Computation of Defects in Materials

Max Veit

July-August 2011

Abstract

The purpose of this project is to investigate and assess, using the MATLAB computer language, some numerical methods used in several fields of computational molecular dynamics. First a theoretical model of a one-dimensional chain of atoms was studied. The atoms in this chain would interact based on the Lennard-Jones potential energy function. Several algorithms were investigated that found configurations of the chain where the total potential energy was lowest. Aspects of the one-dimensional chain were then carried over into a model of a two-dimensional system of atoms. For this model a full simulation of the movement of the atoms in the system was used to study the system. It was found that one of the simplest atom configurations, a square lattice pattern, was unstable. In the simulation, this structure evolved over time into several disconnected regions, called "grains," of a more stable triangular-hexagonal lattice pattern. These structures are similar to crystal grains in real-world polycrystalline materials. Some basic computational thermodynamics (more specifically, Langevin dynamics) was also used in the simulation. It was found that by regulating the "temperature," or average kinetic energy, of the system, the formation of grains could be controlled to some degree.

1 One-Dimensional Chain

1.1 Basic Assumptions

This simulation considers a chain of atoms constrained to move in one dimension only. It constrains the atoms in the chain under Dirichlet boundary conditions and considers only nearest-neighbor interactions. The potential energy between a pair of atoms is modeled by the Lennard-Jones potential function[1]:

$$\varphi(r) = \varepsilon \left(\left(\frac{r_m}{r} \right)^6 - 2 \left(\frac{r_m}{r} \right)^{12} \right)$$

Where r is the distance between the atoms, ε is the depth of the potential energy well (i.e the negative of the value of the potential energy of the pair at its lowest point) and r_m is the spacing at which the pair attains its lowest potential energy. In this simulation ε and r_m are normalized to 1.

1.2 Evenly-Spaced Chain: Energy and Stability

The simulation started as a MATLAB function containing an array of positions x_i for the n atoms in the chain. It spaced the atoms equally by r_m (the natural spacing) and computed the total potential energy of the chain, $\sum_{i=1}^{n-1} \varphi(x_{i+1} - x_i)$. An array of random perturbations δ_i about two orders of magnitude smaller than r_m was then added to the position array and the energy was recomputed. Several random perturbations were tested; in each case the total potential energy was greater than the equally-spaced chain, as expected. This supports the assumption that the evenly-spaced chain was the lowest-energy, and therefore locally the most stable, configuration.

To more rigorously assess the stability of a chain configuration the Hessian matrix of the total-energy function is usually needed. If the eigenvalues of this matrix are all greater than zero then the function is at a stable equilibrium. This was applied to chains "stretched" (the position array was multiplied) by an assortment of scaling factors. It was determined that the evenly-spaced chain became unstable when stretched by a factor greater than approximately 1.15.

1.3 Fractured Chain

The evenly-spaced chain state is not the only equilibrium state, however. A "fractured" chain state, where some atom pairs are separated by a distance r_0 and others are separate by a larger distance r_f, is also an equilibrium. The r_f-spaced pairs are denoted "fractured bonds."

The constraints on the lengths r_0 and r_f are:

$$\begin{aligned} n_u r_0 + n_f r_f &= L \\ \varphi'(r_0) - \varphi'(r_f) &= 0 \end{aligned}$$

where n_u and n_f are the numbers of bonds of length r_0 and r_f, respectively, L is the total length of the chain $(x_n - x_1)$ and $\varphi'(r)$ is the derivative of the Lennard-Jones potential function. These equations were solved with MATLAB to obtain information on fractured states of the chain.

With this algorithm it was possible to make a bifurcation diagram of the chain's state, see Figure 1. This diagram only traces one specific type of fractured state, specifically the one with one fractured bond in the center bond of the chain.

2 Minimization Algorithms

While the solution of the above conditions can find fractured states in one-dimensional atom chains its usefulness diminishes in more realistic two- or three-dimensional lattices where the energy is influenced not only by the lengths of the bonds but the relationships between these

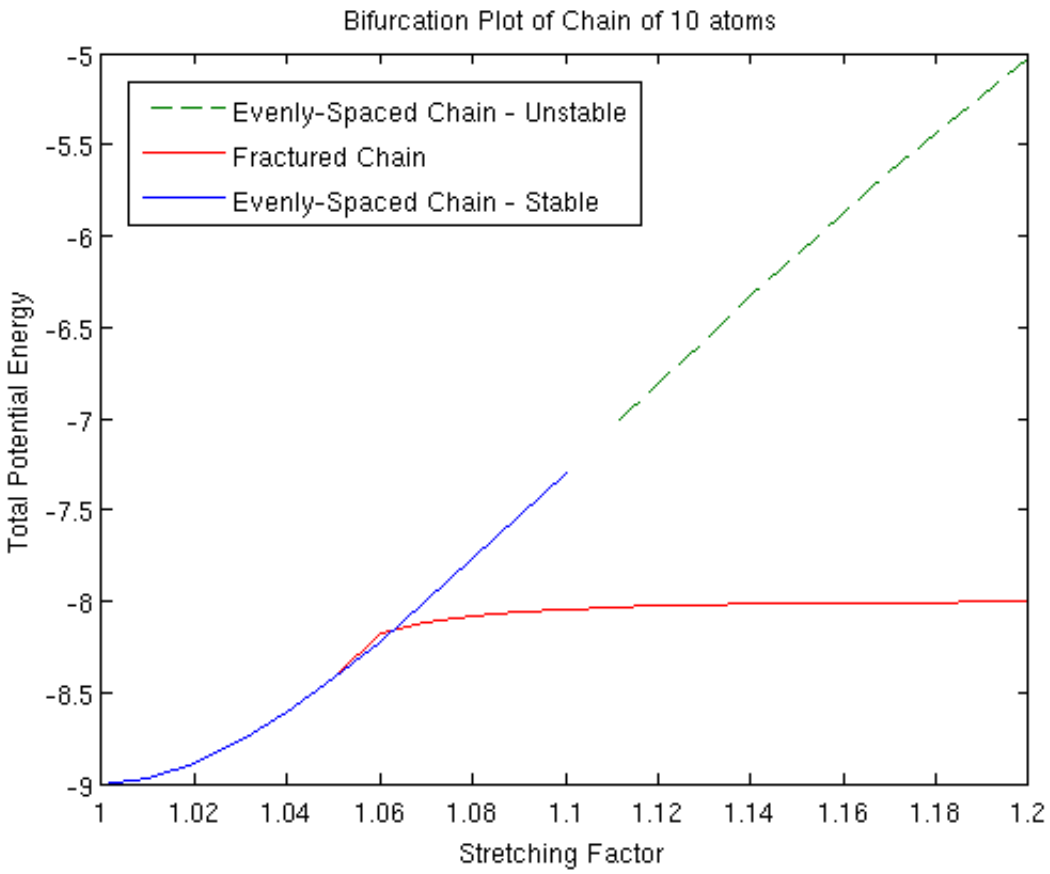

Figure 1: Bifurcation diagram of the chain's state. The red line represents the chain state where one bond in the center of the chain is fractured. This state was stable for all stretch factors in the range tested.

bonds, all exerting forces on atoms at different angles. A more general class of algorithms would be necessary in those cases, where the minimum of an overall function needs to be found.

One such minimization algorithm is called the "steepest-descent method," where steps are taken in the direction of the negative of the gradient until the minimum is reached (i.e. the gradient is small enough). The problem that exists with this method is that the length of the step still needs to be determined. In this case, simply scaling the length of the gradient was not enough; the algorithm only "bounced" around the minimum indefinitely.

A better class of algorithms for determining step size is the "linesearch" method, where the length of the next step is decided by the approximate minimum of the total-energy function constrained along the direction of the gradient. Formally, the length of the next step, α, is chosen so that α approximately minimizes the function $\phi(\alpha) = \mathrm{U}(\mathbf{x}_k - \alpha \nabla \mathrm{U}(\mathbf{x}_k))$ (not to be confused with the interatomic potential $\varphi(r)$), where $\mathrm{U}(\mathbf{x})$ is the total-energy

function of the chain and $\mathbf{x}_k$ is the current best guess as to the minimum of U. There are many different ways to find the approximate minimum of ϕ; the ones I used are described below.

Most algorithms use a set of conditions called the Wolfe conditions to determine whether the value of α is close enough to a minimum of ϕ to advance to the next step. These conditions, in the form used for the steepest-descent method, are (reproduced from [2]):

$$\begin{aligned} \mathrm{U}(\mathbf{x}_k - \alpha_k \nabla \mathrm{U}(\mathbf{x}_k)) &\leq \mathrm{U}(\mathbf{x}_k) + c_1 \alpha_k |\nabla \mathrm{U}(\mathbf{x}_k)|^2 \\ \nabla \mathrm{U}(\mathbf{x}_k - \alpha_k \nabla \mathrm{U}(\mathbf{x}_k)) &\geq c_2 |\nabla \mathrm{U}(\mathbf{x}_k)|^2 \end{aligned}$$

The first condition is called the "Sufficient-Decrease Condition" and the second is called the "Curvature Condition." c_1 and c_2 are constants with $0 < c_1 < c_2 < 1$.

2.1 Testing Conditions

The basic assumptions of the testing environment were the same as those described in Section 1.1 with one exception: The interatomic potential, φ, was modified to a simpler, quadratic version. The potential used was $\varphi(r) = (r - r_m)^2$. Using this potential has the advantages of having one universal minimum and knowing the minimum value beforehand (it is zero, achieved when the chain is spaced evenly by the natural spacing r_m). These methods were tested with a chain of 10 atoms spaced evenly by r_m, perturbed by a random disturbance on the order of $10^{-2} r_m$.

The algorithm is said to have "converged" or "terminated" when the gradient of the energy function $\nabla \mathrm{U}(\mathbf{x}_k)$ has magnitude less than or equal to a value called the "tolerance", or `tol` in the MATLAB code.

All tests were run on a computer with a 2.6 GHz AMD Athlon II X4 CPU, 4 GB of RAM, Ubuntu Linux 11.04, and MATLAB R2011a. The machine's floating-point precision limit (obtained by calling `eps` on the MATLAB command line) is approximately $2.2204 * 10^{-16}$.

2.2 Backtracking Method

This is an algorithm described in [2] that produces a value of α that satisfies the sufficient-decrease condition. The algorithm, in MATLAB code, is:

```
% Steepest-descent method
% Constants:
c = 10^-4;
rho = 0.9;
tol = 10^-6
clear alpha;
% Compute the gradient
gradk = gradU(pos, dfun);
% Initialize iteration counters
iter = 0;
```

```
11  lniter = 0;
    while norm(gradk) > tol
        iter = iter + 1;
        % Define line function (sometimes called ''phi'')
        line = @(alpha) totalU(pos - alpha*gradk, fun);
16
        %% Begin backtracking linesearch
        alpha = rm;
        while line(alpha) > line(0) + c*alpha*(gradU'*gradU)
            lniter = lniter + 1;
21          alpha = rho*alpha;
        end

        % Advance along the (opposite) direction of the gradient
        pos = pos - alpha*gradk;
26      % Recompute the gradient
        gradk = gradU(pos, dfun);
    end
```

The only difference between the various steepest-descent algorithms tested in this section is the way in which `alpha` is chosen. From now on, it will only be necessary to show the part of the algorithm that replaces lines 17-22 as well as the values of the constants (lines 2-5 in this code); nothing else will change.

The variables `iter` and `lniter` are to keep track of the number of iterations the algorithm needs to converge; `iter` counts outer iterations and `lniter` counts inner iterations. The function `totalU(pos, fun)` returns the total energy of the chain when given the positions of the atoms in `pos` and the interatomic potential function `fun`. The function `gradU(pos, dfun)` returns the gradient of the total-energy function when supplied with the positions of the atoms in `pos` and the derivative of the interatomic potential in `dfun`. The value `tol` is used to determine when the search algorithm is finished. When it was adjusted, it was only done on the order-of-magnitude level.

This algorithm worked well, converging to the minimum in 1014 iterations of the outer loop and 13189 total iterations of the inner loop. It converged much faster, and no less accurately, when `rho` was changed to 0.5. The algorithm only needed 455 outer iterations and 912 total inner iterations to converge under those conditions. The smallest value of the tolerance `tol` for which the algorithm would terminate was 10^{-14}.

2.2.1 Modification of Initial Guess

[2] describes an intelligent method for selecting an initial guess for α_k: Assume that the change in the objective function at step k will be the same as the change at step $k-1$, giving $\alpha_k = \alpha_{k-1}\frac{|\nabla \mathrm{U}(\mathbf{x}_{k-1})|}{|\nabla \mathrm{U}(\mathbf{x}_k)|}$. This value of α_k can be used as a starting point for the algorithm that is being used to find the final value of α_k that will be used to make step k. According to [2],

this method produces a better-scaled initial guess than a constant value used at each step.

Implemented in code, this initial-guess selection algorithm is:

```
% Constants:
global c1 c2;
c1 = 10^-4;
c2 = 0.9;
rho = 0.5;
tol = 10^-14;
golden = (1+sqrt(5))/2;
kappa = 1.3;

clear alpha;
% ---- Code Omitted -- More Initialization ---- %
while norm(gradk) > tol
    iter = iter + 1;

    % Finish step-size determination
    if ~(exist('alpha', 'var'))
        alpha0 = 0.5*rm;
    else
        alpha0 = kappa *alpha0 / norm(gradk);
    end

    % ---- Code Omitted -- Loop Body ---- %

    % Begin determining the next step size (req's info from two successive
    % iterations)
    alpha0 = alpha*norm(gradk);

    % Recompute the gradient
    gradk = gradU(pos, dfun);
end
```

The initial guess from this algorithm is stored in the variable `alpha0` for use by each of the algorithms in the body of this loop. I have tested each of the algorithms in this section using both a fixed initial guess and the dynamically computed guess from this algorithm.

I tested this initial guess on the backtracking algorithm. At first the value of `kappa` required some adjustment for the algorithm to converge to an acceptable tolerance. The constant started with a value of 1. The algorithm would not converge to a tolerance stricter than 10^{-4}. A value of 1.1 improved the tolerance dramatically to 10^{-14}. Further experimentation showed that the value $1 + 10^{-6}$ gave optimal results. With this value the algorithm converged to the same tolerance of 10^{-14} with the least amount of outer and inner iterations; it required only 262 outer iterations and 46 inner iterations. This means that in many of the

outer loop iterations the initial guess chosen satisfied the termination conditions of the inner loop, an indication that the initial step size selection algorithm was giving good guesses.

2.3 Golden Section Search

This is an algorithm described in [3] that finds the approximate minimum of a function by bracketing the minimum with points x_1 and x_2 ($x_1 < x_2$), choosing a test point x_3 in the middle of the interval so that the ratio $\frac{x_2 - x_3}{x_3 - x_1} = \varphi$, the golden ratio (again, not to be confused with the interatomic potential function $\varphi(r)$), and choosing another test point x_4 in the larger interval. Depending on the value of x_4 a new interval is chosen with endpoints and midpoints selected from the x_i so that the new interval also brackets the minimum. The value of x_4 is chosen so that both possible new intervals are the same size. The changed sections of MATLAB code are below (unchanged sections are marked as "Omitted"):

```
% Constants:
global c1 c2;
c1 = 10^-4;
c2 = 0.9;
rho = 0.5;
tol = 10^-10;
golden = (1+sqrt(5))/2;
% --- Code Omitted ---- %
    %% Golden Section Search
    % Bound step size
    x1 = 0; x2 = 0.6*rm;
    % Choose the midpoint according to the golden ratio
    x3 = x1 + (x2-x1)/(1+golden);
    % Calculate and store values of line function (for efficiency)
    ln3 = line(x3);
    % Termination: Interval is sufficiently small
    while abs(x2-x1) > 10^-1*rm
        lniter = lniter + 1;
        x4 = x1 - x3 + x2;
        ln4 = line(x4);
        % A bunch of logic to decide what the new interval is
        if ln4 > ln3
            if x4 > x3
                x2 = x4;
            else
                x1 = x4;
            end
        elseif ln4 < ln3
            if x4 > x3
```

```
30                  x1 = x3;
                else
                    x2 = x3;
                end
                x3 = x4;
35              ln3 = ln4;
            else % If ln4 == ln3
                x1 = x3; x2 = x4; x3 = x1 + (x2-x1)/(1+golden);
                ln3 = line(x3);
            end
40      end
        alpha = x3;
    % ---- Code Omitted ---- %
```

This algorithm was more efficient than the backtracking search with fixed initial guess. With the bounds of the initial interval adjusted properly the algorithm converged within 318 outer iterations and 1272 inner iterations. The strictest tolerance that could be used was 10^{-14}, the same as the other algorithm. This suggests that the lowest tolerance that can be used for this problem is algorithm-independent; however, this could just be a coincidence. The norm of a vector of the same length as the gradient in this problem and with all entries set to the machine's floating-point precision limit is approximately $7.0217 * 10^{-16}$.

The test was conducted again, this time setting `x2` to `alpha0` (the dynamically computed initial guess from 2.2.1) in Line 11. With the value of `kappa` at $1 + 10^{-6}$ carried over from Section 2.2 the algorithm failed to converge for tolerances less than 10^{-4}. Changing `kappa` to 1.2 improved the minimum tolerance to 10^{-14}. Other values within the approximate range 1.2 – 2.0 let the algorithm converge to the same tolerance. Larger values reduced the number of outer iterations at the cost of more total inner iterations. For example, with `kappa` at 2.0, the algorithm converged in 322 outer and 1307 inner iterations. With a value of 1.4, it took 370 outer and 1129 inner iterations, and with a value of 1.2 it took 382 outer iterations and 1154 inner iterations. Since `kappa` is acting as the upper bound of an interval in this case it could be increased to any arbitrary value without missing the minimum. However, the golden section search is designed for unimodal functions and the potential function in this problem may be multimodal. Because of this, if the initial interval contains multiple minima, the algorithm will converge to only one of them. Therefore, increasing the upper bound on the interval too much may introduce extraneous minima into the problem and yield an unexpected solution. This type of problem is not specific to the golden section search; however, it seems that the algorithm used here is particularly unreliable for multimodal functions when compared to the other algorihtms tested, see Section 3.1.2.

Depending on the computational cost of an outer versus an inner iteration the new step-length selection algorithm might have improved or worsened the performance of this algorithm, but not significantly.

2.4 Line Search for the Wolfe Conditions

This algorithm comes from [2] (Algorithm 3.2). The authors guarantee it to find a value of `alpha` which satisfies the strong Wolfe conditions if such a value exists. It is somewhat more complicated than the previous two methods; it requires a subroutine:

```
function alpha = lnzoom(alphalo, alphahi, pos)
% Subroutine necessary for the line-search algorithm from
% Nocedal & Wright

while true
    % Interpolate alpha from interval
    alpha = (alphalo + alphahi)/2;
    funval = line(alpha);
    if funval > fun0 + c1*alpha*dfun0 || funval >= line(alphalo)
        alphahi = alpha;
    else
        dfunval = -1*gradU(pos - alpha*gradk, dfun)'*gradk;
        if abs(dfunval) <= -c2*dfun0
            break;
        end
        if dfunval*(alphahi - alphalo) >= 0
            alphahi = alphalo;
        end
        alphalo = alpha;
    end
end
end
```

Note that this function has access to the same variable scope as the other code shown here. The main algorithm follows (as before, only the changed sections are shown):

```
% ---- Code Omitted ---- %
    %% Line Search from Nocedal & Wright
    alpha1 = 0.5*rm; alpha0 = 0; alphamax = rm;
    if alpha1 > alphamax
        alpha1 = alphamax;
        inc = 0;
    else
        % Limit number of bracketing iterations
        inc = (alphamax - alpha1) / 10;
    end
    % Pre-evaluations for efficiency purposes
    fun0 = line(0);
    funval = fun0;
```

```
    dfun0 = -1*norm(gradk);
    while true
        lniter = lniter + 1;
        funval_prev = funval;
18      funval = line(alpha1);
        % Certain conditons, including the Wolfe conditions, determine whethe
        % to begin the zoom phase
        if funval > fun0 + c1*alpha1*dfun0 || ...
                (funval >= funval_prev && lniter > 1)
23          alpha = lnzoom(alpha0, alpha1, pos);
            break;
        end
        % Evaluate the derivative of line on alpha1
        dfunval = -1*gradU(pos - alpha1*gradk, dfun)'*gradk;
28      if abs(dfunval) <= c2*dfun0
            alpha = alpha1;
            break;
        end
        if dfunval > 0
33          alpha = lnzoom(alpha1, alpha0, pos);
            break;
        end
        alpha0 = alpha1;
        % Increase alpha1 for next iteration
38      alpha1 = alpha1 + inc;
    end
% ---- Code Omitted ---- %
```

The algorithm above corresponds to the "bracketing phase" of linesearch algorithms described in [2] while the subroutine corresponds to the "selection phase."

With the constant initial value of `alpha1` the algorithm converged only to a tolerance of 10^{-4} regardless of the initial value of `alpha`. The problem remained when the algorithm from Section 2.2.1 was used to compute the initial guess for `alpha1`. Modification of the value of `kappa` did not help, although scaling of the initial guess is not likely to be an issue with this algorithm, since it will not miss any points satisfying the Wolfe conditions in the interval [0, `alphamax`].

2.5 Newton's Method

Newton's Method is a well-known algorithm for solving nonlinear equations. It is used to find roots of a function $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^n$. Each iteration can be expressed as:

$$\mathbf{x}_k = \mathbf{x}_{k-1} - [\mathbf{Df}(\mathbf{x}_{k-1})]^{-1}\mathbf{f}(\mathbf{x}_{k-1})$$

The sequence $\{\mathbf{x}_i\}$ that is produced from these iterations will normally converge to a root of $\mathbf{f}$. Which root it converges to depends on the choice of starting point. In the case of optimization the function whose root needs to be found is the gradient of the objective function, $\nabla \mathrm{U}(\mathbf{x})$. The derivative of this function is the second derivative of U, or the Hessian matrix, $\mathbf{D}^2\mathrm{U}(\mathbf{x})$. In this case, Newton's method becomes:

$$\mathbf{x}_k = \mathbf{x}_{k-1} - [\mathbf{D}^2\mathrm{U}(\mathbf{x}_{k-1})]^{-1}\nabla \mathrm{U}(\mathbf{x}_{k-1})$$

The convergence of Newton's Method is usually superlinear[4] but in higher dimensions it requires inverting a matrix, which can be quite an expensive operation if the matrix is large. For the one-dimensional atom chain, since only nearest-neighbor interactions are being considered, the Hessian matrix is tridiagonal making its inversion a less expensive operation than if the matrix were full. It seemed to be a good idea to try Newton's Method to see whether it would work somewhat efficiently in this situation. The MATLAB code for the algorithm is:

```
gradk = gradU(pos, dfun);
while norm(gradk) > tol
    iter = iter + 1;
    H = hessianU(pos, d2fun);
    % Trim the zero border of the Hessian and gradient
    H = H(2:n-1,2:n-1);
    gradk = gradk(2:n-1);
    % Find increment, replace zero border
    inc = [0; g*(H \ gradk); 0];
    pos = pos - inc;
    % Recompute the gradient
    gradk = gradU(pos, dfun);
end
```

This algorithm converged, as expected, in one step. This is typical behavior for Newton's method on a quadratic potential, since the method essentially approximates the function locally with a quadratic and advances to the minimum of that quadratic. For functions that are well approximated by quadratics (or functions that *are* quadratics, such as the one under test), the low number of iterations required to reach the minimum may well justify the high computational cost involved in the inversion of a large matrix.

3 Tests of Minimization Algorithms

While some of the above algorithms may work well under the idealized testing conditions from Section 2.1, their use in this project will be under more realistic conditions. Under each set of new conditions each algorithm was tested both with a static value of α_0 and the dynamically computed value from Section 2.2.1. The algorithm's parameters were adjusted

until it converged to the smallest tolerance possible for that algorithm. Once the mimimum tolerance was established other adjustments were made to improve the algorithm's computational efficiency.

Note that the variables and terminology used in this section have the same meaning as those used in Section 2.

3.1 Lennard-Jones Potential

For this set of tests the interatomic potential was changed to the Lennnard-Jones potential described in Section 1.1. This is a good test of the reliability of the minimization algorithms, since this potential has places with very small gradients that are not minima. The quadratic potential had one global minimum and the gradient of this potential always pointed inward, toward the minimum. With the L-J potential there is potential (no pun intended) for the algorithm to get "lost" in a place that is far from the desired minimum.

3.1.1 Backtracking

With a fixed initial value of $\alpha_0 = r_m/2$ the algorithm converged to a tolerance of 10^{-7}. Adjusting α_0 did nothing to improve this tolerance. The optimum value was $\alpha_0 = 0.25 * r_m$, where the algorithm took 104 outer and 494 inner iterations to converge. With values of α_0 less than $0.15 * r_m$ the algorithm could not converge to the established tolerance.

With α_0 dynamically computed the precision of the algorithm improved; a tolerance of 10^{-10} was reached in 1837 outer and 727 inner iterations. The value of κ (`kappa`) used was 1.3. This precision could only be maintained for a very small range of values of κ. At $\kappa = 1.29$ the efficiency of the algorithm improved to 1000 outer and 399 inner iterations.

To compare the relative efficiencies of the two versions of the algorithms (one with a fixed α_0, one with a dynamically computed value) the test of the version with the dynamic initial guess was repeated with the tolerance set tothe same value as was used in the version with the fixed value. The version with the dynamic value reached this tolerance in 192 outer and 102 inner iterations. With κ set to 1.1 it took 340 outer and 79 inner iterations. The best efficiency was reached when $\kappa = 1.0$: The algorithm took 106 outer and 33 inner iterations to converge. However, with this value of κ, the algorithm would not converge to a stricter tolerance than 10^{-7}.

Depending on the computational cost of an outer versus an inner iteration the version with dynamically computed α_0 could be more efficient than the version with a fixed initial guess. It is certainly more precise.

3.1.2 Golden Section Search

With x_2 initially set to r_m the algorithm performed poorly. It needed 10538 outer and 52690 inner iterations to converge to a tolerance of 10^{-3} and returned atom positions that were off the ends of the chain. The same behavior was observed with x_2 at $0.25 * r_m$. Setting x_2 to

$0.1 * r_m$ terminated the inner search much earlier, in most cases before the first iteration, and returned the same type of grossly inaccurate positions seen before.

To try to achieve better results the termination condition of the inner search was adjusted to $|x_2 - x_1| \leq 10^{-2} * r_m$. This algorithm did converge to a more reasonable set of positions in 117 outer and 883 inner iterations with a tolerance of 10^{-6}. Changing the value of x_2 did not improve these results.

With x_2 set to the dynamically computed α_0 the algorithm did not converge even to a tolerance of 10^{-2} and κ set to 0.1. Adjusting various constants did not help; the algorithm still did not converge.

Overall this algorithm was unreliable and did not perform well with a Lennard-Jones potential. One possible cause of this is that the golden section search was designed for unimodal functions. The quadratic potential used earlier was unimodal, so the search worked well. However, the Lennard-Jones potential is less predictable and has a small derivative at large radii, possibly making it unsuitable for the golden section search. As mentioned in Section 2.3 this problem is not specific to the Golden Section Search. However, this search seems less able to handle unusual potentials than the other algorithms tested here.

3.1.3 Wolfe Conditions Linesearch

This algorithm was still limited by the 10^{-4} tolerance but it converged quickly (in 46 outer and 62 inner iterations), possibly due to the high tolerance. Adjusting the constants in the algorithm to make the Wolfe conditions stricter did not improve the tolerance. The constant c_1 could not be adjusted higher than 10^{-1} without causing the algorithm to fail and loop infinitely. However, adjusting c_2 to 0.1 made the search faster, converging in 36 outer and 47 inner iterations. While this algorithm may be less precise than some of the others tested it is reliable (no positions were produced that were off the ends of the chain or otherwise unacceptable) and efficient.

3.1.4 Newton's Method

This algorithm exibited the fast convergence that is typical of Newton's Method: It took 6 iterations to converge to a tolerance of 10^{-10} with an elapsed time of less than 0.02s. When the tolerance was set to `eps`, the amount of variation produced by round-off error, the algorithm only required 1 additional iteration (in < 0.03s) to converge. This algorithm produced the most precise answers yet in an acceptable amount of time with no scaling or linesearches needed. If it did not involve inverting a large Hessian matrix it would easily be the best algorithm to use in this situation.

3.2 Deadloads

Another way to test these algorithms is to apply some virtual "forces" to the atoms and use the algorithms to calculate how these forces influence the atom positions. These forces are often known as "deadloads" since they are constant and do not change with the atom

positions. These forces are implemented by modifying the chain potential

$$\mathrm{U}(\mathbf{x}) = \sum_{i=1}^{n-1} \varphi(x_{i+1} - x_i)$$

to include a force vector $\mathbf{F}$:

$$\mathrm{U}_{\mathbf{F}}(\mathbf{x}) = \sum_{i=1}^{n-1} \varphi(x_{i+1} - x_i) - \mathbf{F}^{\mathrm{T}}\mathbf{x}$$

The gradient of this new potential is $\nabla \mathrm{U}_{\mathbf{F}} = \nabla \mathrm{U} - \mathbf{F}$. Since the gradient of the potential function represents the opposite of the forces on the atoms this new potential causes the atoms to act as if the forces in $\mathbf{F}$ are acting upon them in addition to the interatomic forces.

All algorithms tested above (except the golden section search, which proved unreliable on a non-quadratic potential) were tested with deadloads that acted on the two middle atoms and were directed away from each other. These loads were intended to force the two middle atoms apart causing a type of fracture.

Note that the loads described in this section are in arbitrary units; they only have meaning when compared to values of $\varphi'(r)$. For reference, the value of $\varphi'(1.5)$ is approximately 0.6407.

3.2.1 Backtracking Method

With a relatively small load on the center atoms (approximately the value of $\varphi'(1.5)$ for each atom) this algorithm did not converge to any reasonable tolerance; it only looped indefinitely. With a tiny load of 0.01 on each of the center atoms the algorithm did converge to a tolerance of 10^{-4}. The effect of these loads was barely visible in a graph of the atom positions.

The algorithm did work with larger loads with a loss of tolerance: The tolerance needed for a load of 0.05 each was 10^{-3} and for a load of 0.1 it needed a 10^{-2} tolerance.

3.2.2 Wolfe Conditions Linesearch

This algorithm was more reliable and could handle larger loads than the backtracking method. With loads of 0.1 on each of the center atoms the algorithm converged in 42 outer and 57 inner iterations. It worked with loads as large as 0.9 on each center atom. It looped indefinitely when the load was 1 or greater.

3.2.3 Newton's Method

With the tolerance set to `eps` this algorithm also looped indefinitely. It began to give results when it was set to the more conservative value of 10^{-10}. The algorithm required 6 iterations with loads of 0.1 on each center atom. It gave the same results as the Wolfe Conditions Linesearch when the loads were 0.9 per center atom. It still worked when the loads were as high as 8 per center atom. In this case the algorithm required 8 iterations to produce its results. When these results were plotted the fracture was clearly visible.

As a performance test this algorithm was run on a 100-atom chain with a tolerance of 10^{-10}, a Lennard-Jones potential, and no deadloads. It required 8 iterations and approximately 0.11 seconds to converge to this tolerance. Combined with the previous timed runs of the algorithm this suggests a linear increase in complexity with respect to the number of atoms in the chain, which futher increases the appeal of Newton's Method for use in this situation.

4 Two-Dimensional Lattice

Expanding the chain simulation discussed in the previous sections to include another dimension introduces a multitude of new problems that were not present or could simply be ignored in the one-dimensional case.

The first consideration in making a 2-D lattice simulation is how to store the data. One way might be in a 3-dimensional array, assigning integer coordinates to each atom in the lattice and storing the 2-dimensional position of each atom in a place in the array based on those coordinates. This storage scheme creates an array that resembles the lattice in its shape, making certain operations and accesses easier. The downside to this scheme is that it relies on a specific constant shape for the lattice; if this shape is altered in any way (i.e. by allowing the atoms to drift freely in accordance with Newton's Laws) the shape of the array could become meaningless. Instead this simulation assigns a single permanent index to each atom in the lattice. The positions are stored in a 2-dimensional array: The atoms are indexed along the first dimension and individual X- or Y- coordinates are indexed along the second dimension.

Deciding which atoms interact is also more complex. In the one-dimensional case, the nearest-neighbor interaction model can be used relatively successfully. It only breaks down when the order of the atoms is rearranged due to extreme conditions in a minimization algorithm or simulation. In two dimensions each atom has multiple atoms with which it interacts and there is no way to determine which atoms these are from the method of indexing used in this simulation. Because of this the list of which atoms are considered neighbors needs to be computed and stored separately. There are several approaches to this problem; the code written and used for this simulation implements two: The "full" algorithm computes the distances between every atom pair in the lattice; if two atoms are closer than a given distance r_{cut} they are considered neighbors. This algorithm's computational complexity is $O(n^2)$. The "binning" strategy draws a grid of rectangular "bins" around each atom with side length r_{cut}. Any atoms inside the bins closest to a given atom are considered neighbors of that atom. With this strategy, any two atoms separated by a distance of less than r_{cut} are guaranteed to be neighbors. This algorithm is $O(n)$ but it may include more atoms as neighbors than are necessary to maintain simulation accuracy.

A third consideration is computation of the gradient of the potential-energy function of the lattice. The i-th component of the gradient is now the sum of all the interactions of the neighboring atoms. The negative of the force on atom i due to atom j is $-\varphi'(|x_j - x_i|)\frac{x_j - x_i}{|x_j - x_i|}$. Summing up the negatives of these interaction forces for all of one atom's neighbors yeilds

the gradient of the potential energy of the lattice with respect to that atom.

4.1 Simulation of Lattice Mechanics using Newton's Laws

With an atom lattice it is interesting to simulate the motion of the atoms of the lattice over time according to Newton's Laws instead of simply finding a configuration that minimizes energy. Such a simulation can more realistically track the evolution of the states of a lattice from an initial configuration. Properties such as stability of a given configuration can be inferred from the behavior of the lattice in the simulation.

There are many algorithms used to simulate the mechanics of atoms interacting under certain potentials[6]. The one used in this project to calculate future states of the lattice is called the "Störmer-Verlet Method." A single step in the algorithm is defined (assuming the masses of each particle in the system are set to 1) as[5]:

$$\begin{aligned} \mathbf{p}_{n+\frac{1}{2}} &= \mathbf{p}_n - \frac{\Delta t}{2}\nabla \mathrm{U}(\mathbf{q}_n) \\ \mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t \mathbf{p}_{n+\frac{1}{2}} \\ \mathbf{p}_{n+1} &= \mathbf{p}_{n+\frac{1}{2}} - \frac{\Delta t}{2}\nabla \mathrm{U}(\mathbf{q}_{n+1}) \end{aligned}$$

where $\mathbf{q}_n$ is the vector containing the positions of all the atoms in the lattice at step n, $\mathbf{p}_n$ is a similar vector containing momenta (or, equivalently, velocities), and Δt is the amount of time that passes between one step and the next.

This algorithm is known for its long-term stability. For example, it has the property of effectively conserving the total energy of a system (potential + kinetic), making it useful for studying the transfer of energy in a lattice structure.

4.1.1 Recalculation of Neighbors

At some point the atoms in the lattice will have moved enough that the list of neighbors (the atoms that interact) will be out of date. One way to determine when the list of neighbors needs to be updated is to keep track of the distance each atom has moved. Assuming all atoms initially separated by a distance less than or equal to $(1+\epsilon)r_{cut}$ interact one can easily calculate the distance each atom has moved since the last time the neighbors list was updated. If the sum of the largest two distances is greater than ϵr_{cut} then, in the worst case, the two atoms corresponding to those distances could have moved from being separated by a distance slightly greater than $(1+\epsilon)r_{cut}$ (non-interacting) to being separated by a distance of r_{cut} or less (the maximum non-interaction separation). In this case the neighbors list should be recalculated since there is a possibility of a new interacting pair.

One consequence of using this method is that each atom has an ϵr_{cut} wide "buffer zone" within which another atom may or may not interact. The value of $\varphi'(r_{cut})$ is often small enough that this inexactness does not greatly impact the overall accuracy of the simulation. Usually, however, a cut-off potential is used in place of φ that is continuous and zero for all $r > r_{cut}$. In this case the atom pairs whose separation is less than $(1+\epsilon)r_{cut}$ but greater than

r_{cut} would have their interactions computed but the force between them would evaluate to zero.

Use of this method saves a considerable amount of computational resources. In one simulation of 1500 steps the neighbors list only had to be recalculated 22 times.

4.2 Simulation conditions

A Newton's Laws simulation was run on a 2-dimensional lattice of atoms, initially in a 20-by-20 square configuration with atoms separated by a distance of 1. More precisely, an atom was placed on each set of coordinates (i, j); $\quad i, j \in \mathbb{Z} \cap [0, 19]$. The value of r_{cut} was set to 5 and ϵ was 0.25. The potential energy was calculated as the sum of the 2-body Lennard-Jones potential over all interatomic bonds (interacting pairs); bond angles were not taken into account. The gradient was calculated as described above. Neighbor lists were recalculated using the "full" algorithm, at the steps determined by the method in the previous section. To disturb the unstable equilibrium of a square lattice each atom was given an initial momentum on the order of 10^{-2} per coordinate. The time Δt between one step and the next was 0.01 time units. The simulation was usually run for 1500 steps or 15 time units.

These simulations were all run on the same computer as described in Section 2.1.

4.3 Simulation Results

The simulations took a long time to complete, about 25 minutes per simulation. This was due chiefly to the computational cost of computing the gradient of the energy function; the neighbor recalculations barely factored into the overall time. Each gradient computation took 0.1–0.2 seconds; neighbor recalculations took less than 0.02 seconds. An attempt to parallelize the gradient computation to increase its speed proved unsuccessful. The algorithm could have been speeded up by selecting a smaller value of r_{cut} at the expense of a minor loss in accuracy.

The results computed in the simulation are illustrated in Figure 2. The energy of the system over time is plotted in Figure 3.

In the simulation the lattice, once destablilized, reorganized itself into small regions of hexagonal patterns ("grains") separated by irregular regions. This is strong evidence that a hexagonal structure minimizes the energy of a lattice. The grains appeared to be consolidating somewhat, forming a larger hexagonal lattices separated by smaller irregular regions. Speculation would indicate that if the simulation were allowed to continue long enough the atoms might eventually rearrange themselves into a single hexagonal lattice.

There is a problem with this prediction, however. There appear to be instances where a hexagonal lattice pattern was (at least partially) dissolved and returned to chaos. A graph of the energy of the system reveals a possible reason for this behavior: Excess kinetic energy could be causing vibrations in the lattice structure, disrupting the bonds. Since the Störmer-Verlet algorithm preserves total system energy the potential energy lost when the atoms reorganized was converted to kinetic energy. This energy manifests itself in vibrations of the atoms, possibly acting as a hindrance or disruption to the formation of a single hexagonal lattice.

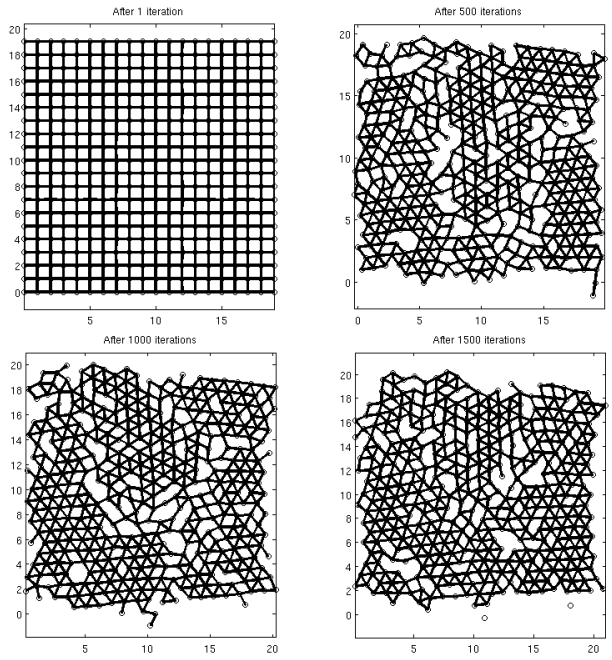

Figure 2: Positions of the atoms in the lattice at various points in the simulation. Lines or "bonds" are drawn between any two atoms separated by a distance less than or equal to 1.1.

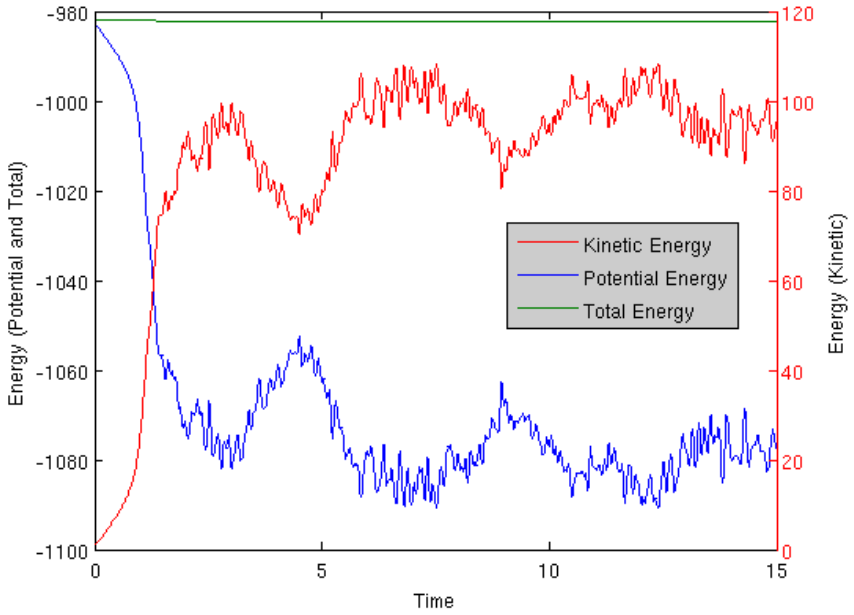

Figure 3: Kinetic, potential and total energy of the lattice system during the simulation.

4.4 Langevin Dynamics

The above simulation used the assumption that these atoms are situated in a vacuum, interacting with no atoms outside the simulation. In the real world this is seldom the case. A modification to the molecular dynamics equations, called Langevin Dynamics, attempts to approximate the effects of outside interference on the movement of the atoms in the simulation. Two terms are added to the force experienced by an atom: A "drag" term that always acts opposite and proportional to the atom's momentum, and a "fluctuation" term, or a normally-distributed random term. The "drag" term is meant to simulate the constant leakage of kinetic energy out of the system and into the surroundings. The "fluctuation" term accounts for the fact that the system's surroundings have a positive temperature and the surrounding atoms are transferring energy into the system by randomly colliding with the system's atoms. The Störmer-Verlet algorithm can be modified to include these terms; the modification is called the BBK Algorithm[6]:

$$\begin{aligned}
\mathbf{p}_{n+\frac{1}{2}} &= \mathbf{p}_n - \frac{\Delta t}{2}\nabla \mathrm{U}(\mathbf{q}_n) - \xi\frac{\Delta t}{2}\mathbf{p}_n + \frac{\sigma\sqrt{\Delta t}}{2}\mathbf{R}_n \\
\mathbf{q}_{n+1} &= \mathbf{q}_n + \Delta t\mathbf{p}_{n+\frac{1}{2}} \\
\mathbf{p}_{n+1} &= \mathbf{p}_{n+\frac{1}{2}} - \frac{\Delta t}{2}\nabla \mathrm{U}(\mathbf{q}_{n+1}) - \xi\frac{\Delta t}{2}\mathbf{p}_{n+\frac{1}{2}} + \frac{\sigma\sqrt{\Delta t}}{2}\mathbf{R}_n
\end{aligned}$$

where ξ and σ are constants (σ actually varies with the mass of the atom; however, since the masses of all the atoms in the system are the same, one constant σ suffices). $\mathbf{R}_n$ is a vector of i.i.d. random normal variables the same size as $\mathbf{q}$ and $\mathbf{p}$.

4.4.1 Results

Several trials were run with these modifications. Initially the drag coefficient ξ was set to 0.3 and σ was left at zero with the intended result that kinetic energy would be gradually bled out of the system and the lattice could more easily settle into a hexagonal state. This was indeed the case, as illustrated by Figures 4 and 5. If the simulation were left running indefinitely it is possible that the atoms would form one complete hexagonal lattice and that the vibrations would nearly cease (the kinetic energy would become nearly zero). However, it is interesting to note that a small grain that is cut off from the main lattice still exists. To investigate whether this structure was stable or not a simulation with identical initial conditions was run for 40 time units instead of the usual 15. Figures 4 and 5 actually display the data from this extended simulation. As these figures indicate, the dislocated grain and the grain boundary are likely actually stable structures. The kinetic energy has been drained to nearly zero by time 40 so there is little vibration in the lattice that could upset these structures. This structure exhibits a similarity to real-world grain boundaries in polycrystalline materials[7].

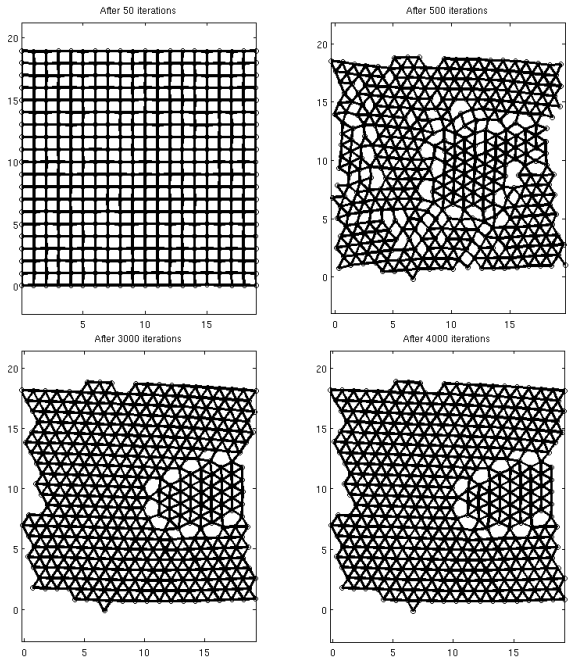

Figure 4: Lattice configurations for a simulation with $\xi = 0.3$ and $\sigma = 0$.

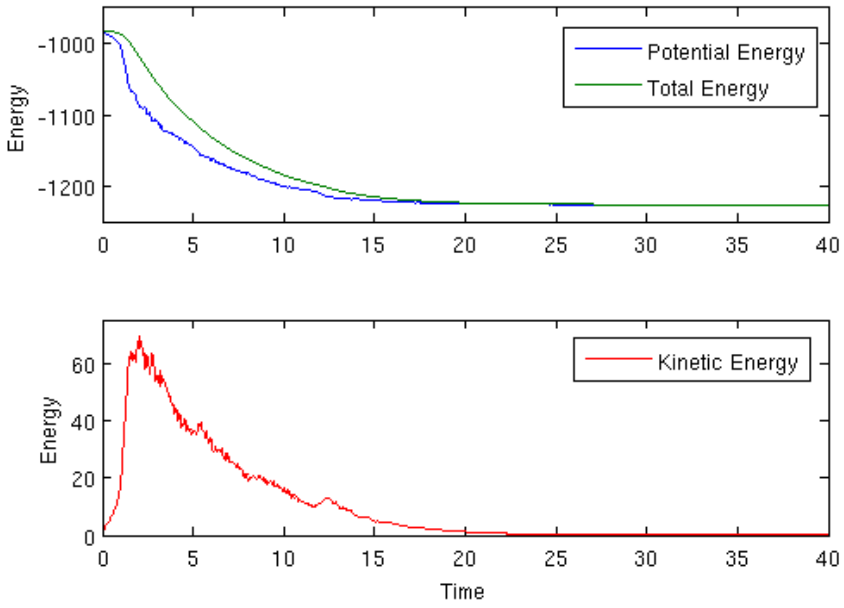

Figure 5: Energy of a system simulated with $\xi = 0.3$ and $\sigma = 0$.

In another trial the drag coefficient was left at 0.3 but the fluctuation coefficient was set to 0.1. The result was a stochastic, unstructured system (Figure 6). By time 15 the entire system expanded in width by a factor of 3, relatively few atoms were within a distance of 1.1 of each other, and there appeared to be no general pattern or order to the atom positions. The energy graph (Figure 7) is also much different from the previous graphs: As the simulation progressed the potential energy steadily increased and approached zero. The kinetic energy steadily increased, eventually reaching a stable value around which it oscillated randomly, probably due to the influence of the Langevin fluctuation term. This large amount of kinetic energy is most likely responsible for the observed behavior of the atom positions; if the atoms have too much kinetic energy they will be unable to form bonds. The high-energy vibrations apparently prevented any stable structures from forming and disrupted any stable structures that did manage to form. This state might be described as a "melted" or even "vaporized" state, where the kinetic energy of the system is too great for stable bonds to form.

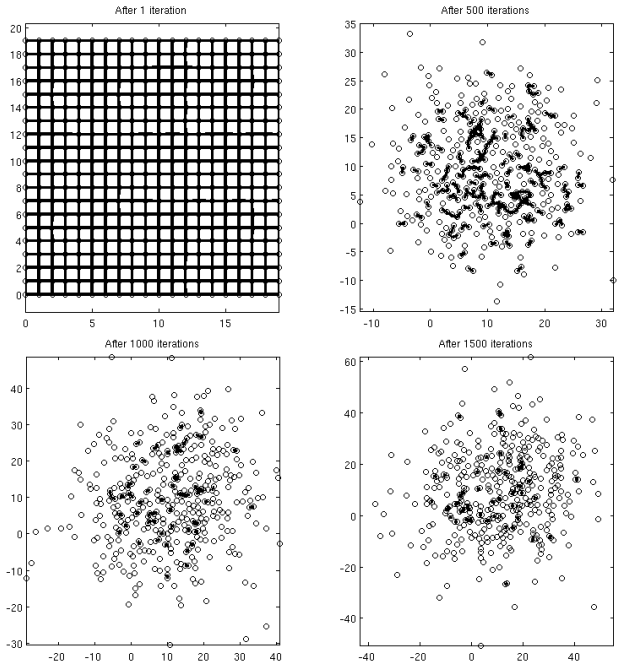

Figure 6: Lattice configurations for a simulation with $\xi = 0.3$ and $\sigma = 0.1$.

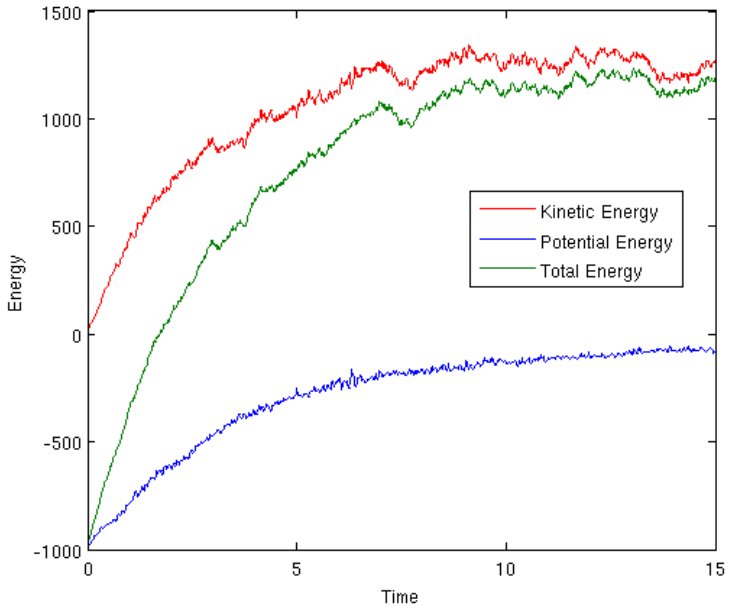

Figure 7: Energy of a system simulated with $\xi = 0.3$ and $\sigma = 0.1$.

When the fluctuation coefficient was set to 0.01 the lattice remained solid. Figure 8 shows the results. The lattice formation process resembles the trial with $\xi = 0.3$ and $\sigma = 0$ (Figures 4 and 5). However, in this trial, the kinetic energy appears to approach a positive nonzero value, indicating small vibrations in the lattice (probably due to the influence of the fluctuation term in the Langevin equation). Another simulation was run under the exact same initial conditions for a longer period of time, in this case 40 time units or 4000 steps. As Figure 9 shows, the kinetic energy eventually leveled off to a positive, stable value. It appears that the simulation reached a stable state by the time 40 time units passed. Figure 10 provides a closer look at the long-term behavior of the kinetic energy. While the K.E. fluctuates significantly between 12 and 16 energy units the overall trend, here illustrated by a least-squares linear fit, shows that the energy has essentially reached a stable value. The fluctuations could be due either to changes in the potential energy due to the vibration of atoms in the lattice or to the external fluctuations introduced by the Langevin fluctuation term.

Overall, these results are close to what would be expected and in some cases are even similar to real-world phenomena, indicating the BBK algorithm was well-behaved in the case of a 2-D system of atoms interacting under the Lennard-Jones potential only. If it were not for the relatively large computational cost of computing the gradient at each step of the algorithm it might be a more useful tool in studying the dynamics of large atom lattices.

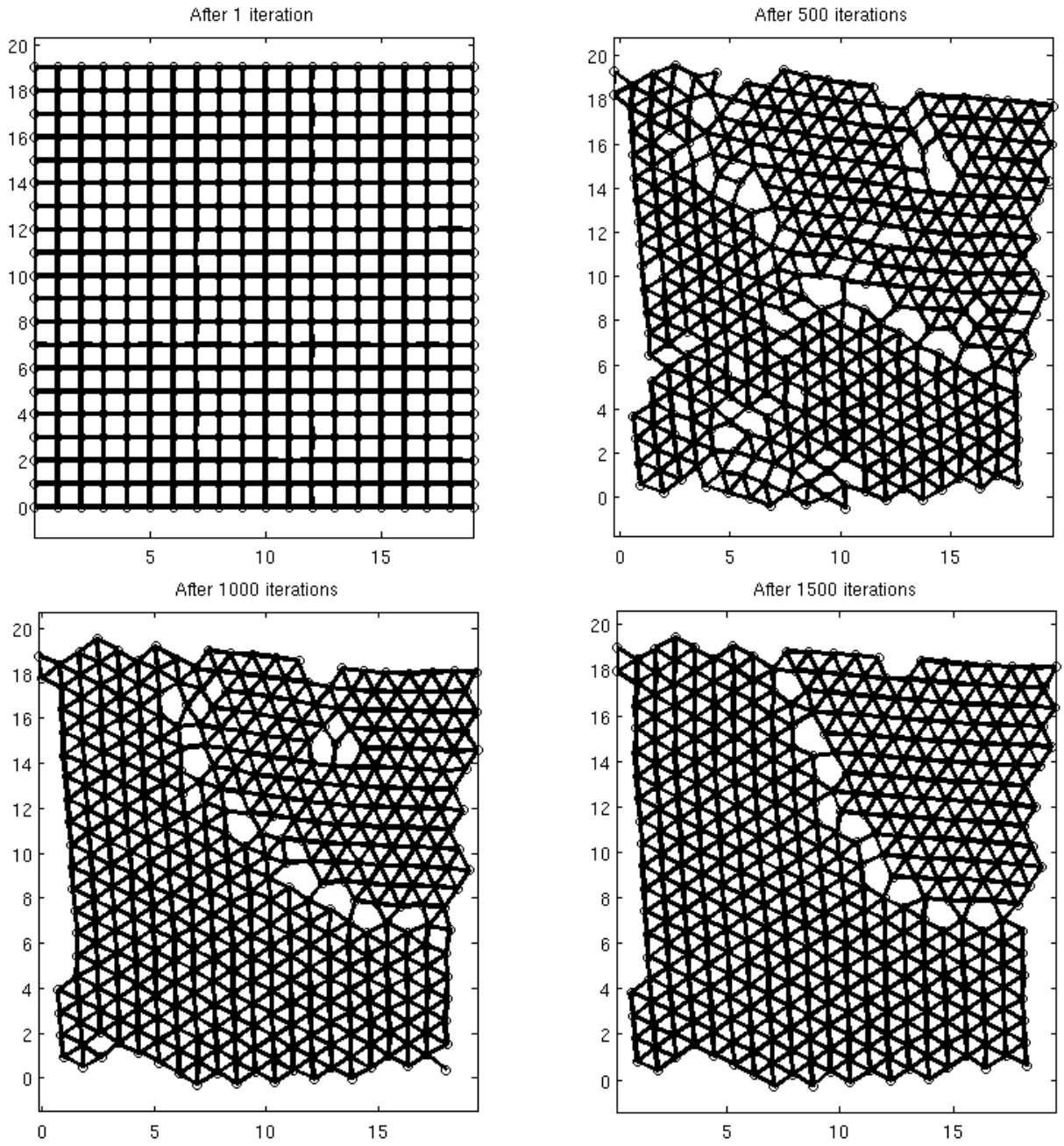

Figure 8: Lattice configurations for a simulation with $\xi = 0.3$ and $\sigma = 0.01$.

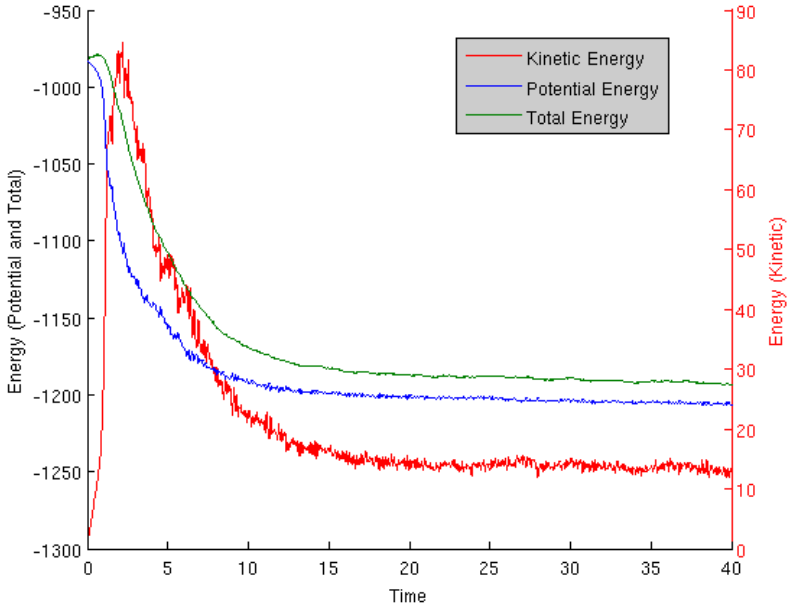

Figure 9: Energy of a 40-second run of a simulation with $\xi = 0.3$ and $\sigma = 0.01$.

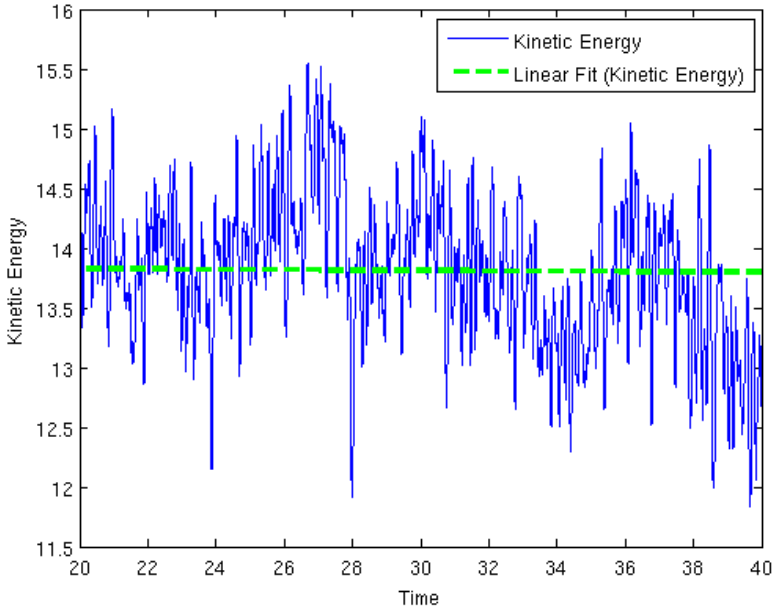

Figure 10: Long-term behavior of the kinetic energy of the simulation with $\xi = 0.3$ and $\sigma = 0.01$, along with a least-squares linear fit of the data.

5 Future Directions

This project has produced some interesting results as well as a substantial codebase for the study of atom lattices. However, it has only scratched the surface as to what research can be done in the field of computational molecular dynamics. This section outlines some possible extensions and applications of the results of this project.

One direction in which this research could be continued is to expand to three-dimensional lattices. The MATLAB programs written for the 2-D lattice simulation have been written in such a way so that they can be used on 3-D structures with little or no modifications. One could investigate what kind of regular 3-D lattices are stable and toward what types the unstable lattices evolve. This type of research might eventually yield practical results about the behavior of real-world crystals. The main problem with such research is that the simulation was extremely slow on a small two-dimensional lattice, so it would probably be next to impossible to simulate a 3-dimensional lattice of approximately the same size in a reasonable amount of time. There are a few ways to improve the speed of the simulation: A more powerful computer could be used, the code could be parallelized, or the algorithms could be rewritten in a lower-level language such as C.

Another direction to expand would be to use a more complex interatomic potential to make other stable structures. For example, graphene is an unstable structure under pure Lennard-Jones interactions as evidenced by a simulation using a graphene-like lattice as the starting structure. With the appropriate modifications to the potential function, including the addition of bond-angle potentials, a stable graphene structure could be formed and its properties could be investigated.

A combination of the previous two extensions could lead to research on carbon nanotubes, structures formed essentially by rolling sheets of graphene into cylinders. A program for generating the positions of atoms in a nanotube was written for this project but it went unused. With the proper potential energy models it could be made into a stable structure. A variety of research projects could then be done with this model, perhaps some with real-world applications. For example, tension or compression could be applied to the nanotube model to study its theoretical failure modes.

Finally, there is much more that can done in the field of computational thermodynamics than simple Langevin dynamics. A variety of numerical methods were described in [6]; however, the only tests of the method that were described were done on a pentane molecule with a complex potential. It could be interesting to apply those methods to a lattice structure and observe the results.

References

[1] "Lennard-Jones potential" from Wikipedia, accessed 05 July 2011 (`http://en.wikipedia.org/Lennard-Jones_potential`)

[2] J. Nocedal and S. J. Wright. *Numerical Optimization.* Springer Verlag, New York, NY 1999.

[3] "Golden section search" from Wikipedia, accessed 22 July 2011 (`http://en.wikipedia.org/wiki/Golden_section_search`)

[4] B. B. Hubbard and J. H. Hubbard. *Vector Calculus, Linear Algebra, and Differential Forms.* 4th Edition. Matrix Editions, Ithaca, NY 2009.

[5] B. Leimkuhler and S. Reich. *Simulating Hamiltonian dynamics.* Cambridge Monographs on Applied and Computational Mathematics, 14. Cambridge University Press, Cambridge, 2004.

[6] E. Cancès, F. Legoll and G. Stoltz, *Theoretical and numerical comparison of some sampling methods for molecular dynamics*, Mathematical Modelling and Numerical Analysis, vol. 41 (2), 351-389 (2007)

[7] "Grain boundary" from Wikipedia, accessed 2 September 2011 (`http://en.wikipedia.org/wiki/Grain_boundary`)

6 Acknowledgements

I would like to thank Professor Mitchell Luskin and Brian Vankoten in the Shool of Mathematics at the University of Minnesota for their invaluable assistance during this project, including discussions of problems to investigate, instruction in the mathematics necessary for me to investigate those problems, and recommendations of literature relevant to the project.

This project was supported by the University of Minnesota's Undergraduate Research Opportunities Program.

A MATLAB programs

You may request an electronic copy of the MATLAB programs I wrote and used for this research project by e-mailing me at `veit0044@umn.edu`.