int f(x)g'(x)dx=f(x)g(x)-\int f'(x)g(x)dx$

$-k^2\widehat{P}(k_1,k_2,t)$. This method can be extended to any order derivative since it only requires integration by parts and the boundary conditions. Therefore, $\nabla^4 C$ becomes $k^4\widehat{C}$).

Transformation of the Double Integration Term:

The transformation of the double integral terms in Eq. 7 and Eq. 8 is adopted from Hu et al. (2007). This transformation involves writing the double integral as a convolution and using the fact that the Fourier transformation of a convolution is the product of the Fourier transformation of each function[3] (Convolution Theorem).

$$\iint \frac{(x_1-\xi_1)\frac{\partial C}{\partial \xi_1}+(x_2-\xi_2)\frac{\partial C}{\partial \xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}\,d\xi_1 d\xi_2 = \iint \frac{(x_1-\xi_1)\frac{\partial C}{\partial \xi_1}\,d\xi_1 d\xi_2}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}} + \iint \frac{(x_2-\xi_2)\frac{\partial C}{\partial \xi_2}\,d\xi_1 d\xi_2}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}} \tag{23}$$

Again we here only show the transformation of the first term. Let $\rho=\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{-1/2}$. Taking the partial derivative of ρ with respect to ξ_1, we have $\frac{\partial\rho}{\partial\xi_1}=\frac{(x_1-\xi_1)}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}$. Substituting it into the first term of Eq. 23, we obtain $\iint\frac{\partial\rho}{\partial\xi_1}\frac{\partial C}{\partial\xi_1}d\xi_1 d\xi_2$, which is the convolution of the partial derivatives of $\rho=\left(x_1^2+x_2^2\right)^{-1/2}$ and $C(x_1,x_2,t)$ (t is constant) with respect to x_1, or $\iint\frac{\partial\rho}{\partial\xi_1}\frac{\partial C}{\partial\xi_1}d\xi_1 d\xi_2=-\frac{\partial\rho}{\partial x_1}*\frac{\partial C}{\partial x_1}$. Using the Convolution Theorem and the method for transforming spatial derivatives (previous transformation), we obtain

[3] $F[f(x)*g(x)]=F[f(x)]\cdot F[g(x)]$

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\left(-\frac{\partial\rho}{\partial x_1}*\frac{\partial C}{\partial x_1}\right)e^{-i(k_1x_1+k_2x_2)}dx_1dx_2 = -\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{\partial\rho}{\partial x_1}e^{-i(k_1x_1+k_2x_2)}dx_1dx_2\int_{-\infty}^{\infty}\int_{-\infty}^{\infty}\frac{\partial C}{\partial x_1}e^{-i(k_1x_1+k_2x_2)}dx_1dx_2$$
$$= -ik_1\hat{\rho}\cdot ik_1\hat{C}$$
$$= k_1^2\hat{\rho}\hat{C} \tag{24}$$

Similarly, the transformation of the second term is $k_2^2\hat{\rho}\hat{C}$. Thus, the Fourier transformation of the double integral is the sum of the two individual transformations, or $k^2\hat{\rho}\hat{C}$. It can be shown that $\hat{\rho}=\frac{2\pi}{k}$, which produces $2\pi k\hat{C}$. Finally, using the method for transforming the Laplacian, we obtain $-2\pi k^3\hat{C}$.

In summary, we have

$$\begin{aligned}
&\frac{\partial C}{\partial t}\Rightarrow\frac{\partial\hat{C}}{\partial t}\\
&\nabla^2 C\Rightarrow -k^2\hat{C}\\
&\nabla^2 P\Rightarrow -k^2\hat{P}\\
&\nabla^4 C\Rightarrow k^4\hat{C}\\
&\iint\frac{(x_1-\xi_1)\frac{\partial C}{\partial\xi_1}+(x_2-\xi_2)\frac{\partial C}{\partial\xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}d\xi_1d\xi_2\Rightarrow 2\pi k\hat{C}\\
&\nabla^2\iint\frac{(x_1-\xi_1)\frac{\partial C}{\partial\xi_1}+(x_2-\xi_2)\frac{\partial C}{\partial\xi_2}}{\left[(x_1-\xi_1)^2+(x_2-\xi_2)^2\right]^{3/2}}d\xi_1d\xi_2\Rightarrow -2\pi k^3\hat{C}
\end{aligned} \tag{25}$$

Consequently, Eq. 5 and Eq. 6 become

$$\frac{\partial\hat{C}_1}{\partial t} = -k^2\hat{P}_1 - 2k^4\hat{C}_1 + 2k^3Q_1\hat{C}_1 + 2k^3Q_2\hat{C}_2 \tag{26}$$

$$\frac{\partial\hat{C}_2}{\partial t} = S\left(-k^2\hat{P}_2 - 2k^4H\hat{C}_2 + 2k^3Q_2\hat{C}_1 + 2k^3Q_3\hat{C}_2\right) \tag{27}$$

Semi-Implicit Method:

Eq. 26 and Eq. 27 can be solved using a semi-implicit method proposed by Chen and Shen (1998). This method treats the non-linear $\hat{P}$ terms explicitly and the linear $\hat{C}$

terms implicitly. First, let $\widehat{P}^n = \widehat{P}(k_1, k_2, t)$, $\widehat{C}^n = \widehat{C}(k_1, k_2, t)$, and $\widehat{C}^{n+1} = \widehat{C}(k_1, k_2, t + \Delta t)$ (with subscripts 1 and 2). Also, let $\frac{\partial \widehat{C}}{\partial t} = \frac{\widehat{C}^{n+1} - \widehat{C}^n}{\Delta t}$ (also with subscripts). Eq. 26 and Eq. 27 become

$$\frac{\widehat{C}_1^{n+1} - \widehat{C}_1^n}{\Delta t} = -k^2 \widehat{P}_1^n - 2k^4 \widehat{C}_1^{n+1} + 2k^3 Q_1 \widehat{C}_1^{n+1} + 2k^3 Q_2 \widehat{C}_2^{n+1} \tag{28}$$

$$\frac{\widehat{C}_2^{n+1} - \widehat{C}_2^n}{\Delta t} = S\left(-k^2 \widehat{P}_2^n - 2k^4 H \widehat{C}_2^{n+1} + 2k^3 Q_2 \widehat{C}_1^{n+1} + 2k^3 Q_3 \widehat{C}_2^{n+1}\right) \tag{29}$$

In matrix form, these equations combine as

$$\frac{1}{\Delta t}\begin{Bmatrix} \widehat{C}_1^{n+1} - \widehat{C}_1^n \\ \widehat{C}_2^{n+1} - \widehat{C}_2^n \end{Bmatrix} = \begin{bmatrix} -2k^4 + 2k^3 Q_1 & 2k^3 Q_2 \\ 2Sk^3 Q_2 & -2SHk^4 + 2Sk^3 Q_1 \end{bmatrix} \times \begin{Bmatrix} \widehat{C}_1^{n+1} \\ \widehat{C}_2^{n+1} \end{Bmatrix} - k^2 \begin{Bmatrix} \widehat{P}_1^n \\ S\widehat{P}_2^n \end{Bmatrix} \tag{30}$$

Solving for $\begin{Bmatrix} \widehat{C}_1^{n+1} \\ \widehat{C}_2^{n+1} \end{Bmatrix}$, we obtain

$$\begin{Bmatrix} \widehat{C}_1^{n+1} \\ \widehat{C}_2^{n+1} \end{Bmatrix} = \begin{bmatrix} 1 + \left(2k^4 - 2k^3 Q_1\right)\Delta t & -2k^3 Q_2 \Delta t \\ -2Sk^3 Q_2 \Delta t & 1 + S\left(2Hk^4 - 2k^3 Q_3\right)\Delta t \end{bmatrix}^{-1} \times \left[\begin{Bmatrix} \widehat{C}_1^n \\ \widehat{C}_2^n \end{Bmatrix} - k^2 \Delta t \begin{Bmatrix} \widehat{P}_1^n \\ S\widehat{P}_2^n \end{Bmatrix} \right] \tag{31}$$

Eq. 31 is in the form that can be implemented in a code. The inverse matrix can be found using a formula for 2x2 matrices[4]. With the temperature constants added in the previous section, Eq. 31 becomes (the key equation)

[4] $\begin{bmatrix} a & b \\ c & d \end{bmatrix}^{-1} = \frac{1}{ad - cb}\begin{bmatrix} d & -b \\ -c & a \end{bmatrix}$

$$\begin{Bmatrix} \widehat{C}_1^{n+1} \\ \widehat{C}_2^{n+1} \end{Bmatrix} = \left[\begin{array}{cc} 1+e^{-\frac{\Delta E}{R}\left(\frac{1}{T}-\frac{1}{T_0}\right)}\left(2k^4-\frac{2k^3Q_1}{1+\beta(T_0-T)}\right)\Delta t & -\frac{2e^{-\frac{\Delta E}{R}\left(\frac{1}{T}-\frac{1}{T_0}\right)}k^3Q_2\Delta t}{1+\beta(T_0-T)} \\ -\frac{2e^{-\frac{\Delta E}{R}\left(\frac{1}{T}-\frac{1}{T_0}\right)}Sk^3Q_2\Delta t}{1+\beta(T_0-T)} & 1+e^{-\frac{\Delta E}{R}\left(\frac{1}{T}-\frac{1}{T_0}\right)}S\left(2Hk^4-\frac{2k^3Q_3}{1+\beta(T_0-T)}\right)\Delta t \end{array} \right]^{-1} \times \left[\begin{Bmatrix} \widehat{C}_1^n \\ \widehat{C}_2^n \end{Bmatrix} - e^{-\frac{\Delta E}{R}\left(\frac{1}{T}-\frac{1}{T_0}\right)}k^2\Delta t \begin{Bmatrix} \widehat{P}_1^n \\ S\widehat{P}_2^n \end{Bmatrix} \right] \tag{32}$$

The concentrations are calculated as follows (Figure 3). In real space, we use C to calculate P. Then we transform C and P to find $\widehat{C}^n$ and $\widehat{P}^n$. In Fourier space we use $\widehat{C}^n$ and $\widehat{P}^n$ to calculate $\widehat{C}^{n+1}$. We transform $\widehat{C}^{n+1}$ into real space and repeat the same process.

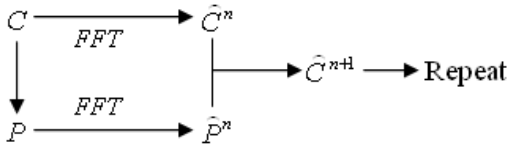


Figure 3: Procedure used to calculate the concentrations

Coordinates in Fourier Space:

The coordinates in Fourier space are not the same as those in real. Fourier space is made up of frequencies as it is also called the frequency domain/space. For a two dimensional set of data, the frequencies for each row (left to right) are

$$0, \frac{1}{N\Delta}, \frac{2}{N\Delta}, \ldots, \frac{1}{2\Delta}-\frac{1}{N\Delta}, \pm\frac{1}{2\Delta}, -\left(\frac{1}{2\Delta}-\frac{1}{N\Delta}\right), \ldots, -\frac{2}{N\Delta}, -\frac{1}{N\Delta}$$

The columns (top to bottom) follow the same sequence. The sign doesn't matter for the middle term. Δ is the sampling interval, which acts like a length scale. For example, $\Delta = 0.1$ could mean there are 0.1 nm per pixel.

Fast Fourier Transform (FFT)

If we could use Eq. 32, the calculations would be incredibly simple. Unfortunately, Eq. 32 is in Fourier space, which we can't intuitively see. In addition, we can't find a formula for the transformation of $P(x_1, x_2, t)$ because it is non-linear, that is, $\widehat{P}(k_1, k_2, t)$ can't be calculated directly in Fourier space. Also, we can't transform Eq. 32 back to real space using the Fourier transformation in the previous section because it is not in the right form.

The solution to this problem is the Fast Fourier Transformation (FFT). This is an efficient and fast algorithm for transforming data sets between real and Fourier space. The Discrete Fourier Transformation (DFT) is defined as

$$F(m_1, m_2) = \sum_{n_2=0}^{N_2-1} \sum_{n_1=0}^{N_2-1} f(n_1, n_2) e^{2\pi i n_1 m_1 / N_1} e^{2\pi i n_2 m_2 / N_2} \tag{33}$$

F is the transformation of the discrete data set *f*. One could simply put this into a computer and obtain the transformations. However, for large data sets, say 512x512, this method can take a *very* long time. For data sets of this size, the FFT algorithm is the best solution; in fact, calculations that would take days and possibly even weeks can be reduced to merely seconds and minutes, which is an enormous advantage.

Take the one dimensional DFT $F(m) = \sum_{n=0}^{N-1} f(n) e^{2\pi i n m / N}$ as an example[5]. This can be rewritten as

[5] The two dimensional DFT is a combination of two one dimension DFT's

$$
\begin{aligned}
F(m) &= \sum_{n=0}^{N/2-1} f(2n) e^{2\pi i m(2n)/N} + \sum_{n=0}^{N/2-1} f(2n+1) e^{2\pi i m(2n+1)/N} \\
&= \sum_{n=0}^{N/2-1} f(2n) e^{2\pi i m n/(N/2)} + e^{2\pi i m/N} \sum_{n=0}^{N/2-1} f(2n+1) e^{2\pi i m n/(N/2)} \\
&= F^e(m) + W^m F^o(m)
\end{aligned} \tag{34}
$$

Thus F can be written as the DFT of the even indices plus a complex constant[6] times the DFT of the odd indices. $F^e(m)$ and $F^o(m)$ are periodic with periods of length $N/2$, thus, $F^e(m) = F^e\left(m - \frac{N}{2}\right)$ and $F^o(m) = F^o\left(m - \frac{N}{2}\right)$ for $m \geq \frac{N}{2}$. F originally requires N^2 operations, but, by separating it into evens and odds, F now only requires $\frac{N^2}{2}$ operations, which is slightly faster. We continue breaking F down into smaller sets of size $N/4$ of evens and odds[7] and so on until we are left with sets of size 1. F eventually only requires $N \log_2 N$ operations, which is significantly faster for large N. In the end, we have a seemingly meaningless string of *e*'s and *o*'s. Actually, this seemingly meaningless string is extremely useful in finding which $f(n)$ goes with $F^{eeoeoo.....oeoe}(m)$. Take the string, reverse it, and let $e = 0$ and $o = 1$. What does this produce? It produces the binary representation of n (in $f(n)$) (Press et al.2002)! Thus, a faster way to break down F until there are N sets of size 1 left is by taking the binary representation of the indices of the initial set and "flipping" them. This is called bit reversing, the first part of the FFT.

[6] Note that W is not the same constant for each successive separation of F. For each successive separation after Eq. 34, the N in W is divided by two.

[7] $F^e(m) = F^{ee}(m) + W^m F^{eo}(m) \qquad W^m = e^{2\pi i m/(N/2)}$

Once we have bit reversed the initial set, we have to regroup everything. This method is called the butterfly method[8], which is also known as the Danielson-Lanzos Formula. The amazing aspect of this formula is that it is iterative. The butterfly method first takes two consecutive elements (after bit reverse) and combines them into a set of size two. Each element of the new set is calculated using a similar formula as that of Eq. 34. There are $N / 2$ such sets. Then, two consecutive sets are combined to create a new set of size four (one element of one set combines with an element of the other) using a similar formula. There are $N / 4$ such sets. This continues until you are left with one large set of size N, which is the transformation. The following is a diagram for combining the elements (size 8). The left side is already bit reversed.

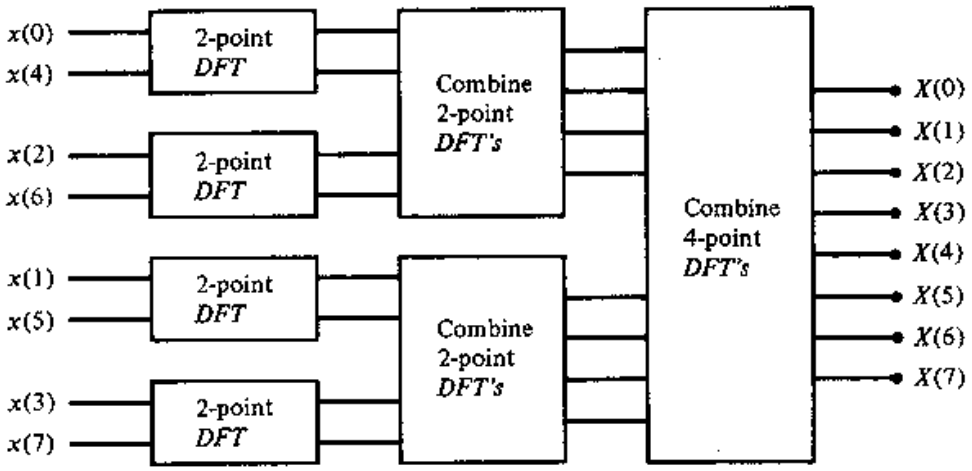
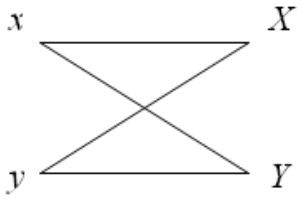


Figure 4: Method of combining elements after bit reversing.
http://www.cmlab.csie.ntu.edu.tw/cml/dsp/training/coding/transform/fft.html

[8] When this method is drawn out, parts of it are shaped like a butterfly.

The following is a simple 2 point DFT



$$
\begin{aligned}
&N = 2 \\
&F^e(1) = F^e(2) \\
&F^o(1) = F^o(2) \\
&X = F^e(1) + W^1 F^o(1) = x + \left(e^{2\pi i/2}\right)^1 y = x - y \\
&Y = F^e(2) + W^2 F^o(2) = F^e(1) + W^2 F^o(1) = x + \left(e^{2\pi i/2}\right)^2 y = x + y
\end{aligned}
\tag{35}
$$

The FFT cannot simulate an infinitive domain. To resolve this problem, the simulation is carried out in a square cell, which is replicated many times to cover the whole space (Lu and Kim, 2005).

Numerical Stability and Convergence Analysis

We have found that Eq. 32 is very sensitive to time step Δt and sampling interval Δ. We did numerical convergence test in order to choose appropriate Δt and Δ. See Appendix A for discussion.

Architecture of the Software

See Appendix C for screenshots of the program.

The main interface gives the user a broad overview of the program in addition to a choice of running a homogeneous or a heterogeneous simulation. Both options will go similar screens. For a heterogeneous simulation, you can create your own initial patterns.

In the heterogeneous parameter input screen, the user may change the default parameter values and select a self-created pattern file and/or parameter file they would like to use in the simulations. In addition, the user can change the size of the simulation square. When the user starts the simulation, the class BitmapConverter is called. This class reads the hex numbers (colors) from the picture file and creates an array storing the corresponding concentrations (e.g. white color (0x000000) implies $C = 0$). After this process, the simulation screen is called. This is where the concentrations are displayed. Here the user may choose to change the time step, the number of time steps to calculate before displaying an image, and the temperature.

The homogeneous parameter input screen is very similar except that the user does not have to input a pattern file. In addition, the class BitmapConverter is bypassed since there aren't any pattern files to read.

All of the calculations are done in the class FFTandCalc.

There is one more class called Parameters. This class stores all of the parameters from the input screen. This prevents transferring large amounts of data between classes and reduces the amount of clatter, making the program more organized and easier to debug.

This software is written in C# because the syntax is similar to Java but is a better language for intense computations. The following is a diagram of the architecture of our software.

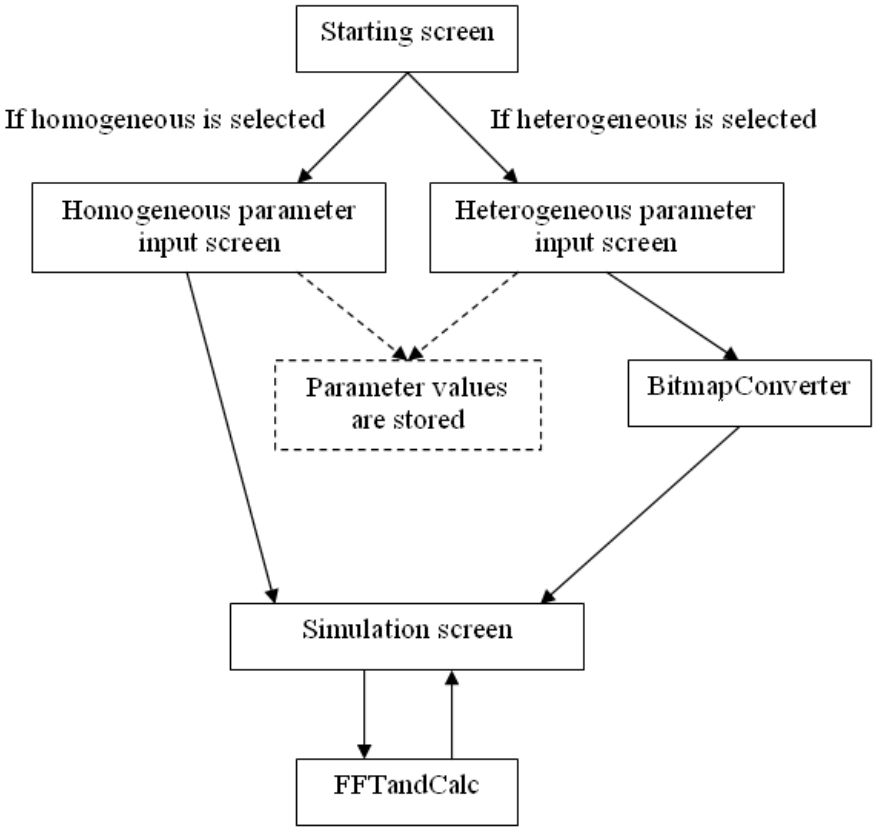


Figure 5: Architecture of our program

Results and Discussion

We have run many simulations after we finished the coding and have obtained many interesting results.

Homogeneous Simulations:

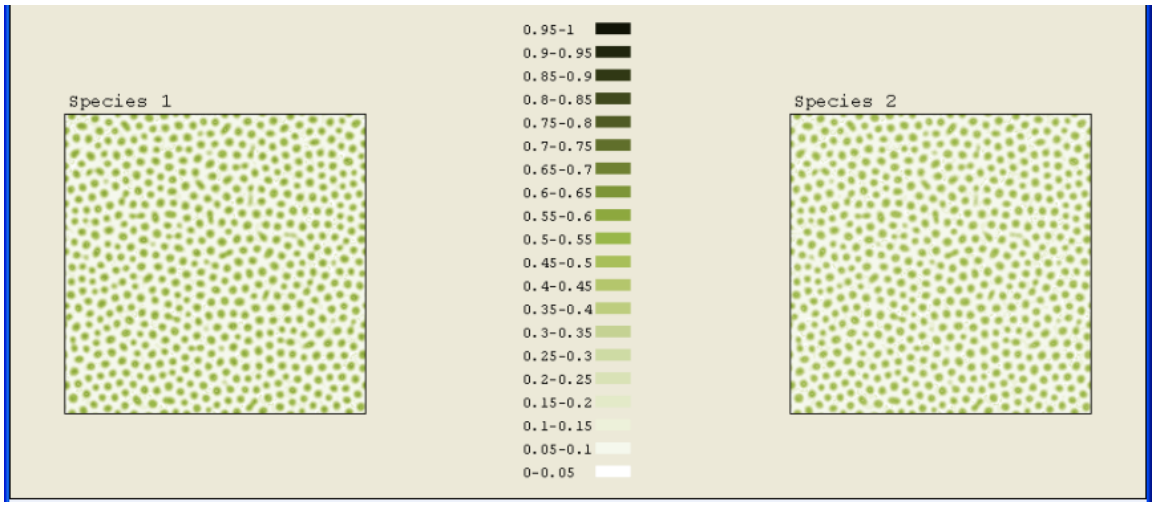
As observed experimentally, for low concentrations, chemicals tend to rearrange themselves into quantum dots. This qualitatively agrees with our simulations as shown in Figure 6A. This simulation starts with low concentrations: $C_1 = 0.15$ and $C_2 = 0.1$. A total scaled time of 0.1 has elapsed in this simulation. Figure 6A shows a nice array of quantum dots with diameters of approximately 3 nm.

As the concentrations of both components increase, the quantum dots become serpentine stripes in Figure 6B. The initial concentrations used in this simulation are $C_1 = 0.25$ and $C_2 = 0.15$. A total scaled time of 0.1 has elapsed.

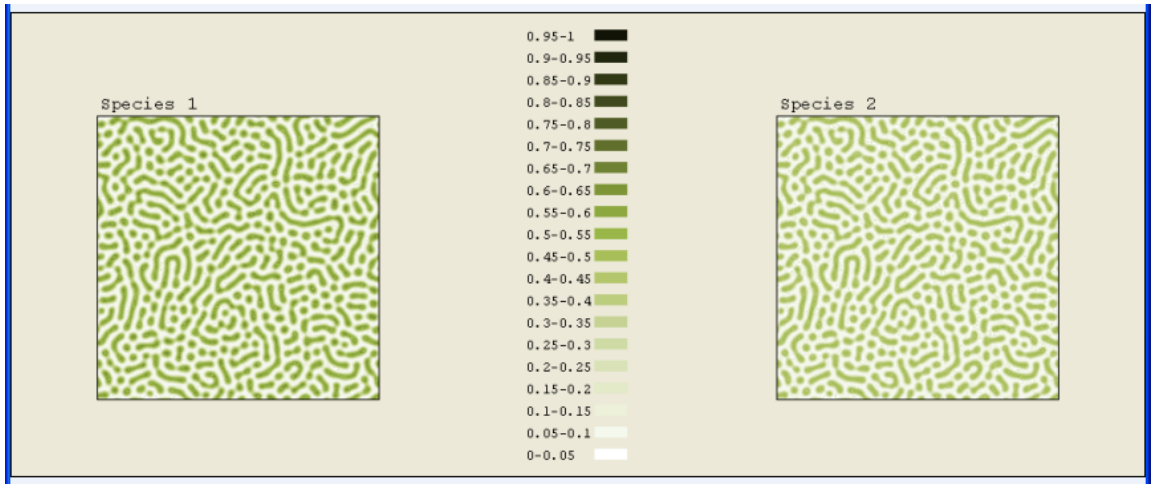
As the concentrations continue to increase, the serpentine stripes are replaced by quantum pits, which are the opposite of quantum dots (*see* Figure 6C). For this simulation, $C_1 = 0.4$ and $C_2 = 0.35$. Again, a total scaled time of 0.1 has elapsed. These transitions from quantum dots to serpentine stripes, and to quantum pits, agree qualitatively with experimental observation in Figure 1.

For the above three simulations, we use parameters $Q_1 = 2$, $Q_2 = 2$, $Q_3 = 2$, $S = 1.0$, $H = 1.0$, $\Omega_{12}^0 = 5$, $\Omega_{12}^1 = 0$, $\Omega_{13}^0 = 5$, $\Omega_{13}^1 = 0$, $\Omega_{23}^0 = 5$, $\Omega_{23}^1 = 0$, *Initial Temperature* = 400, *Percent Perturbation* = 1, $\Delta = 0.2$. The patterns shown in Figure 6 are the main results that we are able to obtain. For concentrations higher than those above, numerical error produces strange effects. We will not show any of those simulations here because they can be misleading.

A



B



C

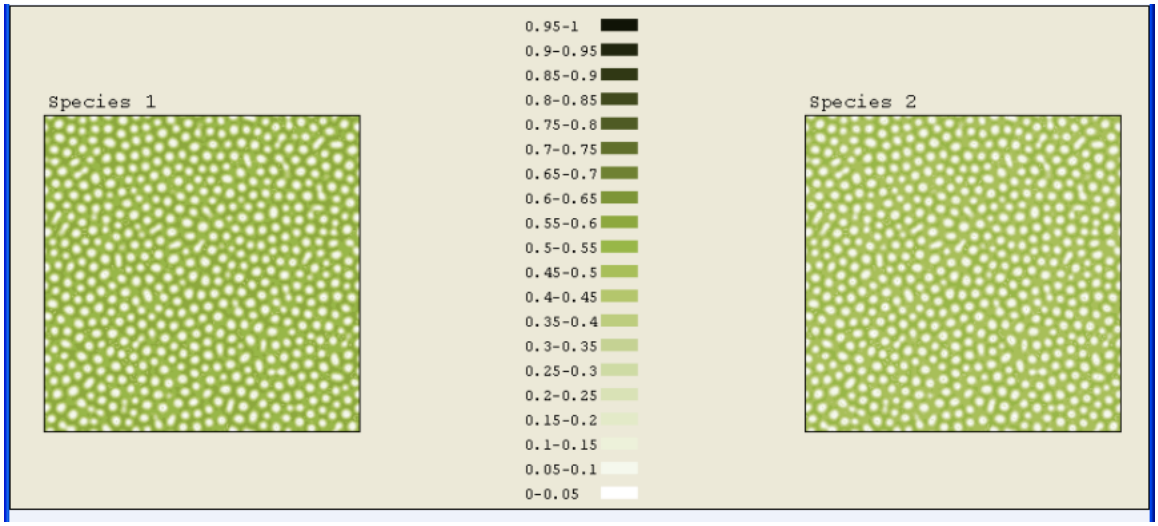


Figure 6: Transition from quantum dots (A) to serpentine stripes (B) and to quantum pits (C) as the concentration increase. The length of the simulations square is 51 nm.

Heterogeneous Simulations:

Heterogeneous simulations have a much wider range of results compared to the homogeneous simulations. For each initial pattern, there is a different result. Thus, heterogeneous simulations are more interesting to study. See Appendix B for more simulations.

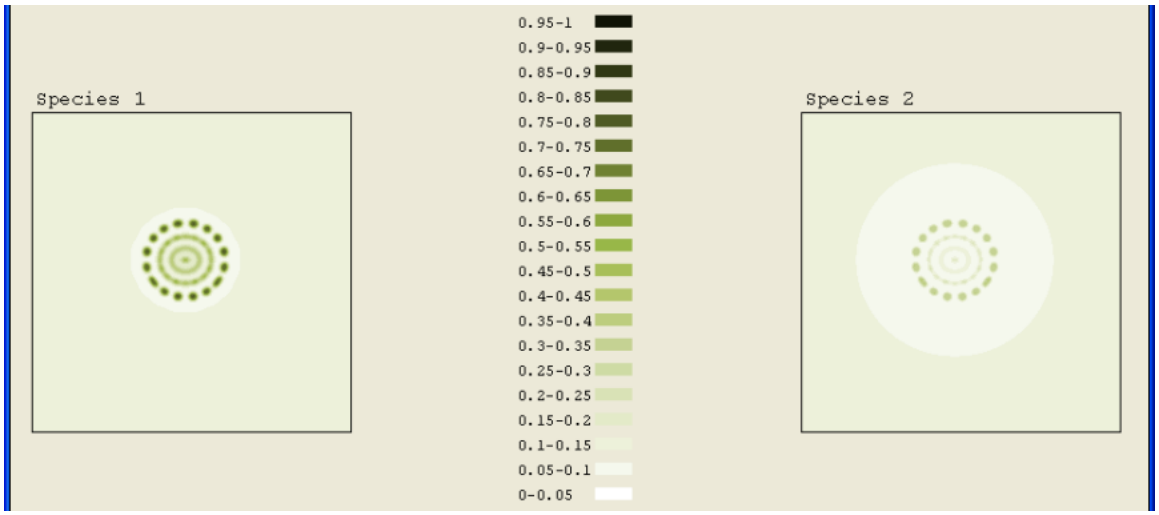
The initial pattern of Figure 7A was a circle of concentration .2 to .25. Outside the circle, the concentration was 0.05. This image was taken after a total scaled time of 1 elapsed. The interesting part of simulation is that it originally formed several rings before quantum dots appeared. If we were to run this simulation even longer, all of the rings would become quantum dots arranged in circles. The ring formation is analogous to the ripples created by dropping a rock in water.

Initially in Figure 7B, half of the simulation square had a concentration of 0.3 to 0.35. The other half had a concentration of 0.05. First, a high concentration stripe formed at the boundary. Gradually over time, more stripes formed. Not much happened to the lower concentration aside from the slight increase caused by the higher concentration spreading out. The formation of the stripes was expected. In this simulation, a total time of 0.9 elapsed.

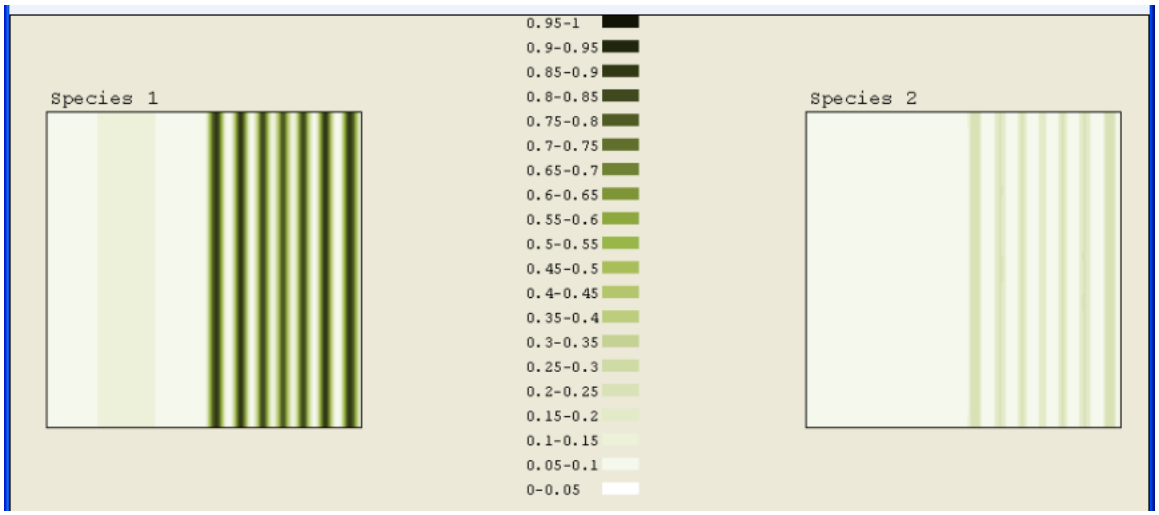
The initial pattern in Figure 7C was a square of concentration 0.4 to 0.45. A total scaled time of 0.9 elapsed. This simulation was not anything like the circle. One might expected it to spread out like the circle. This likely didn't happen due to the configuration (e.g. the sharp edges). Comparing the simulations of a square, a pentagon, and a circle, we can infer that as the number of sides on the polygon increases, the closer the simulation mimics the circle (it spreads out and forms rings), which makes sense.

For the circle and half space simulations, we used $Q_1 = 1$, $Q_2 = 1$, $Q_3 = 1$, $S = 1.0$, $H = 1.0$, $\Omega_{12}^0 = 4$, $\Omega_{12}^1 = 0$, $\Omega_{13}^0 = 4$, $\Omega_{13}^1 = 0$, $\Omega_{23}^0 = 4$, $\Omega_{23}^1 = 0$, *Initial Temperature* = 400, *Percent Perturbation* = 1, $\Delta = 0.2$. For the square simulation, $Q_1 = 2$, $Q_2 = 2$, and $Q_3 = 2$. As you can see, each simulation is drastically different. Thus, you can create many different patterns by different preexisting patterns, which is important for building nano devices.

A



B



C

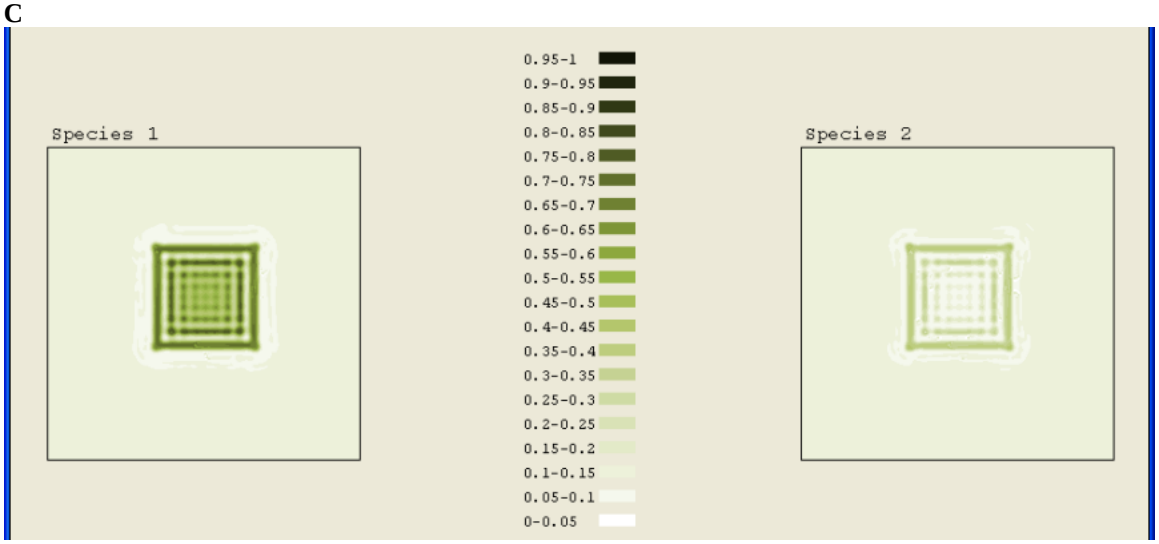


Figure 7: **Pattern formation guided by preexisting patterns. (A) Initial pattern: Circle of concentration .2-.25. (B) Initial pattern: Half of simulation square with concentration 0.3-0.35. (C) Initial pattern: Square of concentration .4-.45. The length of the simulation square is 51 nm.**

Temperature Simulations:

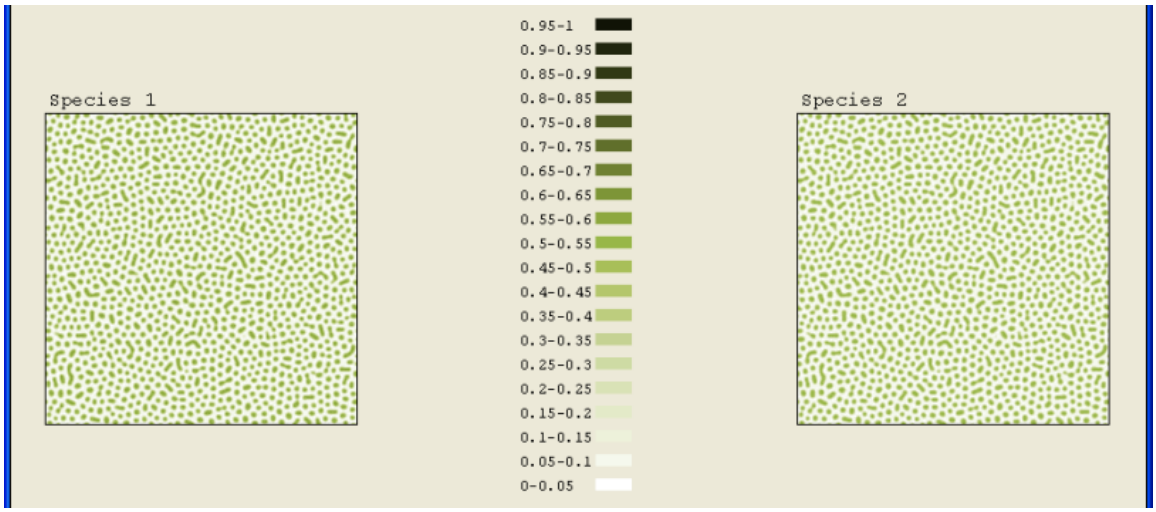
While heterogeneous simulations have many possible outcomes, the ability to control the effects of temperature opens even more possibilities (e.g. controlling the size of patterns). The physical and chemical properties of quantum dots depend on their size (Roduner, 2006). As temperature increases, the chemicals tend to diffuse more as the temperature term multiplied to M suggests and also the mixing of two chemical components becomes more ideal, that is, everything will become homogeneous (spread out). The following three simulations accurately show these expectations. These three simulations are parts of one large simulation. We use the same parameters in Figure 6 with $\alpha_{12} = 0.025$, $\alpha_{12} = 0.025$, $\alpha_{12} = 0.025$, $\beta = 0.0001$, and *Activation Energy* = 20.

In Figure 8A, we run a total scaled time of .1 with a temperature of 300 K (100 K lower than the previous 6 simulations). As we can see, the patterns are finer in size and sharper at the edges because the diffusion decreases and there is more non-ideal mixing.

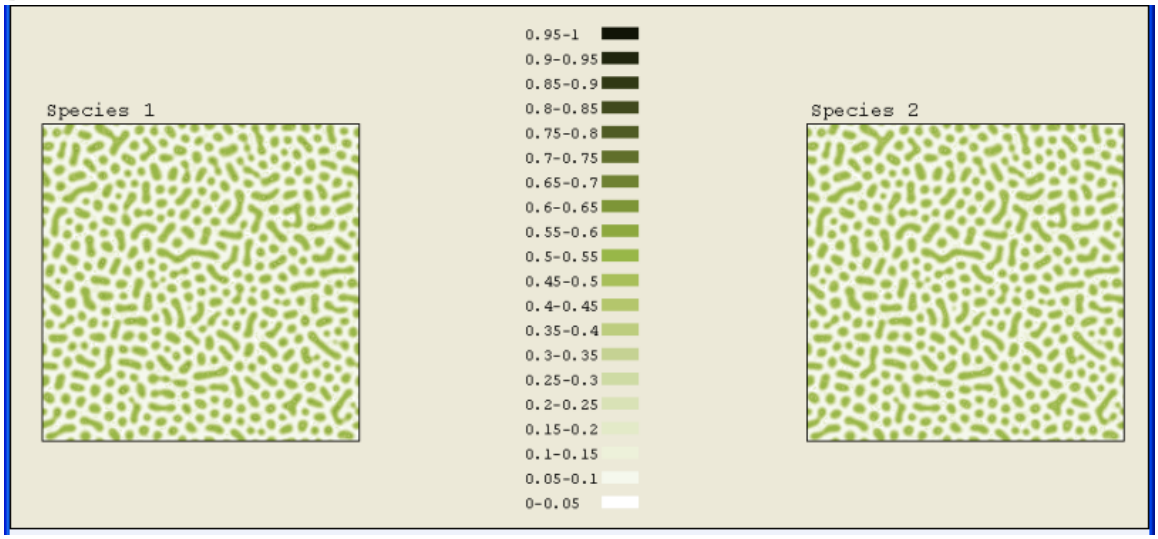
As the temperature increases, we expect the patterns to coarsen (get thicker). Figure 8B is at a scaled time of .2 with a temperature of 400 K (the temperature was changed immediately after .1). The pattern clearly coarsens as a result of the increase of diffusion.

Once the temperature reaches a certain point, the chemicals diffuse so much and the mixing becomes almost ideal, and there is no definite pattern. As we can see in Figure 8C, which is run at 500 K and has a total scaled time of .3, everything became almost homogenized. Therefore, temperature can be used to control the size of the patterns.

A



B



C

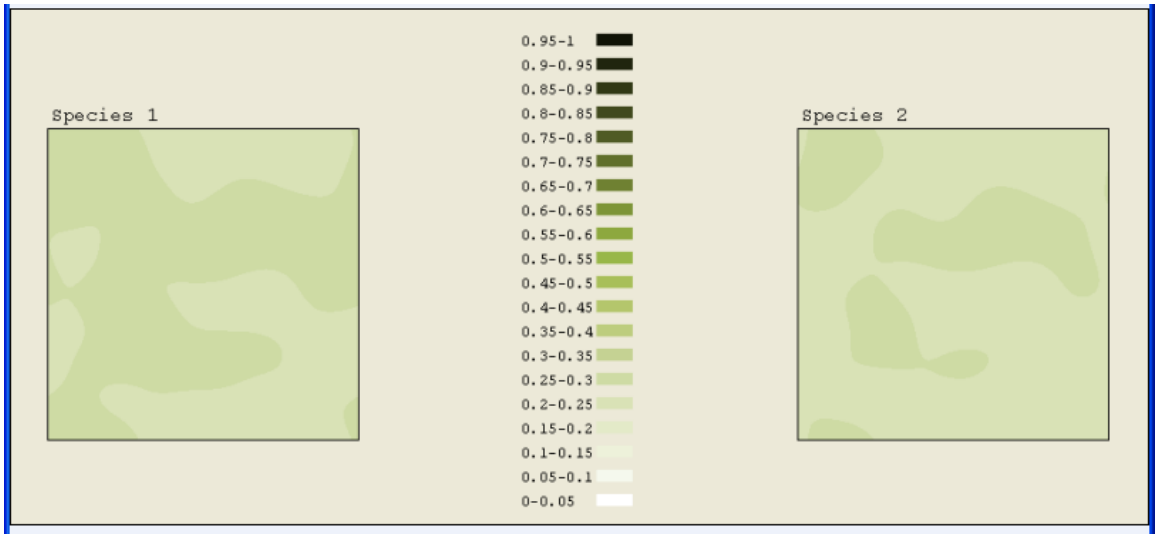


Figure 8: Using temperature to control the size of patterns. The length of the simulation square is 51 nm. (A) Simulation at 300 K. (B) Simulation at 400 K. (C) Simulation at 500 K.

Conclusions

Nanotechnology today is a hot topic. Building devices at a nanoscale has many applications. Nanoscale patterns can be formed by self-assembly. When some chemicals are deposited over an elastic substrate, they rearrange themselves into patterns to achieve the lowest possible energy. Two major factors cause this pattern formation. The minimums in Gibb's free energy drive the phase separation of the chemical components. This separation increases the amount surface free energy. To minimize its total energy, the system reacts by reducing the number of phase boundaries. On the other hand, the surface stress produced by concentration variations tends to create finer patterns by increasing the number of phase boundaries. These two opposing factors cause the system to eventually form a stable pattern.

We have successfully written a program in C# the can simulate pattern formations. Our simulations agree qualitatively with experimental observations and our expectations, which validate our program. In this program, we have successfully implemented the FFT, which was originally written for C++.

Our program has simulated (1) the transitions from quantum dots to serpentine stripes and to quantum pits as the concentration increases; (2) the dependency of heterogeneous pattern formation on preexisting patterns; (3) and the effect of temperature changes on the size of the patterns formed, which is important because the physical and chemical properties of these patterns (e.g. quantum dots) depends on size.

In the future we plan to include other mechanisms to control the pattern formations. Two such mechanisms are electric and magnetic fields. In addition, we will try to solve the equations using a fully implicit method, which will hopefully improve the

numerical stability. We will also look at ways to make our code even more efficient and faster, which will help our studies.

Acknowledgements

We would like to acknowledge our mentor, Dr. Yifeng Wang, for his help by providing us with resources and supporting us along the way. We would like to thank Dr. David Metzler for his help with the math. We would also like to thank Mr. Jim Mims for his support. This has been a really big project and their support is invaluable.

References

Anderson, Greg M., and David A. Crerar. "Solid Solution." Thermodynamics in Geochemistry. Oxford: Oxford University Press, 1993. 364-396.

Chung, Sung-hoon, et al. "Evaluation of Elastic Properties and Temperature Effects in Si Thin Films Using an Electrostatic Microresonator." Journal of Microelectromechanical Systems 122.4 (Aug. 2003): 524-530. 25 Mar. 2008 <http://ieeexplore.ieee.org>.

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/>.

Pelesko, John A., and David H. Bernstein. "Microfluids." Modeling MEMS and NEMS. New York: CRC Press, 2003. 297-314.

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>.

Press, William H., Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. "Fast Fourier Transform." Numerical Recipes in C++. 2nd ed. Cambridge: Cambridge University Press, 2002. 501-541.

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

Appendix A: Numerical Stability and Convergence Analysis

Because we are solving the equations numerically, we have to ensure that we aren't getting large numerical errors. Eq. 32 is sensitive to the choice of time step. By trial and error, we find that the time steps, 0.0001-0.001, are appropriate for the simulations.

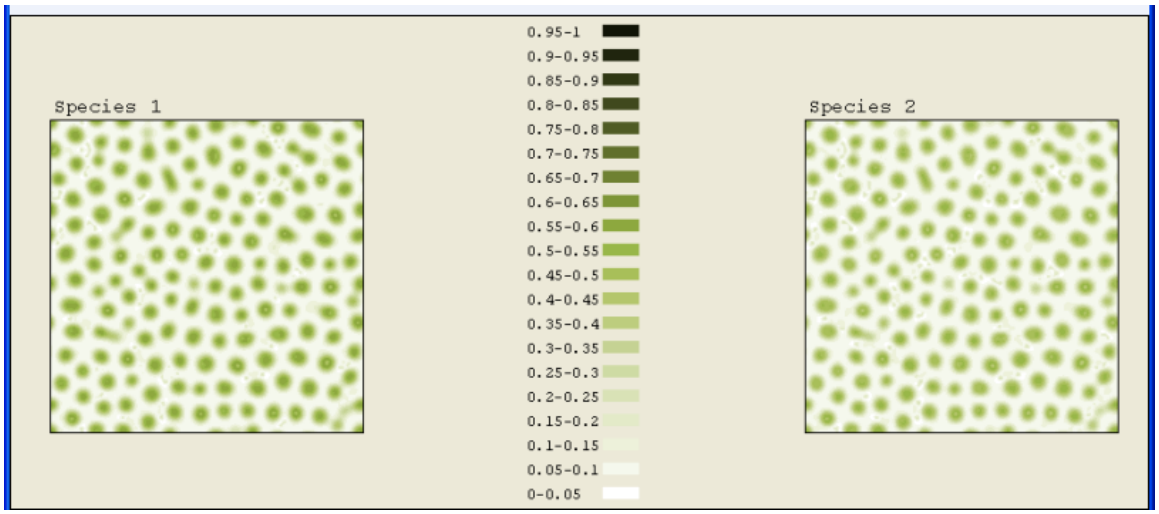
Figure 9A uses a time step of 0.0001. A total scaled time of 0.1 elapses. It appears that there are very few additional "things" (in the white areas) created by numerical instability.

Figure 9B is a continuation of Figure 9A with a time step of 0.001. Clearly it looks similar to Figure 9A, that is, we can still see the quantum dots. However, there is clearly a lot more numerical instability. Random concentrations appear in the white areas and some of the quantum dots have holes. Despite these, this simulation isn't that different from Figure 9A.

Figure 9C, however, is definitely different. We use a time step of 0.01 for this simulation.

Thus, from these simulations, we see that 0.0001-0.001 is good choice for time step. In addition to time step, Δx must also be checked. We will not show any simulations here. We have run many simulations with different Δx and have concluded that when Δx is around 0.05 to 0.6, the simulations are reasonable. We have simulated quantum dots with Δx within this range and have found that the results are practically identical. The parameters are similar to those of Figure 6.

A



B

C

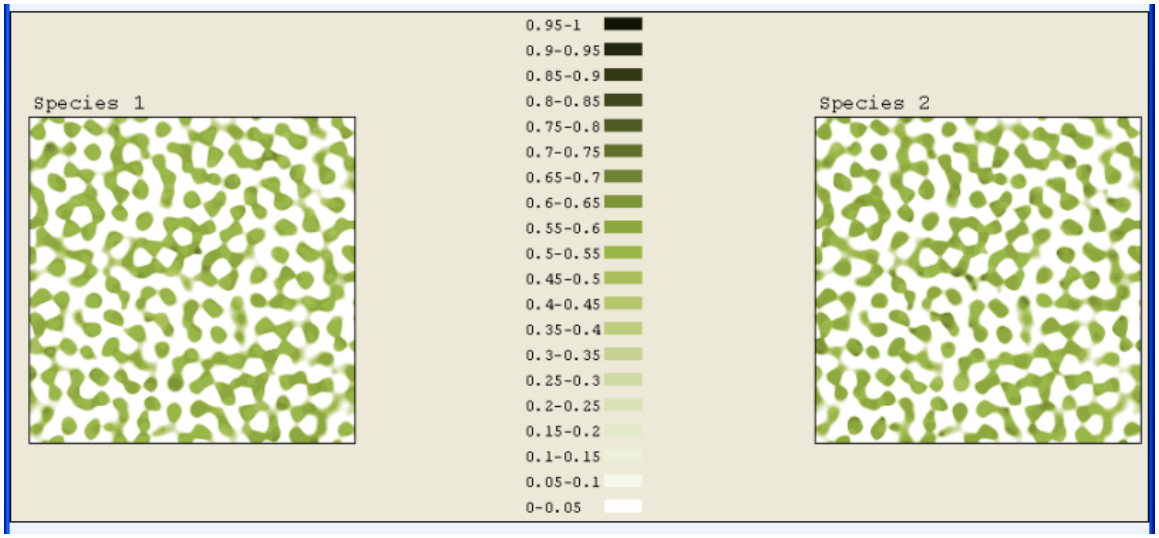
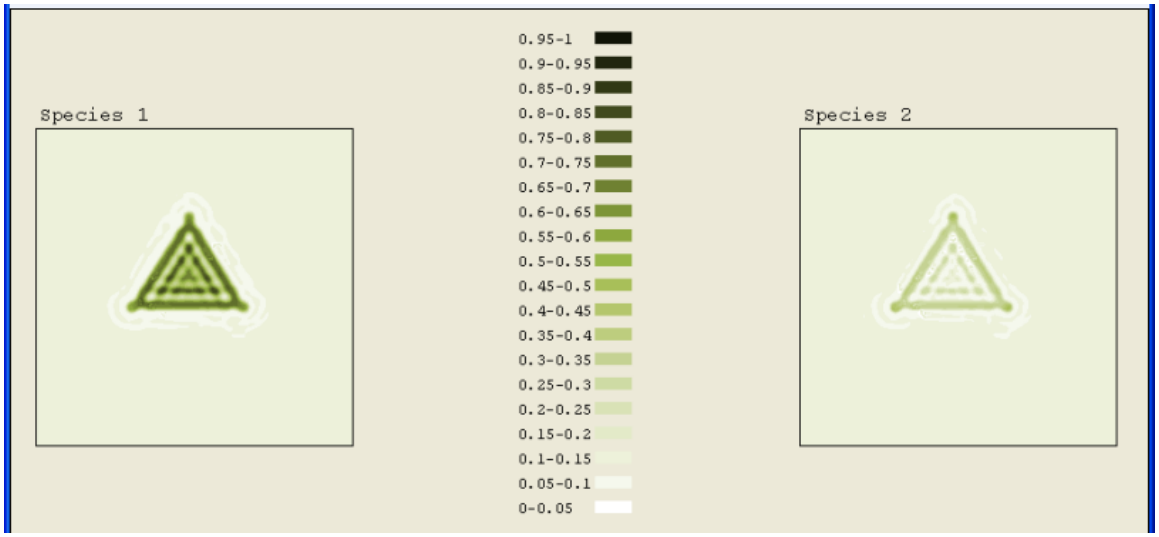


Figure 9: Numerical stability test. (A) 0.0001 time step. (B) 0.001 time step. (C) 0.01 time step. Same scale as before (51 nm).

Appendix B : More Simulations

More heterogeneous simulations are shown in Figure 10 (below). The parameters are similar to those in the previous heterogeneous simulations.

A



B

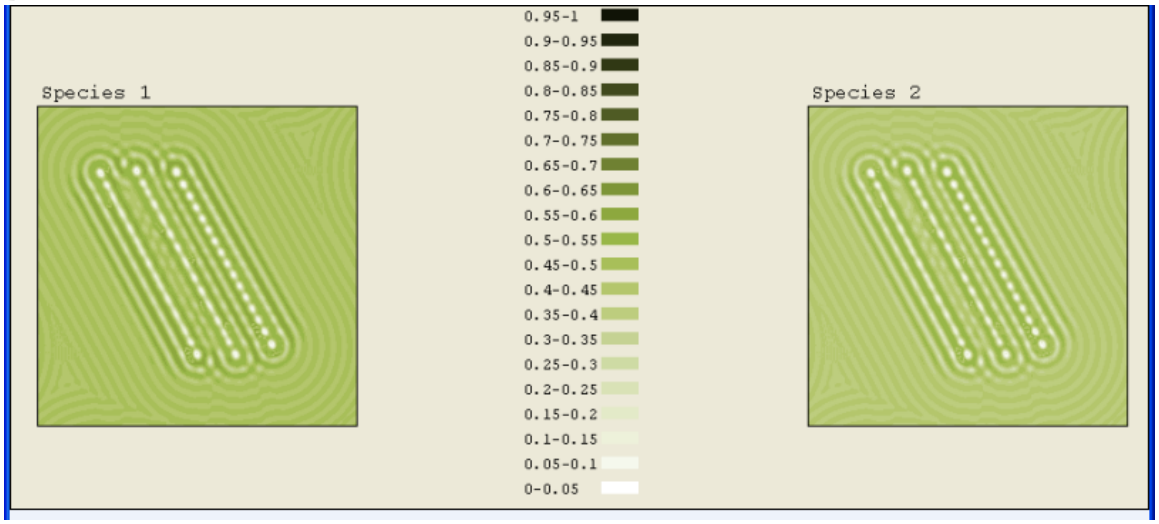
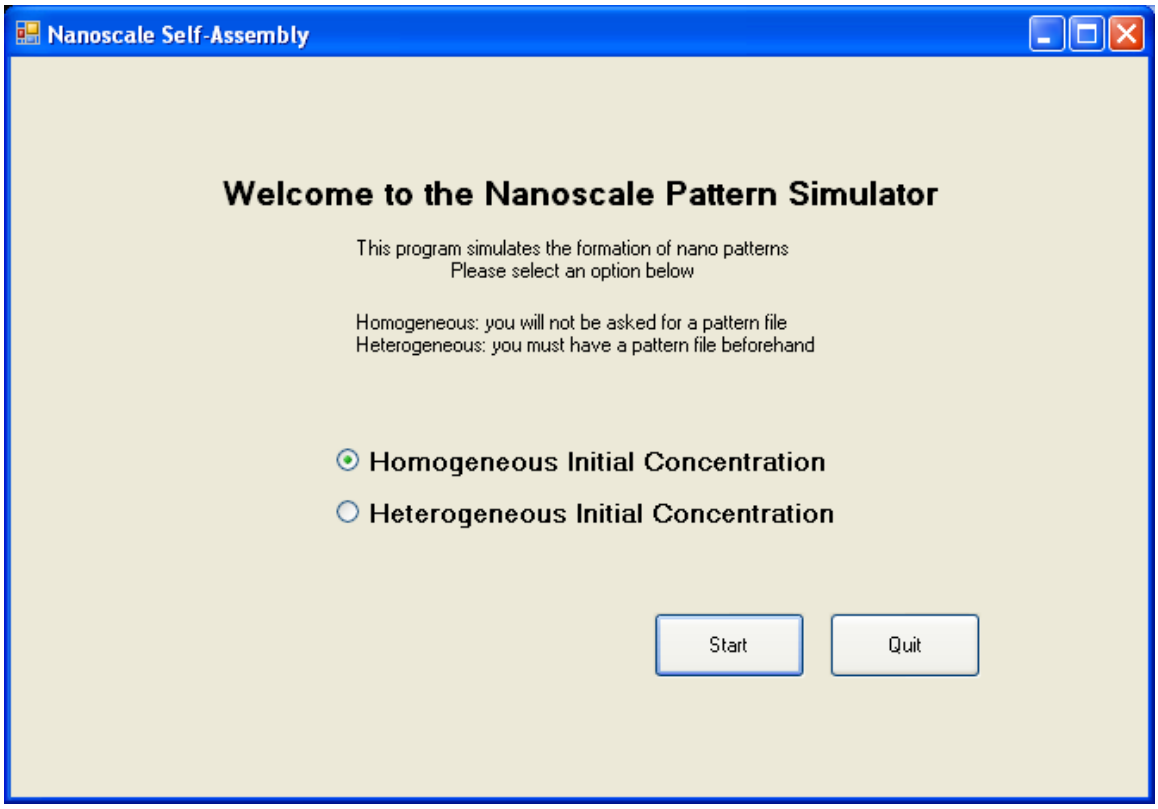


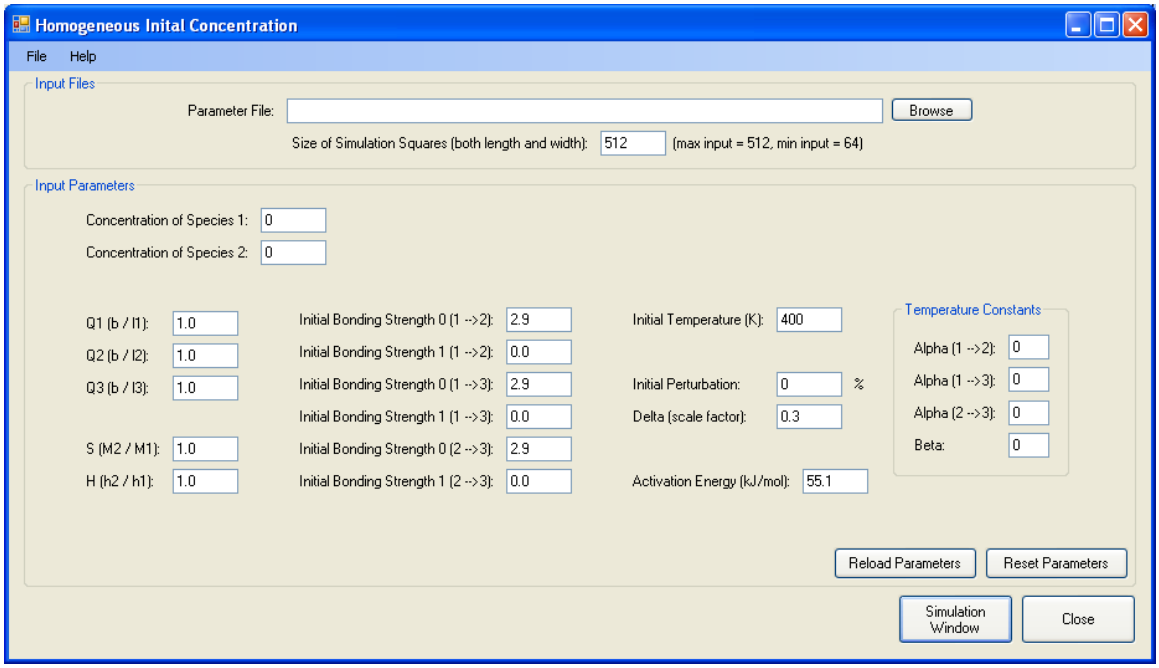
Figure 10: More heterogeneous simulations. (A) The initial pattern was a triangle of concentration 0.5-0.55 and the rest was 0.05. A total scaled time of 0.1 has elapsed. The outer boundary is numerical error. (B) The initial pattern was three lines (as you can see) with concentrations 0.3-0.35 and 0.45-0.5, respectively. The scale is the same as before.

Appendix C: Screenshots of the Program

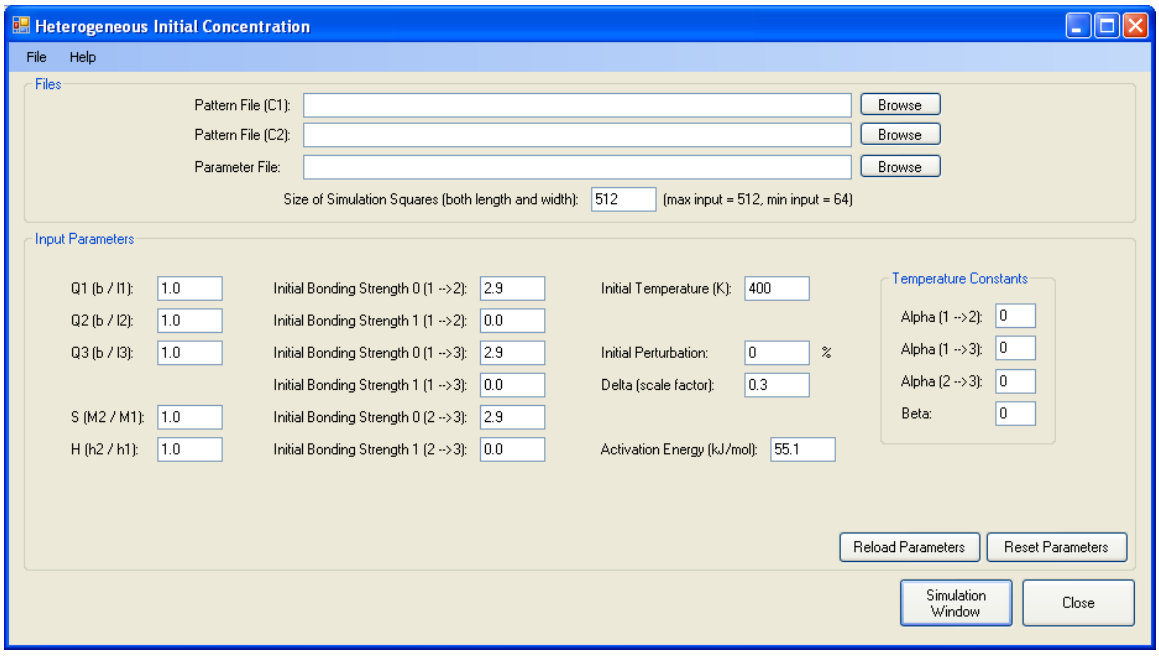
A



B



C



D

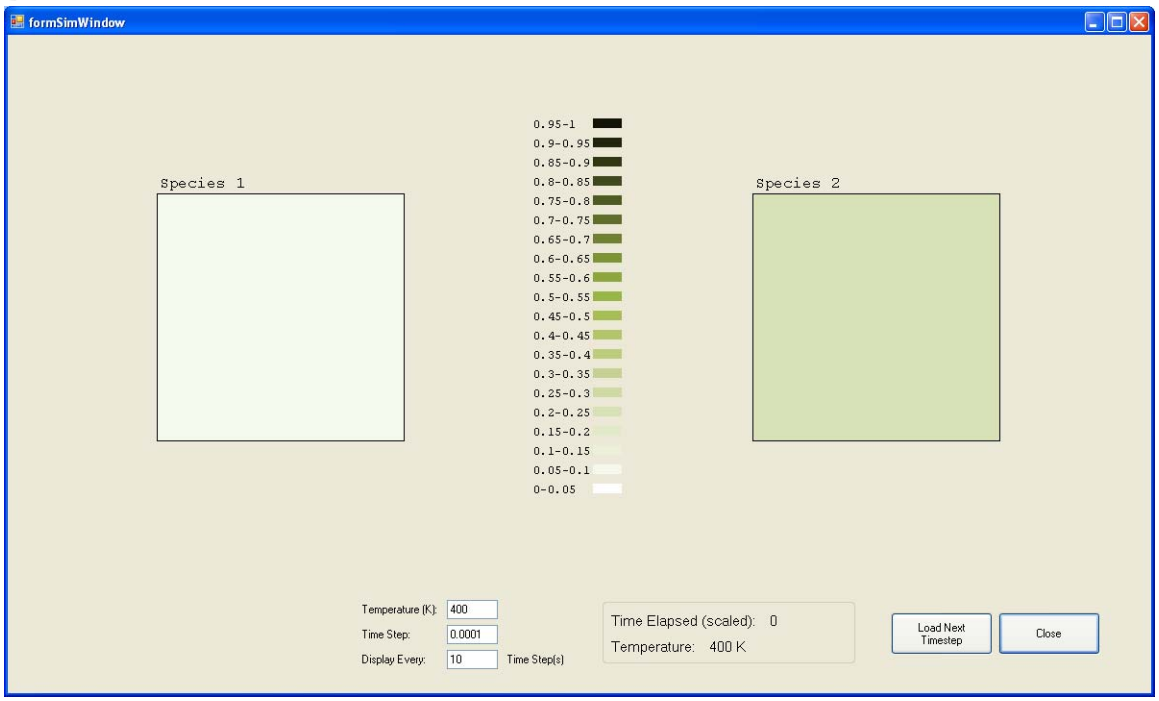


Figure 11: Program screenshots. (A) Start screen. (B) Homogeneous parameter input screen. (C) Heterogeneous input screen. (D) Simulation screen.

Appendix D: Source Code

formStart.cs

```
//=============================================================================
//
// File Name    :  formStart.cs
// Purpose      :  To give a broad overview of the program and provides options
//              :  for creating a homogeneous or heterogeneous simulation
//
// Author       :  Micheal Wang
//              Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
//
//=============================================================================

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

namespace NanoSimulator
{
   public partial class form_start : Form
   {
      /**
       * constructor
       */
      public form_start()
      {
         InitializeComponent();
      }

      //-----------------------------------------------------------------
      /**
       *
       */
      private void form_start_Load(object sender, EventArgs e)
      {

      }

      //-----------------------------------------------------------------
      /**
       * Calls either the heterogeneous or homogeneous input forms depending
       * on which of the choices is selected
       */
      private void bt_start_Click(object sender, EventArgs e)
      {
         if (rbt_homo.Checked.Equals(true))
         {
            formHomogeneous fhomo = new formHomogeneous();
            fhomo.Show();
         }
         else
         {
            formHeterogeneous fhetero = new formHeterogeneous();
            fhetero.Show();
```

```
            }
        }

        //---------------------------------------------------------------------
        /**
         * Exits program
         */
        private void bt_quit_Click(object sender, EventArgs e)
        {
            this.Dispose();
        }

        //---------------------------------------------------------------------
        /**
         * This sets the homogeneous option to true and the heterogeneous option
         * to false
         */
        private void rbt_homo_CheckedChanged(object sender, EventArgs e)
        {
            rbt_homo.Checked.Equals(true);
            rbt_hetero.Checked.Equals(false);
        }

        //---------------------------------------------------------------------
        /**
         * This sets the heterogeneous option to true and the homogeneous option
         * to false
         */
        private void rbt_hetero_CheckedChanged(object sender, EventArgs e)
        {
            rbt_homo.Checked.Equals(false);
            rbt_hetero.Checked.Equals(true);
        }
        //---------------------------------------------------------------------
    }
}
```

formHomogeneous.cs

```
//=============================================================================
//
// File Name    :  formHomogeneous.cs
// Purpose      :  To create an interface for changing parameter values
//
// Author       :  Micheal Wang
//              Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
//
//=============================================================================

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

namespace NanoSimulator
{
    public partial class formHomogeneous : Form
```

```
{
    Parameters pHomo;

    /**
     * constructor
     */
    public formHomogeneous()
    {
        pHomo = new Parameters();
        InitializeComponent();
    }

    //---------------------------------------------------------------------
    /**
     *
     */
    private void formHomogeneous_Load(object sender, EventArgs e)
    {

    }

    //---------------------------------------------------------------------
    /**
     * This exits the homogeneous input screen
     */
    private void closeToolStripMenuItem_Click(object sender, EventArgs e)
    {
        this.Dispose();
    }

    //---------------------------------------------------------------------
    /**
     * This exits the homogeneous input screen
     */
    private void bt_close_Click(object sender, EventArgs e)
    {
        this.Dispose();
    }

    //---------------------------------------------------------------------
    /**
     * This resets the parameters to their default values
     */
    private void bt_reset_Click(object sender, EventArgs e)
    {
        tb_C1.Text = "0";
        tb_C2.Text = "0";
        tb_Q1.Text = "1.0";
        tb_Q2.Text = "1.0";
        tb_Q3.Text = "1.0";
        tb_S.Text = "1.0";
        tb_H.Text = "1.0";
        tb_bond012.Text = "2.9";
        tb_bond112.Text = "0.0";
        tb_bond013.Text = "2.9";
        tb_bond113.Text = "0.0";
        tb_bond023.Text = "2.9";
        tb_bond123.Text = "0.0";
        tb_initTemp.Text = "400";
        tb_noise.Text = "0";
        tb_delta.Text = "0.3";
        tb_deltaE.Text = "55.1";
```

```
}

//---------------------------------------------------------------------
/**
 * This stores all of the parameter values and calls/opens the
 * simulation window
 */
private void bt_startSim_Click(object sender, EventArgs e)
{
   pHomo.homoC1 = Double.Parse(tb_C1.Text);
   pHomo.homoC2 = Double.Parse(tb_C2.Text);

   pHomo.iTemperature = Double.Parse(tb_initTemp.Text);
   pHomo.sqLength = Int16.Parse(tb_simSqLength.Text);
   pHomo.delta = Double.Parse(tb_delta.Text);

   pHomo.cMatSize = 2 * pHomo.sqLength * pHomo.sqLength;
   pHomo.kSize = pHomo.sqLength * pHomo.sqLength;

   pHomo.Q1 = Double.Parse(tb_Q1.Text);
   pHomo.Q2 = Double.Parse(tb_Q2.Text);
   pHomo.Q3 = Double.Parse(tb_Q3.Text);

   pHomo.S = Double.Parse(tb_S.Text);
   pHomo.H = Double.Parse(tb_H.Text);

   pHomo.o_0_12 = Double.Parse(tb_bond012.Text);
   pHomo.o_1_12 = Double.Parse(tb_bond112.Text);
   pHomo.o_0_13 = Double.Parse(tb_bond013.Text);
   pHomo.o_1_13 = Double.Parse(tb_bond113.Text);
   pHomo.o_0_23 = Double.Parse(tb_bond023.Text);
   pHomo.o_1_23 = Double.Parse(tb_bond123.Text);
   pHomo.a_12 = Double.Parse(tb_alpha12.Text);
   pHomo.a_13 = Double.Parse(tb_alpha13.Text);
   pHomo.a_23 = Double.Parse(tb_alpha23.Text);
   pHomo.beta = Double.Parse(tb_beta.Text);

   pHomo.activationEnergy = Double.Parse(tb_deltaE.Text);

   pHomo.percentNoise = Double.Parse(tb_noise.Text) / 100;

   formSimWindow fsm = new formSimWindow(pHomo);
    fsm.homogeneousArray();
    fsm.Show();
}

//---------------------------------------------------------------------
/**
 * This opens the browse dialog box where the user can choose a
 * parameter file
 */
private void bt_parameterBrowse_Click(object sender, EventArgs e)
{
   openParameterFile();
}

//---------------------------------------------------------------------
/**
 * This opens the browse dialog box where the user can choose a
 * parameter file
 */
private void openToolStripMenuItem_Click(object sender, EventArgs e)
```

```
{
    openParameterFile();
}

//---------------------------------------------------------------------
/**
 * This reloads the parameters from the selected file
 */
private void bt_reload_Click(object sender, EventArgs e)
{
    readInParam();
}

//---------------------------------------------------------------------
/**
 * This opens the save dialog box where the user can save the parameters
 */
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
    SaveFileDialog sfd = new SaveFileDialog();
    sfd.Filter = "Text File|*.txt";
    sfd.Title = "Save Image as Text File";
    sfd.FileName = "";

    if (sfd.ShowDialog() == DialogResult.OK)
    {
        System.IO.StreamWriter writer = new System.IO.StreamWriter(sfd.FileName);

        writer.WriteLine(tb_C1.Text);
        writer.WriteLine(tb_C2.Text);
        writer.WriteLine(tb_Q1.Text);
        writer.WriteLine(tb_Q2.Text);
        writer.WriteLine(tb_Q3.Text);
        writer.WriteLine(tb_S.Text);
        writer.WriteLine(tb_H.Text);
        writer.WriteLine(tb_bond012.Text);
        writer.WriteLine(tb_bond112.Text);
        writer.WriteLine(tb_bond013.Text);
        writer.WriteLine(tb_bond113.Text);
        writer.WriteLine(tb_bond023.Text);
        writer.WriteLine(tb_bond123.Text);
        writer.WriteLine(tb_initTemp.Text);
        writer.WriteLine(tb_noise.Text);
        writer.WriteLine(tb_delta.Text);
        writer.WriteLine(tb_alpha12.Text);
        writer.WriteLine(tb_alpha13.Text);
        writer.WriteLine(tb_alpha23.Text);
        writer.WriteLine(tb_beta.Text);
        writer.WriteLine(tb_deltaE.Text);

        writer.Close();
    }
}

//---------------------------------------------------------------------
/**
 * This reduces code since the same code for opening a file is used
 * more than once
 */
private void openParameterFile()
{
    OpenFileDialog ofd = new OpenFileDialog();
```

```
            ofd.Filter = "Text File|*.txt";
            ofd.Title = "Open Text File";
            ofd.FileName = "";

            if (ofd.ShowDialog() == DialogResult.OK)
            {
                tb_parameterFile.Text = ofd.FileName;

                readInParam();
            }
        }

        //---------------------------------------------------------------
        /**
         * This reads in the parameters from the selected file
         */
        private void readInParam()
        {
            if (tb_parameterFile.Text.Equals(""))
            {
            }
            else
            {
                StreamReader paramRd = new StreamReader(tb_parameterFile.Text);

                tb_C1.Text = paramRd.ReadLine();
                tb_C2.Text = paramRd.ReadLine();
                tb_Q1.Text = paramRd.ReadLine();
                tb_Q2.Text = paramRd.ReadLine();
                tb_Q3.Text = paramRd.ReadLine();
                tb_S.Text = paramRd.ReadLine();
                tb_H.Text = paramRd.ReadLine();
                tb_bond012.Text = paramRd.ReadLine();
                tb_bond112.Text = paramRd.ReadLine();
                tb_bond013.Text = paramRd.ReadLine();
                tb_bond113.Text = paramRd.ReadLine();
                tb_bond023.Text = paramRd.ReadLine();
                tb_bond123.Text = paramRd.ReadLine();
                tb_initTemp.Text = paramRd.ReadLine();
                tb_noise.Text = paramRd.ReadLine();
                tb_delta.Text = paramRd.ReadLine();
                tb_alpha12.Text = paramRd.ReadLine();
                tb_alpha13.Text = paramRd.ReadLine();
                tb_alpha23.Text = paramRd.ReadLine();
                tb_beta.Text = paramRd.ReadLine();
                tb_deltaE.Text = paramRd.ReadLine();

                paramRd.Close();
            }
        }
        //---------------------------------------------------------------
    }
}
```

formHeterogeneous.cs

```
//=============================================================================
//
// File Name    :  formHeterogeneous.cs
// Purpose      :  To create an interface for changing parameter values and
//              :  selcting a prescibed pattern
//
// Author       :  Micheal Wang
```

```
//                Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
//
//=============================================================================

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

namespace NanoSimulator
{
   public partial class formHeterogeneous : Form
   {
      Parameters pHetero;
      OpenFileDialog ofd = new OpenFileDialog();

      /**
       * constructor
       */
      public formHeterogeneous()
      {
         pHetero = new Parameters();
         InitializeComponent();
      }

      //-----------------------------------------------------------------
      /**
       *
       */
      private void formHeterogeneous_Load(object sender, EventArgs e)
      {

      }

      //-----------------------------------------------------------------
      /**
       * This closes the heterogeneous input screen
       */
      private void closeToolStripMenuItem_Click(object sender, EventArgs e)
      {
         this.Dispose();
      }

      //-----------------------------------------------------------------
      /**
       * This also closes the heterogeneous input screen
       */
      private void bt_close_Click(object sender, EventArgs e)
      {
         this.Dispose();
      }

      //-----------------------------------------------------------------
      /**
       * This resets the parameters to their default values
       */
```

```
        private void bt_reset_Click(object sender, EventArgs e)
        {
            tb_Q1.Text = "1.0";
            tb_Q2.Text = "1.0";
            tb_Q3.Text = "1.0";
            tb_S.Text = "1.0";
            tb_H.Text = "1.0";
            tb_bond012.Text = "2.9";
            tb_bond112.Text = "0.0";
            tb_bond013.Text = "2.9";
            tb_bond113.Text = "0.0";
            tb_bond023.Text = "2.9";
            tb_bond123.Text = "0.0";
            tb_initTemp.Text = "400";
            tb_noise.Text = "0";
            tb_delta.Text = "0.3";
            tb_deltaE.Text = "55.1";
        }

        //---------------------------------------------------------------------
        /**
         * This stores the parameter inputs and checks to make sure that:
         * (1) there are two pattern files
         * (2) the size of the simulation squares are a power of two
         * (3) the size of the pattern file correspondes to the size input
         */
        private void bt_startSim_Click(object sender, EventArgs e)
        {
            pHetero.Q1 = Double.Parse(tb_Q1.Text);
            pHetero.Q2 = Double.Parse(tb_Q2.Text);
            pHetero.Q3 = Double.Parse(tb_Q3.Text);

            pHetero.iTemperature = Double.Parse(tb_initTemp.Text);
            pHetero.sqLength = Int16.Parse(tb_simSqLength.Text);
            pHetero.delta = Double.Parse(tb_delta.Text);

            pHetero.cMatSize = 2 * pHetero.sqLength * pHetero.sqLength;
            pHetero.kSize = pHetero.sqLength * pHetero.sqLength;

            pHetero.S = Double.Parse(tb_S.Text);
            pHetero.H = Double.Parse(tb_H.Text);

            pHetero.o_0_12 = Double.Parse(tb_bond012.Text);
            pHetero.o_1_12 = Double.Parse(tb_bond112.Text);
            pHetero.o_0_13 = Double.Parse(tb_bond013.Text);
            pHetero.o_1_13 = Double.Parse(tb_bond113.Text);
            pHetero.o_0_23 = Double.Parse(tb_bond023.Text);
            pHetero.o_1_23 = Double.Parse(tb_bond123.Text);
            pHetero.a_12 = Double.Parse(tb_alpha12.Text);
            pHetero.a_13 = Double.Parse(tb_alpha13.Text);
            pHetero.a_23 = Double.Parse(tb_alpha23.Text);
            pHetero.beta = Double.Parse(tb_beta.Text);

            pHetero.activationEnergy = Double.Parse(tb_deltaE.Text);

            pHetero.percentNoise = Double.Parse(tb_noise.Text) / 100;

            //there must be a file in the each field to read
            if (tb_patternFileC1.Text.Equals("") || tb_patternFileC2.Text.Equals(""))
            {
                MessageBox.Show("One or both pattern file fields are empty", "Warning", MessageBoxButtons.OK,
MessageBoxIcon.Warning);
```

```
        }
        //the FFT algorithm can only transform data with dimension lenghths that are powers of 2
        else if (pHetero.sqLength / 64 != 1 && pHetero.sqLength / 128 != 1 && pHetero.sqLength / 256 != 1 &&
pHetero.sqLength / 512 != 1)
        {
            MessageBox.Show("Size of simulation square must be a power of 2 and in the given range", "Warning",
MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }
        //the image size has to match the simulation square size; otherwise, BitmapConverter will read past the end of
the pattern file
        else if (getImgSize(tb_patternFileC1.Text) < Math.Pow(pHetero.sqLength, 2) ||
getImgSize(tb_patternFileC2.Text) < Math.Pow(pHetero.sqLength, 2))
        {
            MessageBox.Show("The size of one or both of the image files is smaller than the input for the simulation
square size", "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning);
        }
        else
        {
            BitmapConverter bmpC = new BitmapConverter(pHetero);
            bmpC.convert(tb_patternFileC1.Text, tb_patternFileC2.Text);
        }
    }

    //---------------------------------------------------------------------
    /**
     * This returns the size of the image file
     */
    private int getImgSize(String imgFilename)
    {
        int picSize = 0;
        StreamReader rd = new StreamReader(imgFilename);
        for (; ; )
        {
            rd.Read();
            picSize++;

            if (rd.EndOfStream == true)
                break;
        }

        rd.Close();
        return (picSize - 54);
    }

    //---------------------------------------------------------------------
    /**
     * This opens the browse dialog box where the user can choose a
     * pattern file
     */
    private void bt_patternBrowse_Click_1(object sender, EventArgs e)
    {
        openPatternFile("C1");
    }

    //---------------------------------------------------------------------
    /**
     * This opens the browse dialog box where the user can choose a
     * pattern file
     */
    private void patternFileToolStripMenuItem_Click(object sender, EventArgs e)
    {
        openPatternFile("C1");
```

```
}

//---------------------------------------------------------------------
/**
 * This opens the browse dialog box where the user can choose a
 * pattern file
 */
private void bt_patternBrowseC2_Click(object sender, EventArgs e)
{
   openPatternFile("C2");
}

//---------------------------------------------------------------------
/**
 * This opens the browse dialog box where the user can choose a
 * pattern file
 */
private void patternFileC2ToolStripMenuItem_Click(object sender, EventArgs e)
{
   openPatternFile("C2");
}

//---------------------------------------------------------------------
/**
 * This opens the browse dialog box where the user can choose a
 * parameter file
 */
private void bt_parameterBrowse_Click(object sender, EventArgs e)
{
   openParameterFile();
}

//---------------------------------------------------------------------
/**
 * This opens the browse dialog box where the user can choose a
 * parameter file
 */
private void parameterFileToolStripMenuItem_Click(object sender, EventArgs e)
{
   openParameterFile();
}

//---------------------------------------------------------------------
/**
 * This reloads the parameters from the selected file
 */
private void bt_reload_Click(object sender, EventArgs e)
{
   readInParam();
}

//---------------------------------------------------------------------
/**
 * Method for saving parameters
 */
private void saveToolStripMenuItem_Click(object sender, EventArgs e)
{
   SaveFileDialog sfd = new SaveFileDialog();
   sfd.Filter = "Text File|*.txt";
   sfd.Title = "Save Paramters";
   sfd.FileName = "";
```

```
    if (sfd.ShowDialog() == DialogResult.OK)
    {
        System.IO.StreamWriter writer = new System.IO.StreamWriter(sfd.FileName);

        writer.WriteLine(tb_Q1.Text);
        writer.WriteLine(tb_Q2.Text);
        writer.WriteLine(tb_Q3.Text);
        writer.WriteLine(tb_S.Text);
        writer.WriteLine(tb_H.Text);
        writer.WriteLine(tb_bond012.Text);
        writer.WriteLine(tb_bond112.Text);
        writer.WriteLine(tb_bond013.Text);
        writer.WriteLine(tb_bond113.Text);
        writer.WriteLine(tb_bond023.Text);
        writer.WriteLine(tb_bond123.Text);
        writer.WriteLine(tb_initTemp.Text);
        writer.WriteLine(tb_noise.Text);
        writer.WriteLine(tb_delta.Text);
        writer.WriteLine(tb_alpha12.Text);
        writer.WriteLine(tb_alpha13.Text);
        writer.WriteLine(tb_alpha23.Text);
        writer.WriteLine(tb_beta.Text);
        writer.WriteLine(tb_deltaE.Text);

        writer.Close();
    }
}

//---------------------------------------------------------------------
/**
 * This is the method for opening a pattern file for C1 or C2
 */
private void openPatternFile(String C1orC2)
{
    ofd.Filter = "Bitmap Image|*.bmp";
    ofd.Title = "Open an Image File";
    ofd.FileName = "";
    ofd.ShowDialog();

    if (C1orC2.Equals("C1"))
    {
        tb_patternFileC1.Text = ofd.FileName;
    }
    else
    {
        tb_patternFileC2.Text = ofd.FileName;
    }
}

//---------------------------------------------------------------------
/**
 * This is the method for opening a parameter file
 */
private void openParameterFile()
{
    ofd.Filter = "Text File|*.txt";
    ofd.Title = "Open Parameter File";
    ofd.FileName = "";
    ofd.ShowDialog();

    tb_parameterFile.Text = ofd.FileName;
```

```
            readInParam();
        }

        //---------------------------------------------------------------
        /**
         * This reads in the parameters from the selected file
         */
        private void readInParam()
        {
            if (tb_parameterFile.Text.Equals(""))
            {
            }
            else
            {
                StreamReader paramRd = new StreamReader(tb_parameterFile.Text);

                tb_Q1.Text = paramRd.ReadLine();
                tb_Q2.Text = paramRd.ReadLine();
                tb_Q3.Text = paramRd.ReadLine();
                tb_S.Text = paramRd.ReadLine();
                tb_H.Text = paramRd.ReadLine();
                tb_bond012.Text = paramRd.ReadLine();
                tb_bond112.Text = paramRd.ReadLine();
                tb_bond013.Text = paramRd.ReadLine();
                tb_bond113.Text = paramRd.ReadLine();
                tb_bond023.Text = paramRd.ReadLine();
                tb_bond123.Text = paramRd.ReadLine();
                tb_initTemp.Text = paramRd.ReadLine();
                tb_noise.Text = paramRd.ReadLine();
                tb_delta.Text = paramRd.ReadLine();
                tb_alpha12.Text = paramRd.ReadLine();
                tb_alpha13.Text = paramRd.ReadLine();
                tb_alpha23.Text = paramRd.ReadLine();
                tb_beta.Text = paramRd.ReadLine();
                tb_deltaE.Text = paramRd.ReadLine();

                paramRd.Close();
            }
        }
        //---------------------------------------------------------------
    }
}
```

BitmapConverter.cs

```
//=============================================================================
//
// File Name    :  BitmapConverter.cs
// Purpose      :  To interpret the colors from a pattern file as concentration
//              :  values
//
// Author       :  Micheal Wang
//              Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
//
//=============================================================================

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

```
namespace NanoSimulator
{
   class BitmapConverter
   {
      Parameters pBmpConv;

      /**
       * constructor
       */
      public BitmapConverter(Parameters p)
      {
         pBmpConv = p;
      }

      //--------------------------------------------------------------------
      /**
       * This calls the converting method and then calls the simulation
       * window
       */
      public void convert(String filenameC1, String filenameC2)
      {
         pBmpConv.C1Mat = colorToConcentration(filenameC1);
         pBmpConv.C2Mat = colorToConcentration(filenameC2);

         formSimWindow fsm = new formSimWindow(pBmpConv);
          fsm.heterogeneousArray();
          fsm.Show();
      }

      //--------------------------------------------------------------------
      /**
       * This returns the concentrations corresponding to the colors in
       * the bmp file
       */
      private double[] colorToConcentration(String filename)
      {
         double[] Mat = new double[pBmpConv.cMatSize];
         byte hex1, hex2, hex3;
         int pixelColor;

         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 = 0; i < pBmpConv.cMatSize; i += 2)
         {
            hex1 = reader.ReadByte();
            hex2 = reader.ReadByte();
            hex3 = reader.ReadByte();
            pixelColor = (hex1 << 16) + (hex2 << 8) + hex3;
            switch (pixelColor)
            {
               case 0x000000://black
                  Mat[i] = 0.99;
                  break;
               case 0x061310:
                  Mat[i] = 0.95;
```

```
            break;
        case 0x0E2520:
            Mat[i] = 0.9;
            break;
        case 0x143830:
            Mat[i] = 0.85;
            break;
        case 0x1D4940:
            Mat[i] = 0.8;
            break;
        case 0x245B4F:
            Mat[i] = 0.75;
            break;
        case 0x2B6F60:
            Mat[i] = 0.7;
            break;
        case 0x32816F:
            Mat[i] = 0.65;
            break;
        case 0x37957D:
            Mat[i] = 0.6;
            break;
        case 0x3EA88D:
            Mat[i] = 0.55;
            break;
        case 0x48B798:
            Mat[i] = 0.5;
            break;
        case 0x59BFA8:
            Mat[i] = 0.45;
            break;
        case 0x6CC6B4:
            Mat[i] = 0.4;
            break;
        case 0x7ECDBD:
            Mat[i] = 0.35;
            break;
        case 0x93D2C5:
            Mat[i] = 0.3;
            break;
        case 0xA4DBCE:
            Mat[i] = 0.25;
            break;
        case 0xB6E2D9:
            Mat[i] = 0.2;
            break;
        case 0xC8EAE3:
            Mat[i] = 0.15;
            break;
        case 0xDAF1ED:
            Mat[i] = 0.1;
            break;
        case 0xEDF8F5:
            Mat[i] = 0.05;
            break;
        default:
            Mat[i] = 0.05;
            break;
    }
    Mat[i + 1] = 0;
}
```

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

            return Mat;
        }
    }
}
```

formSimWindow.cs

```
//=============================================================================
//
// File Name    :  fromSimWindow.cs
// Purpose      :  To display the new concentration as well as time elapsed, etc.
//
// Author       :  Micheal Wang
//              Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
//
//=============================================================================

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

namespace NanoSimulator
{
    public partial class formSimWindow : Form
    {
        Parameters pSimWin;
        FFTandCalc calc;
        Random r = new Random();
        bool warningAlreadyDisp = false;
        bool loadNextIsClicked = false;
        double tempVar;
        double timeStep;
        double timeElapsed;
        int numTimeStepsPerDisp;

        /**
         * constructor
         */
        public formSimWindow(Parameters p)
        {
            pSimWin = p;
            InitializeComponent();
        }

        //--------------------------------------------------------------------
        /**
         *
         */
        private void formSimWindow_Load(object sender, EventArgs e)
        {

        }

        //--------------------------------------------------------------------
```

```
        /**
         * A random noise is added to each concentration based on the "percent
         * noise" input.  If the sum of the concentrations is greater or
         * equal to 1, they are scaled by their sum.
         */
        public void heterogeneousArray()
        {
            for (int i = 0; i < pSimWin.cMatSize; i += 2)
            {
                pSimWin.C1Mat[i] = pSimWin.C1Mat[i] + pSimWin.percentNoise * (r.NextDouble() - 0.5);
                pSimWin.C2Mat[i] = pSimWin.C2Mat[i] + pSimWin.percentNoise * (r.NextDouble() - 0.5);

                tempVar = pSimWin.C1Mat[i] + pSimWin.C2Mat[i];
                if (tempVar > 1)
                {
                    if (warningAlreadyDisp == false)
                    {
                        MessageBox.Show("Note: C1 + C2 must be < 1.  Otherwise, C2[i, j] will scaled i.e. will be divided by
(C1Mat[i] + C2Mat[i])",
                            "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                        warningAlreadyDisp = true;
                    }
                    pSimWin.C1Mat[i] = pSimWin.C1Mat[i] / tempVar;
                    pSimWin.C2Mat[i] = pSimWin.C2Mat[i] / tempVar;
                }
            }

            tb_timeStep.Text = "0.001";
            tb_temp.Text = pSimWin.iTemperature + "";
            calc_k();
        }

        //---------------------------------------------------------------------
        /**
         * The homogeneous arrays are assigned the input values.  The addition
         * of noise and scaling are similar to those in heterogeneousArray().
         */
        public void homogeneousArray()
        {
            pSimWin.C1Mat = new double[pSimWin.cMatSize];
            pSimWin.C2Mat = new double[pSimWin.cMatSize];

            for (int i = 0; i < pSimWin.cMatSize; i += 2)
            {
                pSimWin.C1Mat[i] = pSimWin.homoC1 + pSimWin.percentNoise * (r.NextDouble() - 0.5);
                pSimWin.C2Mat[i] = pSimWin.homoC2 + pSimWin.percentNoise * (r.NextDouble() - 0.5);

                pSimWin.C1Mat[i + 1] = 0.0;
                pSimWin.C2Mat[i + 1] = 0.0;

                tempVar = pSimWin.C1Mat[i] + pSimWin.C2Mat[i];
                if (tempVar > 1)
                {
                    if (warningAlreadyDisp == false)
                    {
                        MessageBox.Show("Note: C1 + C2 must be < 1.  Otherwise, C2[i, j] will scaled i.e. will be divided by
(C1Mat[i] + C2Mat[i])",
                            "Warning", MessageBoxButtons.OK, MessageBoxIcon.Warning);
                        warningAlreadyDisp = true;
                    }

                    pSimWin.C1Mat[i] = pSimWin.C1Mat[i] / tempVar;
```

```
            pSimWin.C2Mat[i] = pSimWin.C2Mat[i] / tempVar;
        }
    }

    tb_timeStep.Text = "0.0001";
    tb_temp.Text = pSimWin.iTemperature + "";
    calc_k();
}

//---------------------------------------------------------------------
/**
 * Since the frequencies in Fourier space remain unchanged throughout
 * the entire simulation, they are calculated ahead of time (see
 * Final Report for details on calculating k).
 */
private void calc_k()
{
    double freq1 = 0;
    double freq2 = 0;
    double fourCellSize = 2 * Math.PI / (pSimWin.sqLength * pSimWin.delta);
    int j = 0;

    pSimWin.k = new double[pSimWin.kSize];

    for (int i = 0; i < pSimWin.kSize; i++)
    {
        pSimWin.k[i] = Math.Sqrt(freq1 * freq1 + freq2 * freq2);

        if ((i + 1) % pSimWin.sqLength <= pSimWin.sqLength / 2)
            freq1 += fourCellSize;
        else
            freq1 -= fourCellSize;

        if ((i + 1) % pSimWin.sqLength == 0)
        {
            if ((j + 1) <= pSimWin.sqLength / 2)
                freq2 += fourCellSize;
            else
                freq2 -= fourCellSize;

            freq1 = 0;
            j++;
        }
    }
}

//---------------------------------------------------------------------
/**
 * This closes simulation window
 */
private void bt_close_Click(object sender, EventArgs e)
{
    this.Dispose();
}

//---------------------------------------------------------------------
/**
 * This calls the calculations class when the button is clicked and
 * the displaying method when the calculations a finished
 */
private void bt_next_Click(object sender, EventArgs e)
{
```

```
    timeStep = Double.Parse(tb_timeStep.Text);
    numTimeStepsPerDisp = Int16.Parse(tb_dispEvery.Text);
    timeElapsed += numTimeStepsPerDisp * timeStep;

    //This prevents calling FFTandCalc again and erasing some of the calculated parameters
    if (loadNextIsClicked == false)
    {
        calc = new FFTandCalc(pSimWin);
        loadNextIsClicked = true;
    }

    bt_next.Hide();
    bt_close.Hide();
    tb_dispEvery.Hide();
    tb_timeStep.Hide();
    tb_temp.Hide();

    calc.calc_TempConstants(Double.Parse(tb_temp.Text));
    calc.calcNext(numTimeStepsPerDisp, timeStep);  //contains a loop

    bt_next.Show();
    bt_close.Show();
    tb_dispEvery.Show();
    tb_timeStep.Show();
    tb_temp.Show();

    Graphics g = p_graphics.CreateGraphics();
    Rectangle r = new Rectangle(0, 0, 1165, 559);
    PaintEventArgs e1 = new PaintEventArgs(g, r);
    p_graphics_Paint(sender, e1);
}

//---------------------------------------------------------------------
/**
 * This displays the calcultated concentrations.  In addition, the
 * outline rectangles, component labels, color scheme, time, and
 * tmperature are displayed.
 */
private void p_graphics_Paint(object sender, PaintEventArgs e)
{
    int xyShift = (512 - pSimWin.sqLength) / 2;  //recenters simulation squares when their size is changed
    int locationX, locationY;
    int xy_to_i;
    SolidBrush colorBrush;
    SolidBrush stringBrush = new SolidBrush(Color.Black);
    Pen pen = new Pen(Color.Black, 1);
    Font font;

    //outline rectangles
    int simRectSize = pSimWin.sqLength + 1;
    locationX = 15 + xyShift;
    locationY = 25 + xyShift;

    e.Graphics.DrawRectangle(pen, locationX, locationY, simRectSize, simRectSize);
    e.Graphics.DrawRectangle(pen, locationX + 620, locationY, simRectSize, simRectSize);

    //color scheme
    font = new Font(FontFamily.GenericMonospace, 9);
    for (int i = 0; i < 100; i += 5)
    {
        colorBrush = new SolidBrush(getColor(i / 100.0));
        e.Graphics.FillRectangle(colorBrush, 597, 447 - 4 * (i - 2), 30, 10);
```

```
            e.Graphics.DrawString(i / 100.0 + "-" + (i + 5) / 100.0, font, stringBrush, 533, 445 - 4 * (i - 2));
        }

        //simulation square labels
        font = new Font(FontFamily.GenericMonospace, 12);
        e.Graphics.DrawString("Species 1", font, stringBrush, locationX, locationY - 20);
        e.Graphics.DrawString("Species 2", font, stringBrush, locationX + 620, locationY - 20);

        //total scaled time elapsed and current temperature
        time.Text = timeElapsed + "";
        temperature.Text = tb_temp.Text + " K";

        //display new concentrations
        for (int y = 0; y < pSimWin.sqLength; y++)
        {
            for (int x = 0; x < pSimWin.sqLength; x++)
            {
                xy_to_i = 2 * (pSimWin.sqLength * y + x);
                locationX = xyShift + x + 16;
                locationY = 25 + xyShift + pSimWin.sqLength - y;

                colorBrush = new SolidBrush(getColor(pSimWin.C1Mat[xy_to_i]));
                e.Graphics.FillRectangle(colorBrush, locationX, locationY, 1, 1);

                colorBrush = new SolidBrush(getColor(pSimWin.C2Mat[xy_to_i]));
                e.Graphics.FillRectangle(colorBrush, locationX + 620, locationY, 1, 1);
            }
        }
    }

    //-----------------------------------------------------------------
    /**
     * Returns a color corresponding to a range of concentrations.
     */
    private Color getColor(double CValue)
    {
        if (CValue < 1 && CValue > 0.95)
            return Color.Black;
        else if (CValue <= 0.95 && CValue > 0.9)
            return Color.FromArgb(16, 19, 6);
        else if (CValue <= 0.9 && CValue > 0.85)
            return Color.FromArgb(32, 37, 14);
        else if (CValue <= 0.85 && CValue > 0.8)
            return Color.FromArgb(48, 56, 20);
        else if (CValue <= 0.8 && CValue > 0.75)
            return Color.FromArgb(64, 73, 29);
        else if (CValue <= 0.75 && CValue > 0.7)
            return Color.FromArgb(79, 91, 36);
        else if (CValue <= 0.7 && CValue > 0.65)
            return Color.FromArgb(96, 111, 43);
        else if (CValue <= 0.65 && CValue > 0.6)
            return Color.FromArgb(111, 129, 50);
        else if (CValue <= 0.6 && CValue > 0.55)
            return Color.FromArgb(125, 149, 55);
        else if (CValue <= 0.55 && CValue > 0.5)
            return Color.FromArgb(141, 168, 62);
        else if (CValue <= 0.5 && CValue > 0.45)
            return Color.FromArgb(152, 183, 72);
        else if (CValue <= 0.45 && CValue > 0.4)
            return Color.FromArgb(168, 191, 89);
        else if (CValue <= 0.4 && CValue > 0.35)
            return Color.FromArgb(180, 198, 108);
```

```
            else if (CValue <= 0.35 && CValue > 0.3)
                return Color.FromArgb(189, 205, 126);
            else if (CValue <= 0.3 && CValue > 0.25)
                return Color.FromArgb(197, 210, 147);
            else if (CValue <= 0.25 && CValue > 0.2)
                return Color.FromArgb(206, 219, 164);
            else if (CValue <= 0.2 && CValue > 0.15)
                return Color.FromArgb(217, 226, 182);
            else if (CValue <= 0.15 && CValue > 0.1)
                return Color.FromArgb(227, 234, 200);
            else if (CValue <= 0.1 && CValue > 0.05)
                return Color.FromArgb(237, 241, 218);
            else if (CValue <= 0.05 && CValue > 1E-6)
                return Color.FromArgb(245, 248, 237);
            else
                return Color.White;
        }
        //---------------------------------------------------------------------
    }
}
```

FFTandCalc

```
//=============================================================================
//
// File Name    :  FFTandCalc.cs
// Purpose      :  To perform all of the calculations
//
// Author       :  Micheal Wang
//                 Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
//
//=============================================================================

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

namespace NanoSimulator
{
    class FFTandCalc
    {
        Parameters pfc;

        /**
         * constructor
         */
        public FFTandCalc(Parameters p)
        {
            pfc = p;
        }

        double tempExpTerm, tempTermA12, tempTermA13, tempTermA23, tempTermBeta, tempRatio;

        //---------------------------------------------------------------------
        /**
         * This calculates the temperature constants used in calculations
         */
        public void calc_TempConstants(double newTemp1)
        {
            double R = 8.314472;    //ideal gas constant J/(mol K)
            double iTemp = pfc.iTemperature;
```

```
    tempExpTerm = Math.Pow(Math.E, 1000 * pfc.activationEnergy / R * (1 / iTemp - 1 / newTemp1));
    tempTermA12 = 1 + pfc.a_12 * (iTemp - newTemp1);
    tempTermA13 = 1 + pfc.a_13 * (iTemp - newTemp1);
    tempTermA23 = 1 + pfc.a_23 * (iTemp - newTemp1);
    tempTermBeta = 1 + pfc.beta * (iTemp - newTemp1);
    tempRatio = newTemp1 / iTemp;
}

//-----------------------------------------------------------------
/**
 * This calculates the next timestep.
 *
 * Because of the natural log in the P1 and P2 functions, C1 and C2
 * cannot equal zero and their sum cannot exceed one.  These
 * restrictions are checked for.
 *
 * The next timestep is calculated.
 */
public void calcNext(int numLoops, double timeStep)
{
    int[] ithDimLength = { pfc.sqLength, pfc.sqLength };
    double[] P1 = new double[pfc.cMatSize];
    double[] P2 = new double[pfc.cMatSize];
    double C1, C2, C11, C22, tempVar, tempVar1;

    for (int numTimeSteps = 0; numTimeSteps < numLoops; numTimeSteps++)
    {
        for (int i = 0; i < pfc.cMatSize; i += 2)
        {
            C11 = (C1 = pfc.C1Mat[i]);
            C22 = (C2 = pfc.C2Mat[i]);

            tempVar = 1 - C1 - C2;

            if (C1 <= 0)
            {
                C11 = 1E-20;
                C1 = 0;
                pfc.C1Mat[i] = 0;
            }

            if (C2 <= 0)
            {
                C22 = 1E-20;
                C2 = 0;
                pfc.C2Mat[i] = 0;
            }

            tempVar1 = C1 + C2;
            if (tempVar1 >= 1)
            {
                C1 = (pfc.C1Mat[i] = pfc.C1Mat[i] / tempVar1);
                C2 = (pfc.C2Mat[i] = pfc.C2Mat[i] / tempVar1);
                tempVar = 1E-20;
            }

            // calculate P1 and P2, "Simulation on nanoscale self-assembly of ternary-epilayers" page 25
            P1[i] = Math.Log(C11 / tempVar, Math.E) * tempRatio;
            P1[i] += C2 * (pfc.o_0_12 + pfc.o_1_12 * ((2 * C1) - C2)) * tempTermA12;
            P1[i] -= C2 * (pfc.o_0_23 + pfc.o_1_23 * ((2 * C1) + (3 * C2) - 2)) * tempTermA23;
            P1[i] += (pfc.o_0_13 * (1 - (2 * C1) - C2) + pfc.o_1_13 *
```

```
                ((6 * C1) + (2 * C2) - (6 * C1 * C1) - (C2 * C2) - (6 * C1 * C2) - 1)) * tempTermA13;

            P2[i] = Math.Log(C22 / tempVar, Math.E) * tempRatio;
            P2[i] += C1 * (pfc.o_0_12 + pfc.o_1_12 * (C1 - (2 * C2))) * tempTermA12;
            P2[i] -= C1 * (pfc.o_0_13 + pfc.o_1_13 * ((3 * C1) + (2 * C2) - 2)) * tempTermA13;
            P2[i] += (pfc.o_0_23 * (1 - C1 - (2 * C2)) + pfc.o_1_23 *
                ((2 * C1) + (6 * C2) - (C1 * C1) - (6 * C2 * C2) - (6 * C1 * C2) - 1)) * tempTermA23;

            P1[i + 1] = 0;
            P2[i + 1] = 0;
        }

        nDimFFT(P1, P2, ithDimLength, 1);//transform P1 and P2 from real to Fourier space
        nDimFFT(pfc.C1Mat, pfc.C2Mat, ithDimLength, 1);//transform C1 and C2 from real to Fourier space

        // update concentrations, "Simulation on nanoscale self-assembly of ternary-epilayers" page 25
        double a, b, c, d, determinant;
        double termC1P1r, termC1P1i, termC2P2r, termC2P2i, term1, term2;
        int ii;
        for (int i = 0; i < pfc.cMatSize; i += 2)
        {
            ii = i >> 1;    // i/2
            term1 = pfc.k[ii] * pfc.k[ii] * timeStep * tempExpTerm;
            term2 = 2 * term1 * pfc.k[ii];

            a = 1 + term2 * (pfc.k[ii] - pfc.Q1 / tempTermBeta);
            b = -pfc.Q2 / tempTermBeta * term2;
            c = pfc.S * b;
            d = 1 + term2 * pfc.S * (pfc.H * pfc.k[ii] - pfc.Q3 / tempTermBeta);
            determinant = 1 / (a * d - b * c);      //used to find inverse of the matrix

            termC1P1r = pfc.C1Mat[i] - term1 * P1[i];  //real part
            termC1P1i = pfc.C1Mat[i + 1] - term1 * P1[i + 1];  //imaginary part
            termC2P2r = pfc.C2Mat[i] - term1 * pfc.S * P2[i];  //real part
            termC2P2i = pfc.C2Mat[i + 1] - term1 * pfc.S * P2[i + 1];  //imaginary part

            pfc.C1Mat[i] = determinant * (d * termC1P1r - b * termC2P2r);
            pfc.C1Mat[i + 1] = determinant * (d * termC1P1i - b * termC2P2i);
            pfc.C2Mat[i] = determinant * (-c * termC1P1r + a * termC2P2r);
            pfc.C2Mat[i + 1] = determinant * (-c * termC1P1i + a * termC2P2i);
        }

        nDimFFT(pfc.C1Mat, pfc.C2Mat, ithDimLength, -1);  //transform C1 and C2 from Fourier to real space
        for (int i = 0; i < pfc.cMatSize; i += 2)
        {
            //The FFT algorithm returns the real data times the product of the lengths of each dimension
            pfc.C1Mat[i] = pfc.C1Mat[i] / pfc.kSize;
            pfc.C2Mat[i] = pfc.C2Mat[i] / pfc.kSize;

            pfc.C1Mat[i + 1] = 0;   //reduce numerical error.  Imaginary part supposed to be zero anyways.
            pfc.C2Mat[i + 1] = 0;   //reduce numerical error

            tempVar1 = pfc.C1Mat[i] + pfc.C2Mat[i];
            if (tempVar1 > 1)
            {
                pfc.C1Mat[i] = pfc.C1Mat[i] / tempVar1;
                pfc.C2Mat[i] = pfc.C2Mat[i] / tempVar1;
            }
        }
    }
}
```

```
//---------------------------------------------------------------------
/**
 * FFT algorithm.  Slightly altered to transform two sets of data at
 * once.
 *
 * Adopted from "Numerical Recipes in C++", pages 528-529.
 */
private void nDimFFT(double[] data1, double[] data2, int[] nn, int isign)
{
   int idim, i1, i2, i3, i2rev, i3rev, ip1, ip2, ip3, ifp1, ifp2;
   int ibit, k1, k2, n, nprev, nrem, ntot;
   double theta, tempi, tempr, wi, wpi, wr, wpr, wtemp, tempSwap;

   int ndim = nn.Length;
   ntot = data1.Length >> 1;
   nprev = 1;
   //bit reverse section
   for (idim = ndim - 1; idim >= 0; idim--)
   {
      n = nn[idim];
      nrem = ntot / (n * nprev);
      ip1 = nprev << 1;
      ip2 = ip1 * n;
      ip3 = ip2 * nrem;
      i2rev = 0;
      for (i2 = 0; i2 < ip2; i2 += ip1)
      {
         if (i2 < i2rev)
         {
            for (i1 = i2; i1 < i2 + ip1 - 1; i1 += 2)
            {
               for (i3 = i1; i3 < ip3; i3 += ip2)
               {
                  i3rev = i2rev + i3 - i2;

                  //swap data1[i3] and data1[i3rev]
                  tempSwap = data1[i3];
                  data1[i3] = data1[i3rev];
                  data1[i3rev] = tempSwap;

                  //swap data1[i3 + 1] and data1[i3rev + 1]
                  tempSwap = data1[i3 + 1];
                  data1[i3 + 1] = data1[i3rev + 1];
                  data1[i3rev + 1] = tempSwap;

                  //swap data2[i3] and data2[i3rev]
                  tempSwap = data2[i3];
                  data2[i3] = data2[i3rev];
                  data2[i3rev] = tempSwap;

                  //swap data2[i3 + 1] and data2[i3rev + 1]
                  tempSwap = data2[i3 + 1];
                  data2[i3 + 1] = data2[i3rev + 1];
                  data2[i3rev + 1] = tempSwap;
               }
            }
         }
         ibit = ip2 >> 1;
         while ((ibit >= ip1) && (i2rev + 1 > ibit))
         {
            i2rev -= ibit;
            ibit >>= 1;
```

```
                }
                i2rev += ibit;
            }
            ifp1 = ip1;
            //FFT calculations section
            while (ifp1 < ip2)
            {
                ifp2 = ifp1 << 1;
                theta = isign * 6.28318530717959 / (ifp2 / ip1);
                wtemp = Math.Sin(0.5 * theta);
                wpr = -2.0 * wtemp * wtemp;
                wpi = Math.Sin(theta);
                wr = 1.0;
                wi = 0.0;
                //Another name for the following method if the "Butterfly method"
                for (i3 = 0; i3 < ifp1; i3 += ip1)
                {
                    for (i1 = i3; i1 < i3 + ip1 - 1; i1 += 2)
                    {
                        for (i2 = i1; i2 < ip3; i2 += ifp2)
                        {
                            k1 = i2;
                            k2 = k1 + ifp1;

                            //Danielson-Lanczos Formula for data 1
                            tempr = wr * data1[k2] - wi * data1[k2 + 1];
                            tempi = wr * data1[k2 + 1] + wi * data1[k2];
                            data1[k2] = data1[k1] - tempr;
                            data1[k2 + 1] = data1[k1 + 1] - tempi;
                            data1[k1] += tempr;
                            data1[k1 + 1] += tempi;

                            //Danielson-Lanczos Formula for data 2
                            tempr = wr * data2[k2] - wi * data2[k2 + 1];
                            tempi = wr * data2[k2 + 1] + wi * data2[k2];
                            data2[k2] = data2[k1] - tempr;
                            data2[k2 + 1] = data2[k1 + 1] - tempi;
                            data2[k1] += tempr;
                            data2[k1 + 1] += tempi;
                        }
                    }
                    wr = (wtemp = wr) * wpr - wi * wpi + wr;
                    wi = wi * wpr + wtemp * wpi + wi;
                }
                ifp1 = ifp2;
            }
            nprev *= n;
        }
    }
  }
}
```

Parameters.cs

```
//=============================================================================
//
// File Name    :  Parameters.cs
// Purpose      :  To store all of the parameter that get tranferred around a lot
//
// Author       :  Micheal Wang
//              Albuquerque Academy
// Created on   :  December 2007
// Copyright    :  All Rights Reserved.
```

```
//
//=======================================================================

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

namespace NanoSimulator
{
   public struct Parameters
   {
      public double[] C1Mat;         //concentration 1 array
      public double[] C2Mat;         //concentration 2 array
      public double[] k;             //magnitude of frequency vector

      public double homoC1;          //homogeneous concentration 1 value
      public double homoC2;          //homogeneous concentration 2 value

      public int sqLength;           //length of a side of the simulation sq
      public double delta;           //nanometers per pixel
      public double iTemperature;    //initial temperature

      public int cMatSize;           //size of the concentrations arrays
      public int kSize;              //size magnitude of frequency vector

      public double Q1;              //constant
      public double Q2;              //constant
      public double Q3;              //constant

      public double S;               //ratio of moltilities (M2/M1)
      public double H;               //ratio of energies per species (h2/h1)

      //o is omega
      public double o_0_12;          //bonding strength 0 from species 1 to 2
      public double o_1_12;          //bonding strength 1 from species 1 to 2
      public double o_0_23;          //bonding strength 0 from species 2 to 3
      public double o_1_23;          //bonding strength 1 from species 2 to 3
      public double o_0_13;          //bonding strength 0 from species 1 to 3
      public double o_1_13;          //bonding strength 1 from species 1 to 3
      //a is alpha
      public double a_12;            //temperature constant
      public double a_13;            //temperature constant
      public double a_23;            //temperature constant
      public double beta;            //temperature constant

      public double activationEnergy; //Energy required for a reaction to start

      public double percentNoise;    //percent of disturbance on the surface
   }
}
```