Simulating atomic structures of metallic nanowires in the grand canonical ensemble

C.J. Flack

Thesis advisors:
J. Bürki
C.A. Stafford

Department of Physics, University of Arizona, 1118 E. 4th Street, Tucson, AZ 85721, USA

Abstract
Monte-Carlo simulated annealing methods have been utilized to determine the atomic structure of metallic nanowires. To improve upon previous simulations and to better model the experimental situation, where the wire is suspended between two metal electrodes, open boundary conditions were used instead of periodic boundary conditions. A simulation within the grand canonical ensemble has been developed, allowing the wire to exchange atoms with the contacts. For a wire of conductance $G = 3G_o$, a helical hollow shell is found to be the equilibrium structure.

1 Introduction

Nanowires are of principal interest in the further development of nanoelectronic technologies, and an understanding of their atomic structure has important implications in understanding the fundamental physics driving their properties.

Early theoretical studies of nanowire structures predicted non-crystalline structures of either icosahedral packing or a helical multishell [1]. These simulations used a classical pair-wise interaction energy, which produces wires unstable to Rayleigh instabilities. Synthesis of gold nanowires by Kondo et al. [2] via an ultra-high vacuum-TEM electron beam thinning provided evidence of multi-shell helical structures in coaxial tubes.

The jellium model for conduction electrons describes a theory of nanocohesion where conductance channels bind together metallic atoms [3]. This theory leads to the

nanoscale free-electron model (NFEM) [4], which states that the electron gas stabilizes metallic wires and generates a confinement potential for the atoms.

In this article, I'll address the use of the Monte-Carlo simulated annealing method for modeling the atomic structure of metallic nanowires to determine the idealized equilibrium structures. I'll discuss the limitations of the canonical ensemble for this type of simulation and the difficulties of implementing the grand canonical ensemble as well as simulated equilibrium structures.

This article is organized as follows: Sect. 2 will detail the nanowire model being used. Sect. 3 will discuss the Monte-Carlo simulated annealing method, its use in the canonical ensemble, and predicted equilibrium structures. Motivations for use of the grand canonical ensemble, implementation, and obtained atomic structures are discussed in Sect. 3. Some concluding remarks are given in Sect 4. Finally, the program source is included as an appendix.

2 Model

There is experimental evidence that the structural stability of metallic nanowires is well described by the nanoscale free-electron model (NFEM) [3,4]. The wire cannot be modeled classically with pair-wise interactions between atoms as per previous studies [1,2], as this leads to Rayleigh instability. Instead the structural stability of the wire is largely due to interplay between surface and quantum-size effects, generating a confinement potential due to the electron gas.

Cylindrical nanowires serve as waveguides for conduction electrons, where the conductance is defined by the number of electron channels lying below the Fermi energy.

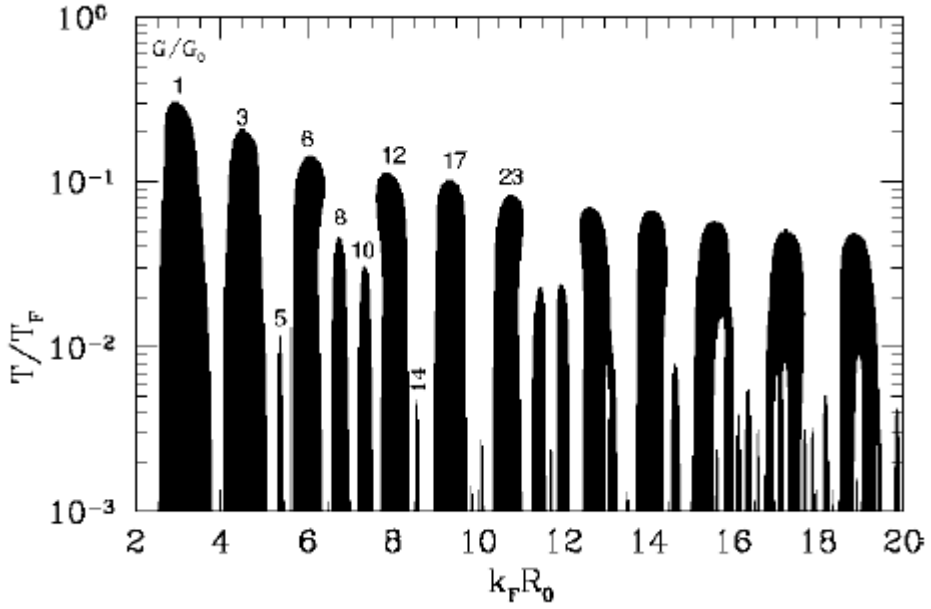


FIG. 1 Stability diagram for cylindrical nanowires. Diagram courtesy of Ref. [4]

The conductance is of the form $G \approx N_C G_o$, where G_o is the conductance quanta, $G_o = 2e^2/h$, where h is Planck's constant and e is the electron charge. The number of conductance channels can be roughly approximated by the Sharvin formula:

$$N_C \approx G/G_o \approx \frac{1}{4}(k_F R)^2 \tag{1}$$

where k_F is the Fermi wave number. Cylindrical nanowires are found within the NFEM to be stable with a number of conductance channels equal to magic conductance values [4]. Fig. 1 shows the linear stability for values of $k_F R_o$ and T/T_F, with T_F being the Fermi temperature such that $\varepsilon_F = kT_F$, where ε_F is the Fermi energy.

Considering an axisymmetric wire in cylindrical coordinates, the total energy of the system can be given by:

$$E(\{\vec{r}_i\}) = \sum_i U(\rho_i, z_i) + \frac{1}{2}\sum_j \sum_{i \neq j} E_{\text{int}}(R_{ij}) \tag{2}$$

Metal	ε_F (eV)	k_F (1/Å)	R_S (1/k_F)	A ($\varepsilon_F k_F$)	ε (eV)	a (Å)	m
Na	3.24	0.92	1.91916	0.31	0.05	4.23	10
Au	5.53	1.21	1.43837	0.26	0.012793	4.08	10

TABLE 1 Table of material dependent units and constants: the Fermi energy of the metal, ε_F; the Fermi wave number, k_F; the atomic radius, R_S; the strength of the confining potential, A; constants for the hard-core repulsive energy: ε, a, m (see Eq. (5)).

The factor of $\frac{1}{2}$ corrects for double counting. The kinetic energy of the atoms is neglected, as the goal is to obtain the equilibrium structures, and kinetic energy is zero in equilibrium.

The confinement potential for a single atom can be written as:

$$U(\rho,z)=\frac{A\left(\rho^2-R^2\right)}{R}+U_{end}(z) \tag{3}$$

where

$$U_{end}(z)=\frac{A}{R_s}\begin{cases}(z-R_S)^2 & z\le R_S\\ (z-L+R_s)^2 & z\ge L-R_S\\ 0 & otherwise\end{cases} \tag{4}$$

where R represents the radius of the wire, L represents its length, A is a constant representing the strength of the confinement potential, and R_S is the atomic radius. All lengths are expressed in units of $1/k_F$ where k_F is the Fermi wave number. Energies are expressed in units of the Fermi energy, ε_F. R_S, A, $1/k_F$, and ε_F are all material dependent. Values for Na and Au are shown in Table 1. The first term of Eq. (3) represents an accurate approximation of the confinement potential obtained as a solution to Poisson's equation using the electron density for the confinement potential. Eq. (4) is a parabolic confining potential for the ends of the wires.

The interaction energy between atoms is defined by a phenomenological hard-core repulsion and a screened Coulomb force:

$$E_{\text{int}}(R_{ij}) = \varepsilon\left(\frac{a}{R_{ij}}\right)^{m} + \frac{e^{-R_{ij}/L_s}}{4\pi\varepsilon_o} \tag{5}$$

where R_{ij} represents the distance between the ith and jth atoms. ε, a, and m are material dependent constants, defined in Table 1, and L_S is the Thomas-Fermi screening length [5] defined as:

$$L_S = 0.815\sqrt{\frac{R_S}{a_o}} \tag{6}$$

where a_o is the Bohr radius, expressed in units of $1/k_F$.

3 Simulated Annealing in the Canonical Ensemble

Monte-Carlo simulated annealing methods use random displacements with a slow cooling method to reach a minimum energy configuration, ideally moving past local extrema to reach the global minimum [6]. The process imitates the annealing of actual metals. Beginning at a high temperature, the constituent atoms have high thermal mobility, allowing them to explore the configuration space. The atoms favor configurations with the lowest local energy. As the temperature is slowly decreased, thermal mobility is lost and atoms are frozen into a minimum energy configuration.

From the Metropolis algorithm [7], new configurations are generated from a random displacement of atoms. The probability of the move being accepted is defined from the Boltzman factor as:

$$p(E) = \exp\left(-\Delta E / kT\right) \tag{7}$$

where ΔE is the change in energy between the new and old configurations, and kT represents the system temperature. Moves where the system's energy is decreased are automatically accepted. A finite probability exists to accept moves that increase energy, permitting the system to escape from local minima.

3.1 The Monte-Carlo simulated annealing program structure

The algorithm used for these simulations generates a random initial configuration of atoms chosen of uniform density from a specified wire radius and number of atoms. Atom placement is constrained to within the wire radius and length, defined by the following:

$$N = \pi R^2 L \frac{k_F^3}{3\pi^2} \tag{8}$$

where N represents the number of atoms, and $\frac{k_F^3}{3\pi^2}$ is the atomic number density for a monovalent metal.

Monte-Carlo moves are conducted within the canonical ensemble, where V, N, and T are externally controlled parameters. Each Monte-Carlo move is a random displacement of one atom in spherical coordinates, so that the move is isotropic without a preferred direction. The move is made through a selection of three parameters: dr, θ, and φ. θ and φ are randomly selected from a uniform distribution within their domain in spherical coordinates, [0,π] and [0,2π], respectively. dr is randomly selected according to the following:

$$dr = \sigma_R \sqrt{\frac{kT}{kT_o}} norm(r) \tag{9}$$

where $norm(r)$ generates a random number on a normal distribution by a routine from Numerical Recipes [8], σ_R is global parameter, kT is the current temperature, and kT_o is the starting temperature. This selection simulates the classical Maxwellian distribution of velocities. New atomic positions are calculated as follows:

$$\begin{aligned} x &= x_o + dr \sin\theta \cos\varphi \\ y &= y_o + dr \sin\theta \sin\varphi \\ z &= z_o + dr \cos\theta \end{aligned} \tag{10}$$

where x_o, y_o, and z_o refer to the old positions. The probability of the move being accepted is defined according to Eq. (6). ΔE is computed as the energy difference between the new and old positions. A random number between 0 and 1, non-inclusive, is compared to the resultant probability. If that number is less than the probability, the move is accepted.

A simulation begins with an initial temperature kT_o. Before annealing, Monte-Carlo steps are performed sequentially on all atoms a preset number of times in order to relax abnormally small interatomic distances that may arise from a random initial configuration. When annealing begins, the same procedure is followed, with kT decreased by 10% after a predetermined number of move attempts. The value of σ_R is adjusted to keep the acceptance rate of moves within a desirable tolerance between 25 and 70%. This parameter is controlled to maintain a balance between numerical efficiency and accepting too many moves. If too many moves are accepted, this indicates program inefficiency where random displacements are fluctuations around the minimum. Annealing ends when kT reaches a globally defined minimum value. Then final atomic positions and radial distribution information are written to files.

Previous simulations [9] were conducted with periodic boundary conditions

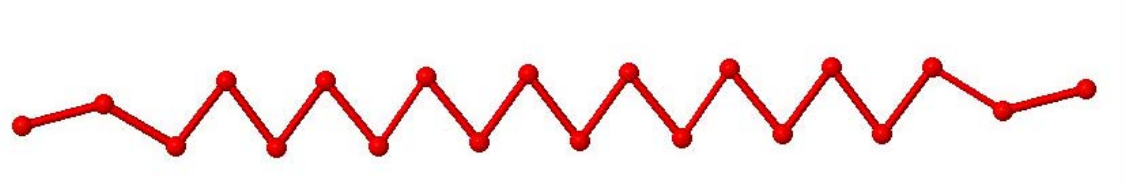


FIG 2 Zigzag equilibrium structure for a 20 atom wire of conductance G = $1G_o$. Image generated using Jmol: an open-source Java viewer for chemical structures in 3D. http://www.jmol.org/

along the wire as chiral atomic structures were expected from prior studies [1,2], but are not known a priori. These conditions were defined according to the helicity of expected high-order chiral structures in an attempt to obviate the wire ends. The periodic boundary conditions caused frustration in the wires because of incommensurability between wire length and the period of chirality. Furthermore, periodic boundary conditions are an unphysical condition of the metal contacts. The use of open boundary conditions allows the system to avoid frustration and better represents the physical reality.

A reheating algorithm was implemented in several simulations to improve the annealing. The wire temperature was raised to increase thermal mobility and move out of minimum energy configurations with defects or changes in chirality within the length of the wire. While defects are physical, the simulation attempts to find the idealized equilibrium structure.

3.2 Equilibrium structures in the canonical ensemble

Simulations in the canonical ensemble for a conductance of G = $1G_o$ show a zigzag equilibrium structure (Figure 2). The plane of the wire was found to be arbitrary. Slight rotations along the wire length represent a finite torsional stiffness and small thermal fluctuations about the planar structure.

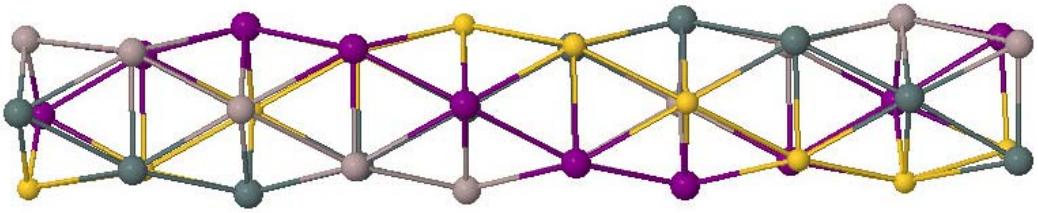


FIG 3 Equilibrium structure for a 40 atom wire of conductance G = $3G_o$. Structure composed of four atomic stands wrapped in helical hollow shell.

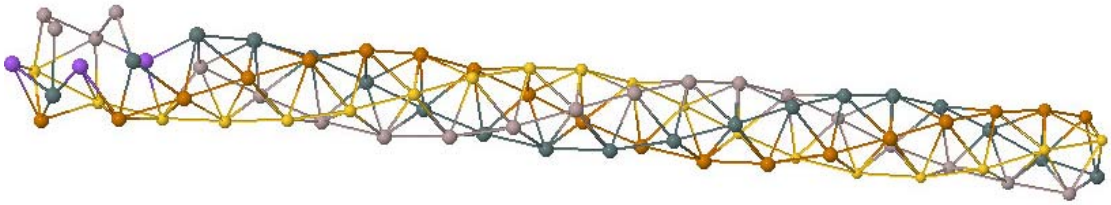


FIG 4 Atomic structure with defect for an 80 atom wire of conductance G = $3G_o$. Structure predominantly equilibrated except for wire end with a defect trapped by the annealing process, suggesting a higher order structure.

Simulations for a wire of conductance G = $3G_o$ result in a helical hollow shell with four atomic strands as the equilibrium structure (Figure 3). From expected structures, simulations were run with a number of atoms an integer multiple of four. Simulations revealed that, dependent upon initial atomic placement, defects such as an odd number of atoms at a wire end are difficult to remove. Open boundary conditions remove frustrations but tend to trap defects at the ends of the wire by the annealing process, resulting in deformed or higher order structures at the corresponding wire end (Figure 4).

4 Simulated annealing in the grand canonical ensemble

The difficulty within the canonical ensemble of annealing out defects at the ends of the wires demonstrates that it does not entirely represent the physical situation being modeled. Experimentally produced wires are suspended between two electrodes, which

can function as atomic reservoirs. Furthermore, TEM images of gold nanowires show that the number of atoms within the wire does not remain constant.

Implementation of the grand canonical ensemble allows for atomic interchange between the wire and the supporting contacts. In this regime, the chemical potential of the system, μ, is an external parameter, along with V and T, while N is allowed to vary. Care must be taken in selection of μ in order to reach an equilibrium state without depleting or flooding the wire with atoms. A basic implementation of the grand canonical ensemble in a Monte-Carlo simulation is detailed by Frenkel [10].

From inclusion of the grand canonical ensemble, the probability to accept a move is then given by the Gibbs factor:

$$p(E) = \exp\left[-\left(\Delta E - \mu \Delta N\right)/kT\right] \tag{11}$$

4.1 Implementation of grand canonical move types

Utilization of the grand canonical ensemble requires two new move types in addition to the canonical Monte-Carlo move: the removal or addition of an atom to the wire. These moves were attempted with a fixed probability. Grand canonical move types only occur after a designated temperature is reached in the annealing process. Removal type moves are attempted with a probability dependent on position:

$$p(z) = \begin{cases} C\exp\left(-|z|/R_S\right) & z \le 4R_S \\ C\exp\left(-|z-L|/R_S\right) & z \ge L - 4R_S \\ 0 & otherwise \end{cases} \tag{12}$$

where C is a normalization constant.

If a removal type move is rejected, then a normal Monte-Carlo move is still performed. If a removal type move is tried, then the program attempts to add an atom on

the same side of the wire at a uniformly distributed random radial position and at a z position defined as follows:

$$z = \begin{cases} -R_S \ln(1 - Z_o) \\ L - R_S \ln(1 - Z_o) \end{cases} \tag{13}$$

where Z_o is a random number between 0 and 1.

Without a priori knowledge of the equilibrium structure, the value of μ is not known prior to simulation. Initial simulations calculated the mean and standard deviation of the chemical potential for all atoms at a distance greater than $4R_S$ away from the wire ends to avoid deviations from defects at the wire ends. These values were calculated via:

$$\mu \approx E(N) - E(N-1); \quad \mu_i = \sum_{j \neq i} E_{\text{int}}(R_{ij}) + U(\rho_i) \tag{14}$$

$$\overline{\mu} = \frac{\sum_i \mu_i}{N_t}; \quad \overline{\nu} = \frac{\sum_i \mu_i^2}{N_t}; \quad \sigma_\mu = \sqrt{\overline{\nu} - \overline{\mu}^2} \tag{15}$$

This calculation of the chemical potential underestimates the chemical potential, as the chemical potential is defined according to a system in equilibrium. In this calculation, $E(N)$ is close to equilibrium, but $E(N-1)$ is not, as the energy is calculated without allowing the system to relax.

4.2 Equilibrated structures

Simulations showed that the calculated value for μ underestimated the actual value, leading to an initial removal of ~20% of the atoms in the wire. To compensate for this deficit, μ is adjusted as follows once the temperature for grand canonical move types is reached:

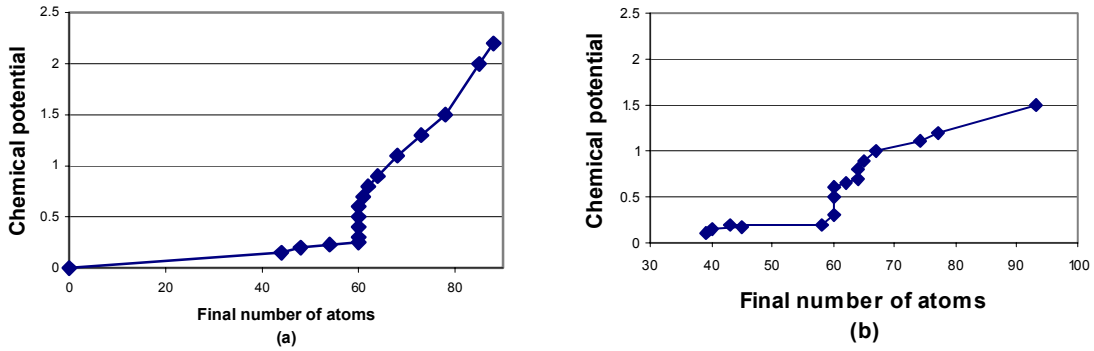


Fig. 5 Final number of atoms in $3G_o$ wire with constant chemical potential. A) Initial number of atoms: 60. b) Ion addition radial range changed to $[0,R+2R_S]$. Initial number of atoms: 60.

$$\mu = \mu_o - \sigma_\mu \frac{kT}{kT_o}(N - N_o) \tag{15}$$

N_o refers to the initial number of atoms, and μ_o is the previous value of the chemical potential. The adjustment is made several times for every value of kT. With this adjustment, the chemical potential then oscillated between favoring removals and additions without reaching an equilibrium number.

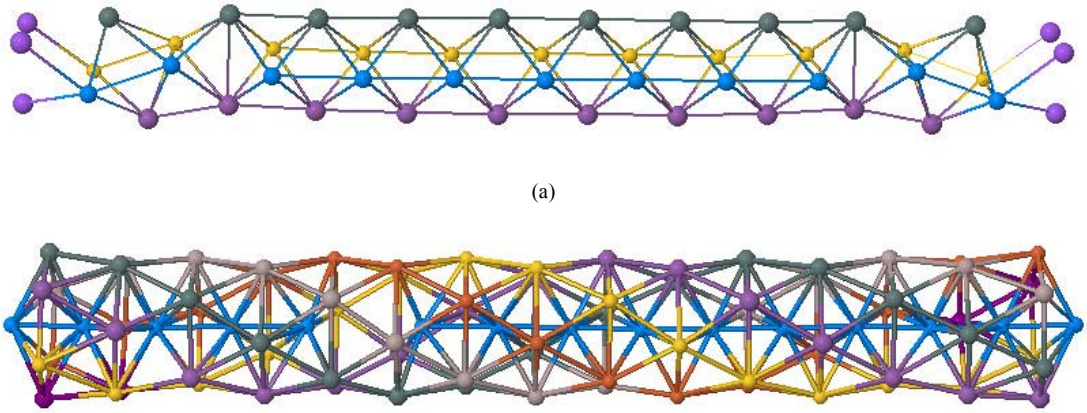
Simulations run with a constant chemical potential as a global parameter revealed surprising results. Figure 5a shows a graph of the chemical potential versus the final number of atoms in the wire. This shows that the aforementioned scheme for grand canonical moves favors atom removals for lower values of the chemical potential, and atom additions for higher values. The sharp rise around N = 60 represents a region where no grand canonical Monte-Carlo moves were accepted, meaning the model acted as within the canonical ensemble. Furthermore, the shape of the curve implies a systematic disposition to atom removal. This disposition stems from a combination of the following

factors: kinetic effects, the method for inserting a new atom, and a non self-consistent confining potential.

The atom addition method inserts a new atom into an effectively frozen configuration, not allowing the system to relax before computing the Gibbs factor. A physical consequence can be paired to this behavior. Ion addition in a physical system would be a gradual process, whereas the simulated annealing move attempts to insert an atom within a random place in configuration space of a pre-existing structure, without any a priori reason to attempt to place the atom within that structure. To adjust this parameter, the radial placement of a new atom was extended to $R_{wire} + 2R_S$, to simulate the physical process of an atom moving along the surface of the wire. The results are shown in Figure 5b. This change decreases the height of the rise, but does not completely resolve the issue.

The present implementation of the grand canonical ensemble does not utilize a self-consistent confinement potential. When an atom is removed or added, the electron gas is considered to remain constant. In reality, an addition or decrease in the number of atoms in the wire would change the strength of the confinement potential.

Simulations with an adequate selection of constant chemical potential generated equilibrium structures for wires of conductance $G = 3G_o$ without defects at the wire ends, representing the same equilibrium structure found shown in Figure 3 for the canonical ensemble.



(a)

(b)

FIG 6 Equilibrium structures for wires of conductance G = $3G_o$ annealed in the grand canonical ensemble with constant chemical potential. Simulations were begun with 60 atoms. (a) Underfilled wire of 48 atoms with four atomic strands. (b) Overfilled wire of 93 atoms. Has five stands of atoms in a helical shell, with a line of atoms through the wire axis.

Simulations with constant chemical potentials also resulted in the generation of equilibrium structures of underfilled and overfilled wires of conductance G = $3G_o$ (Figure 6). Depletion of enough atoms sufficiently decreases the interaction energy that favors chiral structures, resulting in a stacked, four-atom structure (Fig. 6a). Addition of enough atoms causes the structure to favor a chiral structure of 5 atomic chains with a line of atoms through the center of the wire (Fig. 6b).

5 Discussion

The use of canonical and grand canonical Monte-Carlo simulated annealing routines reveals several equilibrium structures for metallic nanowires. Use of open boundary conditions not only provides a more physical model by removing frustration, but also shows a preference for chirality in annealed structures. Use of energetics

approximated from the nanoscale free-electron model implies the stability of these structures.

Implementation of the grand canonical ensemble into the model simulates atomic interchange between the wire ends and the contact electrodes and allows defects in the wire ends to anneal out via interchange. Simulations with a constant chemical potential throughout the annealing reveal a strong dependence on μ for atom additions or removals and needs to be further investigated.

References

[1] O. Gülseren, F. Ercolessi, E. Tosatti, Phys. Rev. Lett. **80**, 3775 (1998)
[2] Y. Kondo, K. Takayanagi, Sci. **289**, 606 (2000)
[3] C.A. Stafford, D. Baeriswyl, J. Bürki, Phys. Rev. Lett. **79**, 2863 (1997)
[4] J. Bürki, C.A. Stafford, Appl. Phys. A **81**, 1519 (2005)
[5] N.W. Ashcroft, N.D. Mermin. *Solid State Physics*. (1976)
[6] *Numerical Recipes in C*, 10.9, pp.444-455
[7] N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, E. Teller, J. Chem. Phys. **21**, 1087 (1953)
[8] *Numerical Recipes in C*, 7.2, pp. 289-290
[9] D. Conner, Master Thesis (2006), N. Rioradan, Independent studies with C.A. Stafford (2007)
[10] D. Frenkel. *Introduction to Monte Carlo Methods*.

Appendix: Program Code

```
/* Simulated Annealing Program for modeling the atomic structure of
metallic nanowires
 * Fall 2007, Spring 2008
 * Written by Corey Flack
 * To compile: cc <filename> -o <executable name> -lm
 * To run: ./<executable name>
 * Grand Canonical Version 4 - Constant Chemcial Potential
 */

#include <stdio.h>
#include <math.h>
#include <time.h>
#include <stdlib.h>

#define Efermi_Na 3.24         // Fermi energy of Na (in eV)
#define Efermi_Au 5.53         // Fermi energy of Au (in eV)
#define kT_start 0.1           // Starting temperature (in units of
Efermi)
#define kT_end 0.000001        // Ending temperature for annealing
#define kT_rate 0.9            // Percentage adjustment for decreasing
annealing temperature
#define N_INIT 1000            // Number of loops before beginning
simulated annealing
#define Nloop 3500             // Number of loops before attempting to
decrease the temperature
#define TEST 100               // Test number of bins for analyzing
radial distribution
#define POT_CONST 0.31         // Constant for the confining potential,
presently defined for Na
#define CUTOFF 6.0             // Atoms further than CUTOFF*Screening
Length are ignored for Nearest Neighbor calculation
#define RS 1.91916             // Atomic radius in units of 1/kF.
Defined as (3*M_PI/4)^(1/3)
#define SIG_Ro 0.5             // RMS value for radial displacement
#define kF_Na 0.92             // kF for Sodium in units of inverse
Angstroms
#define kF_Au 1.21             // kF for Gold in units of inverse
Angstroms
#define Ao 0.529177*kF_Na      // Bohr radius in units of 1/kF
#define SL 0.815*sqrt(RS/Ao)   // Thomas-Fermi screening length
//#define EPSO 0.805*sqrt(Efermi)// Permeativity in natural units
#define EPSO 0.0194
#define UPPER_L 0.70           // Upper limit for acceptance rate
#define LOWER_L 0.25           // Lower limit for acceptance rate
#define dSIG 0.5               // Change of sigma for modifying
acceptance rates
#define CHECK 500              // Number of Monte-Carlo loops before
calculating total energy
#define CP_CALC 750           // Number of Monte-Carlo loops before
calculating chemical potential
#define CP_INIT 0.03           // Temperature to initiate calculation
of chemical potential
#define GC_INIT 0.02           // Temperature to initiate Grand
Canonical move type
```

```
#define GC_MOVE 0.85           // Tolerance value for random move
selection of simulated annealing move type
#define CONT 4.                 // Integer number of sig_r that
simulates contact with the atomic resevoirs
#define ADDTL 40               // Additional number of atoms to include
in memory allocation
#define NORM 0.530783          // Normalization constant for GC
probability, calculated so that P(z) integrated from 0 to CONT*RS = 1
#define ALPHA 0.
#define TOLERANCE 0.3
#define FACTOR 50
#define CHECK_VAL 100
#define CHEMICAL 0.20

// Constants for Hard-Core Phenomenological Repulsion, presently set
for Sodium (Na)
#define HCR_m 10               // Power factor m
#define HCR_eps 0.05/Efermi_Na // Proportionality constant epsilon, in
units of Ef
#define HCR_aAng 4.23          // Radiua constant a, in Angstroms
#define HCR_a HCR_aAng*kF_Na   // Radius constant a, in units of 1/kF

// Constants for random number generator function
#define IA 16807
#define IM 2147483647
#define AM (1.0/IM)
#define IQ 127773
#define IR 2836
#define NTAB 32
#define NDIV (1+(IM-1)/NTAB)
#define EPS 1.2e-7
#define RNMX (1.0-EPS)

//=================================================================
==============================
/* Define the stuctures to be used for storing the wire and ion
dimensions, locations, and dimensions
   Wire structure will contain wire specific information: radius,
length, number of atoms, conductions channels, and number of slices
   Ion structure will contain inform such as X, Y, and Z positions, as
well as the potential energy of the ion
   ManyIon structure is the structure for the set of all the ions
*/
//=================================================================
==============================

typedef struct ion_struct {
  double X, Y, Z;
  double Epot, Ei;
  int N_acpt;                      // Number of accepted Monte-Carlo
moves for particular atom
} ion;

typedef struct ion_orders {
  int Zorder;                          // Ion order along length of
wire; array position 0 corresponds to first ion in wire, array value to
ion #
```

```
  int invZorder;                          // Inverse ion order; array
position 0 corresponds to ion #0, array value to order #
} order;

typedef struct wire_struct {
  float L, R;                             // Wire length, radius
  double CPm, CPstd, CP;                      // Chemical potential mean
and standard deviation for wire
  int NAtoms, NC;                         // Number of atoms and
conduction channels
  ion *p;                                 // Ion structure pointer
  order *o;                               // Order structure pointer
  int GCtry, GCacpt, GCtry_tot, GCacpt_tot;                     //
Number of Grand-Canonical type (ion removal/addition) moves tried and
accpeted
} ManyIons;

//=================================================================
==============================
// Function Prototypes
void RandomPositions(ManyIons *mi);   // Function that generates random
initial positions of atoms
float PotentialEnergy(float rho, float z, float Length, float Radius);
// Function to calculate Potential energy
void Allocate(ManyIons *mi, int NAtoms);                   // Function
to allocate memory for ion structures
void freeIons(ManyIons *mi);                              // Function
to free memory after running of program
float Repulsion(float R);                                    //
Function to calculate the repulsive energy
void WritePositions(ManyIons *mi, char *filename);           //
Function to write ion positions
double AnalyzeRadialDistribution(ManyIons *mi, int Bins, char
*filename);    // Function to create histogram of radial distribution
void AnalyzeNearestNeighbors(ManyIons *mi, int Bins, char *filename);
// Function to create histogram of nearest neighbor distances
void MC_Move(ManyIons *mi, int j, float sig_r, float kT, int *stat,
long *idum);                                  // Monte-Carlo Move
sequence
void Sort(ManyIons *mi);                                    // Function
that sorts the atomic arrays
double TotalEnergy(ManyIons *mi, int pt);                // Function to
calculate the total ionic energy for the distribution
double InteractionEnergy(ManyIons *mi, int j);             // Function
to calculate the interaction energy of only one ion
float ran1(long *idum);                                     // Function
to generate a random number from 0 to 1; from Numerical Recipes
float Normal(long *idum);                                   // Function
from Numerical Recipes to generate a normally distributed random
deviant with zero mean and standard deviation sig_r
void ChemicalPotential(ManyIons *mi, float kT, char *filename);
// Function to calculate the chemical potential for all ions
void InitiateIonAcceptanceRate(ManyIons *mi);
// Function to initialize the number of accepted Monte-Carlo moves per
atom for Chemical Potential cacalculations
```

```
void ShiftIons(ManyIons *mi, int j);              // Function to shift
ions after a change in ion number: j represents ion removed, if greater
than NAtoms, represents added ion
void UnwrapWire(ManyIons *mi, double AvR, char *filename);       //
Function to unwrap the final wire to view structure; R value is average
radius of wire
int GrandCanonicalMove(ManyIons *mi, int j, float sig_r, float kT, int
*stat, long *idum); // Function called by MC_Move to add and remove
ions from the wire
void AdjustChemicalPotential(ManyIons *mi, float kT, int N, char
*filename);     // Routine to adjust the chemical potential based off
of the number of atoms in the wire
//=====================================================================
==============================
// Main function
int main(int argc, char **argv)
{
  int N, i, j, k, T;
  ManyIons ions;
  int Z_order;
  float kT, sig_r, ratio, ratioGC;
  int N_acpt, N_try, Ntot_try, Ntot_acpt, stat, Num;
  long idum;
  double TE;
  FILE *fp;
  int j2;
  ion *p;
  order *o;
  double AvR;

  printf( "Wire radius in 1/kF = ");                      // Get wire
radius from user input
  scanf("%f", &ions.R);

  ions.NC = (0.25 * ions.R * ions.R);               // Calculate
number of conduction channels from user input
  printf( "The wire has %d channels.\n", ions.NC);
  printf("Eps_o = %f\n", EPSO);
  printf( "Enter the number of atoms to include in the simulation: ");
//User input to determine number of atoms and wire length
  scanf("%d", &N);
  ions.NAtoms = N;

  ions.L = 3.0 * M_PI * ions.NAtoms / (ions.R * ions.R);

  // Allocate memory for the ions
  Allocate(&ions, ions.NAtoms);

  printf( "Wire length is: %f 1/kF.\n", ions.L);

  // Set random positios
  RandomPositions(&ions);

  // Analyze Initial Distribution
  //  AnalyzeRadialDistribution(&ions, TEST,
"/usr3/cjflack/Thesis_Work/SimAnnealing/Test/InitialHistogram.dat");
```

```
  AnalyzeRadialDistribution(&ions, TEST,
"./SimTest1/InitialHistogram.dat");

  // Sort initial atomic positions
  Sort(&ions);

  //Write Initial Positions
  WritePositions(&ions, "./SimTest1/InitialPositions.dat");

  // Simulated annealing loop
  kT = kT_start;
  N_acpt = 0;
  N_try = 0;
  Ntot_try = 0;
  Ntot_acpt = 0;
  idum = 0-rand();
  T = 0;
  Num = 0;
  ions.GCtry = 0;
  ions.GCacpt = 0;

  // Initial Monte-Carlo moves before Simulated annealing
  sig_r = SIG_Ro;
  for (j=1; j <= N_INIT; j++) {
    for (i=0; i < ions.NAtoms; i++) {
      MC_Move(&ions, i, sig_r, kT, &stat, &idum);
    }
  }

  // Calculate "initial" energy of the system
  TotalEnergy(&ions, 1);

  stat = 0;
  N_acpt = 0;
  N_try = 0;
  sig_r = SIG_Ro * sqrt(kT/ kT_start);

  while (kT > kT_end) {
    if (ions.NAtoms == 0) {
      printf("All atoms removed! Chemical potential value
underestimated\n");
      break;
    }
    if (ions.NAtoms >= N + ADDTL) {
      printf("Too many atoms added! Chemical potential value
overestimated.\n");
    }
    for (j = 1; j <= Nloop; j++) {
      for (i = 0; i < ions.NAtoms; i++) {
      MC_Move(&ions, i, sig_r, kT, &stat, &idum);

      // i refers to ion order #; define pointer to actual ion
      o=ions.o+i;
      j2 = o->Zorder;          // Atom number
      p = ions.p+j2;            // Pointer to ion

      N_try = N_try + 1;
```

```
      Num = Num + 1;
      N_acpt = N_acpt + stat;
      p->N_acpt = p->N_acpt + stat;

      }
      if (fmod(j, CHECK) == 0) {
      TE = TotalEnergy(&ions, 0);
      fp = fopen("./SimTest1/TotalEnergy.dat", "a+");
      fprintf(fp, "%d\t%f\t%f\t%d\n", Num, TE, kT, ions.NAtoms);
      fclose(fp);
      }
      /*
      if (fmod(j, CP_CALC) == 0) {
      if (kT < CP_INIT && kT > GC_INIT) {
      ChemicalPotential(&ions, kT, "./SimTest1/ChemicalPotential.dat");
      }
      if (kT < GC_INIT) {
      AdjustChemicalPotential(&ions, kT, N,
"./SimTest1/ChemicalPotentialMean.dat");
      }
      }
      */
    }

    ratio = (float)N_acpt/((float)N_try);
    ratioGC = (float)ions.GCacpt/((float)ions.GCtry);
    if (ratio > UPPER_L) {
      sig_r = sig_r / dSIG;
      printf("Acceptance rate: %f  Sigma: %f\n", ratio, sig_r);
    }
    else if (ratio < LOWER_L) {
      sig_r = sig_r * dSIG;
      printf("Acceptance rate: %f  Sigma: %f\n", ratio, sig_r);
    }
    else if (ratio >= LOWER_L && ratio <= UPPER_L) {

printf("##============================================================#
#\n");
      printf("T = %f , sig_r = %f\n", kT, sig_r);
      printf("=========================\n");
      printf("Number of tried moves: %d   Number of accepted moves:
%d\n", N_try, N_acpt);
      printf("Acceptance rate: %f\n", ratio);
      if (kT < GC_INIT) {
      printf("Number of tried removals/additions: %d\t Number of
accepted: %d\n", ions.GCtry, ions.GCacpt);
      printf("Acceptance rate of GC moves: %f\n", ratioGC);
      //    printf("Chemical potential: %f\n", ions.CPm);
      }
      printf("Number of atoms: %d\n", ions.NAtoms);
      WritePositions(&ions, "./SimTest1/CurrentPositions.dat");
      TotalEnergy(&ions, 1);
      AnalyzeRadialDistribution(&ions, TEST,
"./SimTest1/CurrentHistogram.dat");

      kT = kT * kT_rate;
      sig_r = SIG_Ro * sqrt(kT/ kT_start);
```

```
      T = T+1;
      printf("T = %d\n", T);
      //      if (T == 50) {
      // printf("Reheating test\n");
      //  kT = kT_start * pow(kT_rate, 10);
      //}
      //      if (T == 100) {
      //  printf("Reheating test #2\n");
      //    kT = kT_start * pow(kT_rate, 75);
      //}
      Ntot_try = Ntot_try + N_try;
      Ntot_acpt = Ntot_acpt + N_acpt;
      ions.GCtry_tot = ions.GCtry_tot + ions.GCtry;
      ions.GCacpt_tot = ions.GCacpt_tot + ions.GCacpt;
      ions.GCtry = 0;
      ions.GCacpt = 0;
      N_acpt = 0;
      N_try = 0;
    }
  }

  printf("Total tries: %d  Total number of accepted moves: %d Overall
Acceptance rate: %f\n", Ntot_try, Ntot_acpt,
(float)Ntot_acpt/(float)Ntot_try);
  printf("Number of tried removals/additions: %d\t Number of accepted:
%d\n", ions.GCtry, ions.GCacpt);
  printf("Acceptance rate of GC moves: %f\n", (float)ions.GCacpt_tot /
((float)ions.GCtry_tot));
  AvR =  AnalyzeRadialDistribution(&ions, TEST,
"./SimTest1/FinalHistogram.dat");
  WritePositions(&ions, "./SimTest1/FinalPositions.dat");
  UnwrapWire(&ions, AvR, "./SimTest1/UnwrappedWire.dat");

  freeIons(&ions);
  return 0;
}

//=====================================================================
==============================
// Function to generate random initial positions for N_Atoms
void RandomPositions(ManyIons *mi)
{
  double Theta, Rad, Z, X, Y;                  // Temporaray theta and
radial coordinates; X, Y, and Z coordinates
  int j;
  ion *p;
  srand((unsigned)time(NULL));

  for (j=0; j < mi->NAtoms; j++) {
    p = mi->p+j;

    Theta = 2.0 * M_PI * 1*(rand()/(RAND_MAX+1.));        // Temporary
Theta coordinate
    Rad = mi->R * sqrt(1*rand()/(RAND_MAX+1.));           // Temporary
R coordinate
    Z = 1*(rand()/(RAND_MAX+1.));
```

```
    p->X = Rad*cos(Theta);                                  // X
component
    p->Y = Rad*sin(Theta);                                  // Y
component
    p->Z = mi->L * Z;                                       // Z
component

    // Calculate potential energy of each ion
    p->Epot = PotentialEnergy(Rad, p->Z, mi->L, mi->R);
    }

  return;
}

//=================================================================
==============================
// Potential energy function
float PotentialEnergy(float rho, float z, float Length, float Radius)
{
  //====================================//
  // Rho: Radial coordinate of ion      //
  // z: z coordinate of ion             //
  // Length: length of wire             //
  // Radius: Radius of wire             //
  // Z confining potential begins at    //
  // Z = Rs and Z = L-Rs to keep ions   //
  // within the wire                    //
  //====================================//
  float U_Rho, U_z, U_tot;     //Results of calculation, for rho and z
direction respectively

// Z-portion
  if (z <RS) {
    U_z = POT_CONST *(z-RS)*(z-RS) /RS;
  } else if (z > (Length-RS)) {
    U_z = POT_CONST  * (z-(Length-RS))*(z-(Length-RS)) /RS;
  } else {
    U_z = 0.0;
  }

  // Rho portion
  U_Rho = POT_CONST/Radius * (rho*rho - Radius*Radius);

  // Total confining potential energy as the sum of the two
  U_tot = U_Rho + U_z;

  return U_tot;
}
//=================================================================
==============================
float Repulsion(float R)
{
  float Rep, Col, Rtot;
  Rep = HCR_eps * pow((HCR_a / R), HCR_m);
  Col = exp(-R/SL)/(4.0*M_PI*EPSO*R);
  Rtot = Col + Rep;
  return Rtot;
```

```
}

//=================================================================
==============================
void WritePositions(ManyIons *mi, char *filename)
{
  FILE *fp;
  ion *p;
  int i, j;
  order *oi;

  fp = fopen(filename,"a+");
  fprintf(fp, "## Ion positions. \n");
  fprintf(fp, "## Wire radius in kf: %f \n", mi->R);
  fprintf(fp, "## Wire length in kF: %f \n", mi->L);
  fprintf(fp, "## Number of atoms: %d \n", mi->NAtoms);
  fprintf(fp, "## Number of conduction channels: %d \n", mi->NC);
  fprintf(fp, "\n");
  fprintf(fp, "##
============================================================\n");
  fprintf(fp, "## Z              X            Y\n");

  //Write ion information
  /* for (j=0; j < mi->NAtoms; j++) {
    p = mi->p+j;
    fprintf(fp, " %f        %f      %f\n", p->Z, p->X, p->Y);
  }
  */

  // Write ion information in atomic order
  for (j=0; j< mi->NAtoms; j++) {
    oi = mi->o+j;
    i = oi->Zorder;
    p = mi->p+i;
    fprintf(fp, " %f        %f      %f\n", p->Z, p->X, p->Y);
  }

  fprintf(fp, "\n");
  fclose(fp);
  return;
}

//=================================================================
==============================
void freeIons(ManyIons *mi)
{
  free(mi->p);
  free(mi->o);
  mi->NAtoms = 0;
  return;
}

//=================================================================
==============================
double AnalyzeRadialDistribution(ManyIons *mi, int Bins, char
*filename)
{
```

```
  int Nbins;
  double dr, r, r_next, rho, AvR;
  FILE *fp;
  ion *pi;
  int i, j, B[Bins+1];

  for (i=0; i <= Bins; i++) {
    B[i] = 0;
  }

  dr = 1.5*mi->R / Bins;
  AvR = 0;

  // Distrubute radii into bins
  for (j=0; j < mi->NAtoms; j++) {
    pi = mi->p+j;
    rho = sqrt(pi->X * pi->X + pi->Y * pi->Y);              //
Calculate radii
    r = 0.;
    r_next = r + dr;
    AvR = AvR + rho;                        // Sum radii to calculate
average radius
    for (i=0; i <= Bins; i++) {             // Loop through bins to
place radii
      if (rho >= r && rho < r_next) {
         B[i] = B[i]+1;
         }
      r = r + dr;
      r_next = r_next + dr;
    }
  }

  AvR = AvR / mi->NAtoms;                  // Computer average radius

  fp = fopen(filename, "a+");
  fprintf(fp, "## Radial distribution of ionic positions.\n");
  fprintf(fp, "## Wire radius: R = %f kF \n", mi->R);
  fprintf(fp, "## Wire length: L = %f kF \n", mi->L);
  fprintf(fp, "## Number of bins: %d \n", Bins);
  fprintf(fp, "## Number of atoms: %d \n", mi->NAtoms);
  fprintf(fp, "## Average radius: %f \n", AvR);
  fprintf(fp, "## Bin width: dr = %f kF \n", dr);
  fprintf(fp, "\n## ======================================== \n");
  fprintf(fp, "##R (kF)       Count \n");

  r = 0.;
  for (i=0; i <= Bins; i++) {
    fprintf(fp, "%f          %d    \n", r, B[i]);
    r = r + dr;
  }
  fprintf(fp, "\n");
  fclose(fp);
  return AvR;
}
//=================================================================
==============================
/*void AnalyzeNearestNeighbors(ManyIons *mi, int Bins, char *filename)
```

```
{
  int Nbins;
  FILE *fp;
  ion *pi;
  ion *pj;
  order *oi;
  order *oj;
  int i, i2, j, j2, k, Nn[Bins+1];
  double dNn, dZ, Rij,
*/
  // Initialize bin values
  /*for (i=0; i <= Bins; i++) {
    B[i] = 0;
  }

  dNn = 1.5 * CUTOFF/Bins;

  for (j = 0; j < mi->NAtoms; j++) {
  oj = mi->o+j; */
  //    j2 = oj->Zorder;        // Atom number
  //    pj = mi->p+j2;          // Pointer to ion
  //    for (i = 0; i < mi->NAtoms; i++) {
  //      oi = mi->o+i;
  //      i2 = oi->Zorder;      // Atom number
  //      pi = mi->p+i2;        // Pointer to ion
  /*      dZ = fabs(pj->Z  - pi->Z);
      if (dZ < CUTOFF) {
      Rij = sqrt((pj->X - pi->X)*(pj->X - pi->X) + (pj->Y - pi->Y)*(pj-
>Y - pi->Y) + dZ*dZ);
      Nn = 0;
      for (k = 0; k <= Bins; k++) {
        if (Rij >= Nn && Rij < (Nn + dNn)) {
          Nn[k]= Nn[k]+1;
          break;
        }
        else Nn = Nn + dNn;
      }
      }
    }
  }

  fp = fopen(filename, "a+");
  fprintf(fp, "## Nearest neighbor distances.\n");
  fprintf(fp, "## Wire radius: R = %f kF \n", mi->R);
  fprintf(fp, "## Wire length: L = %f kF \n", mi->L);
  fprintf(fp, "## Number of bins: %d \n", Bins);
  fprintf(fp, "## Number of atoms: %d n", mi->NAtoms);
  fprintf(fp, "## Bin width: dNn = %f kF \n", dr);
  fprintf(fp, "\n## ======================================== \n");
  fprintf(fp, "##Rij (kF)       Count \n");

  Nn = 0.;
  for (i=0; i <= Bins; i++) {
    fprintf(fp, "%f          %d    \n", Nn, B[i]);
    Nn = Nn + dNn;
  }
  fprintf(fp, "\n");
```

```
  fclose(fp);
  return;
}
  */
//=====================================================================
==============================
void Allocate(ManyIons *mi, int NAtoms)
{
  int N;
  N = NAtoms + ADDTL;
  mi->p = (ion *)malloc(N * sizeof(ion));
  mi->o = (order *)malloc(N * sizeof(order));
  if (mi->p==NULL || mi->o==NULL) {
    fprintf(stderr,"Memory allocation failed.\n");
    abort();
  }
  return;
}
//=====================================================================
==============================
void MC_Move(ManyIons *mi, int j, float sig_r, float kT, int *stat,
long *idum)
{
  double dr, theta, phi, nE, Ei;
  int j2;
  double nX, nY, nZ, nr, dE, nEp, nEi, Bolz, eta;  // New coordinate
values, and evergy values
  double oldE, oldEi, oldEp, oX, oY, oZ;                        // Old
energy values
  ion *p;
  order *o;
  float MT;                                          // Random number used
for move determination
  int move=0;                                        // Type of Move: 0
(default) represents Monte-Carlo move, 1 represents adding or removing
an ion
  double Pz, MZ;
  FILE *fp;
  fp = fopen("./SimTest1/EnergyCheck.dat", "a+");

  // j refers to ion order #; define pointer to actual ion
  o=mi->o+j;
  j2 = o->Zorder;          // Atom number
  p = mi->p+j2;            // Pointer to ion

  // Grand Canonical Move Determination
  if (kT <= GC_INIT) {

    // Calculate removal possibility by exp(-|Z-Zo|/RS)

    if (p->Z <= CONT * RS) {
      Pz = NORM * exp(-fabs(p->Z) / RS);
    } else if (p->Z >= (mi->L - (CONT * RS))) {
      Pz = NORM * exp(-fabs(p->Z - mi->L) / RS);
    }

    MZ = 1*(rand()/(RAND_MAX + 1.));
```

```
    if (MZ < Pz) {                              // Compare move
probability to random number between 0 and 1
      MT = 1*(rand()/(RAND_MAX + 1.));      // Random number used for
move determination
      if (MT > GC_MOVE) {

      // Grand canonical move selected; if atom is not removed, random
move is still done
      move = GrandCanonicalMove(mi, j, sig_r, kT, stat, idum);
      }
    }
  }

  if (move == 0) {

    // Generate random move coordinates
    // Calculate the new dr via a normal distribution:

    dr = sig_r*Normal(idum);
    theta = M_PI * 1*(rand()/(RAND_MAX + 1.));
    phi = 2.0 * M_PI * 1*(rand()/(RAND_MAX + 1.));

    // Calculate old energies
    oldEi = InteractionEnergy(mi, j);
    oldEp = p->Epot;
    oldE = oldEp + oldEi;

    // Store old position data
    oX = p->X;
    oY = p->Y;
    oZ = p->Z;

    // Generate new coordinates
    nX = oX + dr*sin(theta)*cos(phi);
    nY = oY + dr*sin(theta)*sin(phi);
    nZ = oZ + dr*cos(theta);
    nr = sqrt(nX*nX + nY*nY);

    // Save new coordinates in structure for interaction energy
calculation
    p->X = nX;
    p->Y = nY;
    p->Z = nZ;

    nEp = PotentialEnergy(nr, nZ, mi->L, mi->R);
    nEi = InteractionEnergy(mi, j);

    // Calculate the Bolzman factor for move probability
    dE = nEp + nEi - oldE;
    Bolz = exp(-dE/ kT);

    // Random number used for move acceptance
    eta = 1*(rand()/(RAND_MAX + 1.));

    if (eta > Bolz) {
      // Rejected move
```

```
      *stat = 0;
      p->X = oX;
      p->Y = oY;
      p->Z = oZ;
    } else {
      *stat = 1;

      // May need to add conditional statements in case the new
coordinate is outside the bounds of the wire, although the confining
potential should prevent this
      /* Arrays of Zorder need to be adjusted for new positions
       if (nZ > p->Z) {
       do {
       i = invZorder[j];
       if (i == Natoms) {
       exit;
       }        // Ion j is already the right-most atom

       This whole segment seems irrelevant if I just rerun the sort
program. However, the sort program takes up more CPU cycles, thereby I
should find a more optimal method.
       However, for the time being, it should be an adequate test.
      */

      if (dE > CHECK_VAL) {
      fprintf(fp, "MC Move: %f\t\t%e\t\t%f\n", dE, Bolz, eta);
      }

      p->Epot = nEp;
      Sort(mi);          // Re-sort atoms
    }
  }

  fclose(fp);
  return;
}
//=====================================================================
=====================================================================
=========
int GrandCanonicalMove(ManyIons *mi, int j, float sig_r, float kT, int
*stat, long *idum)
{
  // Subroutine to attempt ion removal or ion addition utilizing the
Grand Canonical ensemble
  // *stat returns a value of 0, 1, or 2, depending on the number of
accepted moves
  // (removal and addition of ions are regarded as seperate moves

  ion *pi;                        // Pointer to ion being removed
  ion *pn;                        // Pointer to ion being added
  order *on;                      // Pointer to order structure of new
ion
  order *oi;
  double oldEp, oldEi, oldE, mu;  // Old energy values and chemical
potential (mu)
  double Ep, Ei, newE, dE;
  double Bolz, eta;
```

```
  double Rad, Theta;
  double Z, Zo, Pz, MZ, Zold;
  int i, n, in, move;
  FILE *fp;

  oi = mi->o+j;
  i = oi->Zorder;
  pi = mi->p+i;

  Zold = pi->Z;

  fp = fopen("./SimTest1/EnergyCheck.dat", "a+");

  // Attempt to remove ion j

  // Get old energy values
  oldEp = pi->Epot;
  oldEi = InteractionEnergy(mi, j);
  oldE = oldEp + oldEi;
  newE = 0.;
  dE = newE - oldE;

  // Average chemical potential value
  mu = CHEMICAL;

  // Calculate Bolzman factor in the Grand Canonical ensemble and
random number for move probability
  Bolz = exp(-(dE + mu) / kT);
  eta = 1*(rand()/(RAND_MAX + 1.));



  if (eta > Bolz) {
    // Rejected move, normal MC move will still be performed
    move = 0;
    *stat = 0;

  } else {
    // Ion removed from wire
    move = 1;
    *stat = 1;
    mi->GCacpt = mi->GCacpt + 1;
    if (dE > CHECK_VAL) {
      fprintf(fp, "%f\t\t%f\t\t%e\t\t%f\t-1\n", dE, mu, Bolz, eta);
    }

    // Shift array values to account for removed ion
    ShiftIons(mi, j);
  }

  // Attempt to add an ion
  Zo = 1*(rand()/(RAND_MAX + 1.));           // Generate random number
for use in calculating new Z coordinate
  Z = -RS * log(1 - Zo);                     // Calculate new Z
position

  if (Zold > (mi->L - (CONT * RS))) {
```

```
    Z = mi->L - Z;                              // If test ion is between
L-RS and L, adjust Z coordinate of new ion to be on proper end of wire
    Pz = NORM * exp(-fabs(Z - mi->L)/RS);
  } else {
    Pz = NORM * exp(-fabs(Z)/RS);
  }

  if (Z > mi->L + RS) {                         // Automatically reject
added ions too far out of the wire
    Pz = -1.;
  }

  MZ = 1*(rand()/(RAND_MAX + 1.));              // Random number for move
probability

  if (MZ < Pz) {
    // Attempt to add ion
    // Generate random radial position from uniform distribution
    Theta = 2.0 * M_PI * 1*(rand()/(RAND_MAX+1.));
    Rad = (mi->R + 2*RS) * sqrt(1*rand()/(RAND_MAX+1.));

    // Store new ion information at end of ion array
    n = mi->NAtoms;
    pn = mi->p+n;

    mi->NAtoms = mi->NAtoms + 1;           // Increase N atoms value to
allow for interaction energy calculation

    // Store new ion into structure
    pn->X = Rad * cos(Theta);
    pn->Y = Rad * sin(Theta);
    pn->Z = Z;

    // Sort ions for proper interaction energy calculation and
determination of order #
    Sort(mi);
    on = mi->o+n;
    in = on->invZorder;

    // Calculate potential and interaction energies
    Ep = PotentialEnergy(Rad, Z, mi->L, mi->R);
    Ei = InteractionEnergy(mi, in);
    newE = Ep + Ei;

    // Calculate Bolzman factor in Grand-Canonical Ensemble for move
determination
    Bolz = exp(-(newE - mu) / kT);

    eta = 1*(rand()/(RAND_MAX+1.));


    if (eta > Bolz) {

      // Move rejected; reduce NAtoms and resort atoms
      mi->NAtoms = mi->NAtoms - 1;
      Sort(mi);
```

```
    } else {
      // Move accepted
      *stat = *stat + 1;
      pn->Epot = Ep;

      mi->GCacpt = mi->GCacpt + 1;
    }

  }

  fclose(fp);
  mi->GCtry = mi->GCtry + 2;
  return move;
}
//=====================================================================
=====================================================================
=========
void Sort(ManyIons *mi)
{
  ion *p;
  order *op, *io;
  int i, j, k, N;             // Integers used for looping
  float Zo, Zn, Z, Zc;           // Z values to use for sorting; set
initial check value as wire length
  FILE *fp;                   // File pointer to print order into file
for checking purposes

  Zo = 5 * mi->L;
  N = mi->NAtoms;
  Zn = Zo;

  // Find leftmost atom
  for (k=0; k < N; k++) {
    op = mi->o+k;
    if (k == 0) {
      for (j=0; j < N; j++) {
      p = mi->p+j;
      Zc = p->Z;
      if (Zc < Zo) {
        Zo = Zc;
        i = j;
      }
      }
      op->Zorder = i;
      //f Ion number: %d\n", Zo, i);
      // Sort other ions
    } else {
      Zn = 100 * mi->L;
      for (j=0; j < N; j++) {
      p = mi->p+j;
      Zc = p->Z;
      if (Zc > Zo) {          // If current Z is greater than the
previous Z, and less than Zn
        if (Zc <= Zn) {
          if (Zc == Zn) {
            //          printf("Ions in same z position! Ion %d\n", j);
            Zn = Zc - 0.00001;
```

```
            p->Z = p->Z - 0.00001;
            //printf("Zc = %f  Zn = %f  j = %d    i = %d\n", Zc, Zn, j,
i);
            i = j;
          } else {
            Zn = Zc;
            i = j;
            //   printf("Ions in same position!\n");

            //      } else if (Zc < Zn) {
            //Zn = Zc;
            //i = j;
          }
        }
      }
      }
      op->Zorder = i;
      Zo = Zn;
    }

  }

  for (j=0; j < mi->NAtoms; j++) {
    io = mi->o+j;
    k = io->Zorder;
    op = mi->o+k;
    op->invZorder = j;
  }

  return;
}
//=====================================================================
================================================
double TotalEnergy(ManyIons *mi, int pt)
{
  //=======================================//
  // Variable definition                   //
  // Etot: Total energy of system          //
  // P: Potential Energy                   //
  // I: Interaction Energy                 //
  //=======================================//
  double Etot, U, I;
  double Eptot=0, Ei, Eitot=0;
  float Ri;
  int i, j;
  ion *p;
  float lowR;
  lowR = 1.;
  FILE *fp;

  // Calculate Total Potential Energy for atoms
  for (j=0; j < mi->NAtoms; j++) {
    p = mi->p+j;
    Eptot = Eptot + p->Epot;
  }

  // Calculate Total Ion Interaction Energy
```

```
  for (j=0; j < mi->NAtoms; j++) {
    Ei = InteractionEnergy(mi, j);
    Eitot = Eitot + Ei;
  }


  Etot = Eptot + 0.5*Eitot;               // Total energy; 1/2 to
prevent double counting of interaction energy

  // Print statements if output desired
  if (pt == 1) {
    printf("Potential energy calculated: %f Ef\n", Eptot);
    printf("Interation energy calculated: %g Ef\n", Eitot);
    printf("Energies:\nPotential: %f   Interaction: %g   Total: %f\n",
Eptot, Eitot, Etot);
  }

  return Etot;
}
//=========================================================================
==========================================
double InteractionEnergy(ManyIons *mi, int j)
    {
      // *mi refers to the structure containing all data for the wire
      // j refers to the number ion being picked, in reference to it's
order along the z-axis

      ion *p;        // Pointer to refer to reference ion
      ion *pi;       // Pointer to refer to interaction ion
      order *or;      // Pointer to refer to the order structure for
the reference ion
      order *oi;     // Pointer to refer to the order structure for the
interaction ion

      int i, i2;         // Integers used for atom number reference
      int k, j2;             // Integer used for looping
      int N, Nn;             // Number of total atoms in wire
      double maxD, Eint, dZ, dY, dX, Rij;       // Maximum distance for
calculation; energy; Z separation between ions, distance between ions

      maxD = CUTOFF;    // Max distance between ions set as the
globally defined cutoff distance

      N = mi->NAtoms;   // Define the number of atoms
      Nn = 0;           // Initialize nearest neighbors counter
      Eint = 0.;        // Initial intertnal energy value

      // Using the number j, find the jth element of the order array to
find the atom number for the reference ion
      or = mi->o+j;
      i = or->Zorder;    // I refers to atom number - define pointer
for ion coordinates
      p = mi->p+i;

      j2 = j+1;          // Advance to first ion

      // Calculate energies for ions to the right
```

```
      while (j2 < N) {

      // Next ion pointers
      oi = mi->o+j2;
      k = oi->Zorder;
      pi = mi->p+k;

      dZ = fabs(p->Z - pi->Z);
      if (dZ < CUTOFF) {
        dX = fabs(p->X - pi->X);
        dY = fabs(p->Y - pi->Y);
        Rij = sqrt(dX * dX + dY * dY + dZ * dZ);
        if (Rij <= CUTOFF) {
          Eint = Eint + Repulsion(Rij);
          Nn = Nn + 1;
        }
      }
      if (dZ >= CUTOFF) {
        break;
      }
      j2 = j2+1;       // Advance to next ion; calculate last to ensure
loop breaks properly
      }

      j2 = j-1;     // Reset j2 counter value

      // Calculate energies for ions to the left
      while (j2 >= 0) {

      // Next ion pointers
      oi = mi->o+j2;
      k = oi->Zorder;
      pi = mi->p+k;

      dZ = fabs(p->Z - pi->Z);
      if (dZ < CUTOFF) {
        dX = fabs(p->X - pi->X);
        dY = fabs(p->Y - pi->Y);
        Rij = sqrt(dX * dX + dY * dY + dZ * dZ);
        if (Rij <= CUTOFF) {
          Eint = Eint + Repulsion(Rij);
          Nn = Nn + 1;
        }
      }
      if (dZ >= CUTOFF) {
        break;
      }
      j2 = j2-1;      // Advance to next ion
      }

      return Eint;
    }


//=====================================================================
=========================================
// ran1 function from Numerical-Recipes
```

```
/* " 'Minimal' random number generator of Park and Miller with Bays-
Durham shuffle and added safeguards. Returns a uniform random deviate
between 0.0 and 1.0 (exclusive of the endpoint values). Call with idum
a negative integer to initialize; thereafter, do not alter idum between
successive deviates in a sequence. RNMX should approximate the largest
floating values that is less than 1."
   Numerical Recipes in C, Chapter 7-1, page 280 */
float ran1(long *idum)
{
  int j;
  long k;
  static long iy=0;
  static long iv[NTAB];
  float temp;

  if (*idum <= 0 || !iy) {          // Initialize
    if (-(*idum) < 1) *idum=1;      // Be sure to present idum=0
    else *idum = -(*idum);
    for (j=NTAB+7;j>=0;j--) {
      k=(*idum)/IQ;
      *idum=IA*(*idum-k*IQ)-IR*k;
      if (*idum < 0) *idum += IM;
      if (j < NTAB) iv[j] = *idum;
    }
    iy=iv[0];
  }
  k = (*idum)/IQ;
  *idum=IA*(*idum-k*IQ)-IR*k;
  if (*idum < 0) *idum += IM;
  j=iy/NDIV;
  iy=iv[j];
  iv[j] = *idum;
  if ((temp=AM*iy) > RNMX) return RNMX;
  else return temp;
}
//=========================================================================
==========================================
// Function from Numerical Recipes in C, 7-2, pg 289-290: "Returns a
normally distributed deviate with zero mean and unit variance, using
ran1(idum) as the source of uniform deviates
float Normal(long *idum)
{
  float ran1(long *idum);
  static int iset=0;
  static float gset;
  float fac, rsq, v1, v2;

  if (*idum < 0) iset=0;
  if (iset == 0) {
    do {
      v1 = 2.0*ran1(idum)-1.0;
      v2 = 2.0*ran1(idum)-1.0;
      rsq=v1*v1+v2*v2;
    } while (rsq >= 1.0 || rsq == 0.0);
    fac=sqrt(-2.0*log(rsq)/rsq);
```

```
    //Box-Muller transformation to get two normal deviates. Return one
and save the other for next time
    gset=v1*fac;
    iset=1;
    return v2*fac;
  } else {
    iset=0;
    return gset;
  }
}
//=====================================================================
========================================
void ChemicalPotential(ManyIons *mi, float kT, char *filename)
{
  FILE *fp;
  ion *pi;
  order *op;
  int i, j;
  double CP, P, I;
  double mean, mean_sq, rms, st_dev;
  mean = 0;
  mean_sq = 0;

  fp = fopen(filename, "a+");   // Open file
  fprintf(fp, "##Ion Number\t Number of Moves\t kT\t CP\n");

  // Loop over all atoms and calculate chemical potential for each atom
  for (j=0; j < mi->NAtoms; j++) {

    op = mi->o+j;
    i = op->Zorder;                // Ion number
    pi = mi->p+i;
    if (pi->Z >= (mi->L - (4 * RS))) { break; }
    if (pi->Z >= 4 * RS) {
      P = pi->Epot;
      I = InteractionEnergy(mi, j);
      CP = P + I;
      mean = mean + CP;
      mean_sq = mean_sq + CP*CP;

      //  fprintf(fp, "## Chemical Potential Calculations for
Wire.\n");
      //  fprintf(fp, "## Wire Radius: R = %f 1/kF \n", mi->R);
      //  fprintf(fp, "## Wire Length: L = %f 1/kF \n", mi->L);
      //  fprintf(fp, "## Number of atoms: %d \n", mi->NAtoms);
      //  fprintf(fp, "## Atom number       Chemical Potential
(eF)\n");
      fprintf(fp, "%d\t%d\t%f\t%f\n", j, pi->N_acpt, kT, CP);
// Prints ion number, number of accepted Monte-Carlo moves, and
Chemical potential to file
    }
  }
  fclose(fp);

  fp = fopen("./SimTest1/ChemicalPotentialMean.dat", "a+");
  mean = mean/(mi->NAtoms);
  mean_sq = mean_sq/(mi->NAtoms);
```

```
  st_dev = sqrt(mean_sq - mean*mean);
  //  fprintf(fp, "##Mean\t\tST.DEV\n");
  fprintf(fp, "%f\t\t%f\t\t%d\n", kT, mean, mi->NAtoms);             //
Prints in file: mean and standard deviation


  fclose(fp);                       // Close file

  mi->CPm = mean + fabs(ALPHA*mean);
  mi->CPstd = st_dev;

  return;
}

//=====================================================================
=======================================
void ShiftIons(ManyIons *mi, int j)          // j refers to ion being
removed
{
  ion *p;        // Pointer to refer to ion being removed or added
  ion *pi;       // Pointer to refer to other ion
  order *or;     // Pointer to refer to the order structure for the
removed/added ion
  order *oi;     // Pointer to refer to the order structure for the
other ion

  int j2, k, k2;  // Ion number, other ion order, other ion number
  int i;          // Looping number

  for (i = j; i < mi->NAtoms; i++) {

    or = mi->o+i;
    j2 = or->Zorder;         // Ion number
    p = mi->p+j2;            // Actual ion information

    k = i + 1;

    oi = mi->o+k;
    k2 = oi->Zorder;
    pi = mi->p+k2;

    p->X = pi->X;
    p->Y = pi->Y;
    p->Z = pi->Z;
    p->Epot = pi->Epot;
    p->Ei = pi->Ei;
    p->N_acpt = pi->N_acpt;
  }

  mi->NAtoms = mi->NAtoms - 1;
  Sort(mi);

  return;
}
//=====================================================================
==============================
void UnwrapWire(ManyIons *mi, double AvR, char *filename)
```

```
{
  FILE *fp;
  ion *p;
  int i, j;
  order *oi;
  double R;

  fp = fopen(filename, "a+");
  fprintf(fp, "##Unwrapped Wire.\n");
  fprintf(fp,
"##====================================================================
==\n");
  fprintf(fp, "##\tZ\t\tavR\n");

  // Write position information in atomic order
  for (j=0; j < mi->NAtoms; j++) {
    oi = mi->o+j;
    i = oi->Zorder;
    p = mi->p+i;
    R = AvR * atan(p->Y / p->X);

    if (p->X < 0) {
      R = R + M_PI;
    }
    fprintf(fp, "%f\t\t%f\n", p->Z, R);
  }

  fclose(fp);

  return;
}
//====================================================================
============================
void AdjustChemicalPotential(ManyIons *mi, float kT, int N, char
*filename)
{
  FILE *fp;
  double CP, CPm, CPstd;
  double gamma, alpha;
  int dN;

  fp = fopen(filename, "a+");

  dN = mi->NAtoms - N;
  CPm = mi->CPm;
  CPstd = mi->CPstd;
  gamma = CPstd / GC_INIT;
  alpha = gamma * kT/FACTOR;

  CP = CPm - alpha * (float)dN;
  mi->CPm = CP;
  fprintf(fp, "%f\t\t%f\t\t%d\n", kT, CP, mi->NAtoms);
  fclose(fp);

  return;
}
```