# Molecular Simulation of Materials, Homework 1

Matthew Grasinger

September 15, 2016

Table 1: Implementations of functions to calculate $2x^4 - 3x^4 + 4x^2 - 3$.

| **Function name** | **Description** |
|---|---|
| `part_i_slowest` | Calculates polynomial using the `pow` function. |
| `part_i_fast` | Calculates polynomial by replacing each exponent with the correspond series of multiplications (e.g. $x^6 \rightarrow x * x * x * x * x * x$) |
| `part_i_fastest` | Calculates polynomial by: (1) caching the value of $x^2 = x * x$; (2) caching the value of $x^4 = x^2 * x^2$, and (3) calculating the polynomial as: $2 * x^4 * x^2 - 3 * x^4 + 4 * x^2 - 3$ |

## 1 Problem 1

i) The polynomial computation was implemented in three templated functions of the signature `template <typename T> T f(const T)`. Each of these three functions are described in Table i) and the source code is presented in Section 1.1 on the next page.

Theoretically, `T part_i_fast(const T)` and `T part_i_fastest(const T)` should be more computationally efficient than `T part_i_slowest(const T)` because the `pow` is a computationally expensive method to calculate $x^6$, $x^4$, and $x^2$. In addition, if you count the multiplication and division operations, `T part_i_fast(const T)` requires 12 operations and `T part_i_fastest(const T)` only requires 6. Therefore, of the three implementations `T part_i_fastest(const T)` should be the fastest.

Each of the three functions were profiled to measure its computation time. The results of (1) the return value of each function for each input number, and (2) the associated computation time are presented in the output (Section 1.2 on page 5). For every case (i.e. each combination of number and return type), `part_i_slowest` had the most computing time; and in general, `part_i_fastest` required less computing time than `part_i_fast`. Therefore, the results were consistent with what one would intuitively expect given that the `pow` function is computationally expensive, and `part_i_fastest` had the least amount of multiplication operations.

ii) A function was written to determine whether or not an integer is prime, `bool is_prime(const int x)`. Its implementation is given in Section 1.1 on the next page. The algorithm works by:

(a) checking to see if $x < 1$. If $x$ is zero or negative, it cannot be prime.

(b) checking to see if $x = 1$ or $x = 2$. If $x$ is 1 or 2 then it is prime.

(c) checking to see if $x$ can be divided by 2 without remainder. If $x$ can be divided by 2 without remainder, and is not 2 (as verified by the previous step), then $x$ is even.

(d) checking each odd number less than $x$ to see if it divides $x$ without remainder. Only the odd numbers need to be checked because the previously step accounted for all of the even numbers. If an odd number is found to divide $x$ without remainder, then $x$ is not prime. Otherwise, $x$ is prime.

Although this algorithm is simple, it is theoretically twice as efficient as the brute force approach (checking every positive integer less than $x$ to see if it divides into $x$ without remainder).

iii) There is a clear difference between the polynomial functions that have a return type of `int` and those that have a return type of `double`. For the smallest numbers, 11 and 28, all of the functions agree. However, for the numbers greater than 28 (45, 397, 677, 951, 2552, 6447) the functions with return type `int` experience numerical overflow (i.e. the computed value is greater than what can be stored). This happens because an `int` is only 4 bytes (on my machine) and a double is 8 bytes.

## 1.1 Source: start.cpp

```
// start.cpp
//
// This program performs a series of mathematical operations on a set of integers read in
    ↪  from a file.
//
//*****************
#include <iostream> // this is a standard C++ header, and should be included in all
    ↪ programs
#include <fstream>
#include <iomanip>
#include <cmath>
#include <time.h>
#include "mprof.hh" // simple profile functions
using namespace std;
//*****************

// declaration of functions/subroutines

int polynomial(int i);  // a polynomial to be evaluated

template <typename T>
T part_i_slowest(const T x) {
  return 2 * pow(x, 6) - 3 * pow(x, 4) + 4 * pow(x, 2) - 3;
}

template <typename T>
T part_i_fast(const T x) {
  return 2 * x*x*x*x*x*x - 3 * x*x*x*x + 4 * x*x - 3;
}

template <typename T>
T part_i_fastest(const T x) {
  T x2 = x*x;
  T x4 = x2*x2;
  return 2 * x4*x2 - 3 * x4 + 4 * x2 - 3;
}

bool is_prime(int x);

// declaration of input and output stream

ifstream data_in("5input.txt"); // input data
ofstream data_out("5output.txt"); // output data

// the main part of the program - should always be type "int"

int main() {

  #define outs data_out // decide on output stream here

        // variable declarations
    int i,j,k; // loop counters
    const int upper = 8; // upper limit on calculation loop, corresponds to number of
        ↪ entries in the input file
                                        // declared as a "const int" so that arrays can
                                            ↪  be defined with this size
    int numbers[upper]; // array to store numbers, in C++, the array index starts at 0!

    bool sq; // boolean that indicates if the number is a perfect square
    int pol; // value of polynomial function
    int quotient, divisor, remainder;
    char square[200]; // strings for output

    // read in data from file
    for(i=0;i<upper;i++) data_in>>numbers[i];

    // calculation loop

    for(i=0;i<upper;i++){ // the loop will run over the elements in numbers[]

            // first, determine if the number is a perfect square

            sprintf(square,"␣is␣not␣a␣perfect␣square"); //default
            if(numbers[i]<0) sprintf(square,"␣is␣negative␣so␣that␣the␣concept␣of␣a␣
               ↪ perfect␣square␣is␣undefined␣for␣real␣numbers");
                else {
                        quotient = numbers[i];
                        divisor = 2;
                        sq = false;
                        while(sq == false && quotient > divisor) {
                                 quotient  = numbers[i]/divisor;
                                 remainder = numbers[i]%divisor;
                                 if (quotient == divisor && remainder == 0) {
                                         sq = true;
                                         sprintf(square,"␣is␣a␣perfect␣square");
                                 }
                                 else divisor = divisor + 1;
                        }
                }


            // second, evaluate the polynomial function

            pol = polynomial(numbers[i]);

      int resi;
      double resd;
      chrono::duration<double> elapsed_time;

      outs << "
         ↪ ======================================================================================
         ↪ n";
      outs << "calculating␣2x^6␣-␣3x^4␣+␣4x^2␣-␣3\n";
      outs << "\nresult,␣time\n\n";
      outs << setw(35) << "‘int␣part_i_slowest(int)‘"
           << setw(35) << "‘int␣part_i_fast(int)‘"
           << setw(35) << "‘int␣part_i_fastest(int)‘" << ’\n’;
      outs << "
         ↪ --------------------------------------------------------------------------------------
         ↪ n";

            tie(resi, elapsed_time) = profile(part_i_slowest<int>, numbers[i]);
      outs << setw(17) << resi << ",␣" << setw(15) << elapsed_time.count();

            tie(resi, elapsed_time) = profile(part_i_fast<int>, numbers[i]);
      outs << setw(17) << resi << ",␣" << setw(15) << elapsed_time.count();

            tie(resi, elapsed_time) = profile(part_i_fastest<int>, numbers[i]);
      outs << setw(17) << resi << ",␣" << setw(15) << elapsed_time.count();
      outs << "\n\n\n\n";

      outs << setw(35) << "‘double␣part_i_slowest(double)‘"
```

```
          << setw(35) << "‘double␣part_i_fast(double)‘"
          << setw(35) << "‘double␣part_i_fastest(double)‘" << '\n';
      outs << "
         ↪ -------------------------------------------------------------------------------------------
         ↪ n";

            tie(resd, elapsed_time) = profile(part_i_slowest<double>, static_cast<double
               ↪ >(numbers[i]));
      outs << setw(17) << resd << ",␣" << setw(15) << elapsed_time.count();

            tie(resd, elapsed_time) = profile(part_i_fast<double>, static_cast<double>(
               ↪ numbers[i]));
      outs << setw(17) << resd << ",␣" << setw(15) << elapsed_time.count();

            tie(resd, elapsed_time) = profile(part_i_fastest<double>, static_cast<double
               ↪ >(numbers[i]));
      outs << setw(17) << resd << ",␣" << setw(15) << elapsed_time.count();
      outs << "\n\n";

      if (is_prime(numbers[i]))
        outs << numbers[i] << "␣is␣prime.\n";
      else
        outs << numbers[i] << "␣is␣not␣prime.\n";

            //output the data

            outs<<"The␣number␣"<<numbers[i]<<square<<"␣and␣returns␣a␣value␣of␣"<<pol<<"␣
               ↪ when␣inserted␣into␣the␣function␣f"<<endl;

    }
}

//
   ↪ //////////////////////////////////////////////////////////////////////////////////////////
   ↪
int polynomial(int i) {
    int f;

    int x2=i*i;
    f = 3*x2*x2+4*x2-3;

    return f;
}
//
   ↪ //////////////////////////////////////////////////////////////////////////////////////////
   ↪

bool is_prime(int x) {
  if (x < 1) return false;
  if (x == 1 || x == 2) return true;
  if (x % 2 == 0) return false;

  for(int d = 3; d < x; d+=2)
    if (x % d == 0) return false;

  return true;
}
```

## 1.2 Output

```
=====================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

         ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------
         3499680,      1.714e-05          3499680,        1.08e-07          3499680,         9.7e-08



    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------
      3.49968e+06,        5.73e-07      3.49968e+06,        1.02e-07      3.49968e+06,           1e-07

11 is prime.
The number 11 is not a perfect square and returns a value of 44404 when inserted into the function f
=====================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

         ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------
       961939773,         9.6e-07        961939773,          6.6e-08        961939773,           8e-08



    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------
       9.6194e+08,        4.58e-07       9.6194e+08,           8e-08       9.6194e+08,         6.5e-08

28 is not prime.
The number 28 is not a perfect square and returns a value of 1847101 when inserted into the function f
=====================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

         ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------
      -2147483648,        8.42e-07       -584631712,         7.1e-08       -584631712,         7.4e-08



    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------
      1.65952e+10,        4.63e-07      1.65952e+10,        1.03e-07      1.65952e+10,           8e-08

45 is not prime.
The number 45 is not a perfect square and returns a value of 12309972 when inserted into the function f
=====================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

         ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------
      -2147483648,        8.79e-07       -361058976,         6.9e-08       -361058976,         7.8e-08
    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------------
      7.83013e+15,        4.76e-07      7.83013e+15,        1.06e-07      7.83013e+15,         7.4e-08

397 is prime.
The number 397 is not a perfect square and returns a value of 1507977044 when inserted into the function f
===========================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

          ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------------
      -2147483648,        8.85e-07       -402784224,         7.1e-08       -402784224,         6.9e-08



    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------------
      1.92558e+17,        4.52e-07      1.92558e+17,        1.11e-07      1.92558e+17,         5.9e-08

677 is prime.
The number 677 is not a perfect square and returns a value of -1161942476 when inserted into the function f
===========================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

          ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------------
      -2147483648,        8.62e-07       -129610176,         7.2e-08       -129610176,         7.3e-08



    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------------
      1.47949e+18,        4.92e-07      1.47949e+18,        1.02e-07      1.47949e+18,           6e-08

951 is not prime.
The number 951 is not a perfect square and returns a value of 1400797988 when inserted into the function f
===========================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3

result, time

          ‘int part_i_slowest(int)‘            ‘int part_i_fast(int)‘          ‘int part_i_fastest(int)‘
-----------------------------------------------------------------------------------------------------------
      -2147483648,        8.28e-07      -2071637763,         6.8e-08      -2071637763,         7.5e-08



    ‘double part_i_slowest(double)‘      ‘double part_i_fast(double)‘    ‘double part_i_fastest(double)‘
------------------------------------------------------------------------------------------------------------
      5.52477e+20,        4.64e-07      5.52477e+20,        1.06e-07      5.52477e+20,           8e-08

2552 is not prime.
The number 2552 is not a perfect square and returns a value of -1029852931 when inserted into the function f
===========================================================================================================
calculating 2x^6 - 3x^4 + 4x^2 - 3
```

```
result, time

          'int part_i_slowest(int)'                'int part_i_fast(int)'             'int part_i_fastest(int)'
-------------------------------------------------------------------------------------------------------------------
      -2147483648,        7.24e-07       879593088,          6.3e-08       879593088,          5.9e-08



    'double part_i_slowest(double)'       'double part_i_fast(double)'    'double part_i_fastest(double)'
--------------------------------------------------------------------------------------------------------------------
       1.43607e+23,        4.64e-07      1.43607e+23,         1.12e-07      1.43607e+23,            6e-08

6447 is not prime.
The number 6447 is not a perfect square and returns a value of -314716604 when inserted into the function f
```

## 2 Problem 2

The two papers that are of interest to me, and that have been included in this submission are *Atomistic-to-continuum multiscale modeling with long-range electrostatic interactions in ionic solids*, Marshall and Dayal [1] and *Introduction to the Kinetic Monte Carlo Method*, Voter [2]. I chose [1] because I am interested in techniques and computational approaches for multiscale modeling. I am interested in multiscale modeling because (1) it is a relatively new area of research in numerical methods, (2) it presents challenges that are non-trivial from an analysis, or mathematical standpoint (e.g. proving uniqueness and existance of solutions, well-posedness, etc.), and from a standpoint of computational efficiency and (3) there are many problems in engineering that require information and analysis on multiple scales (e.g. crack initiation at material defects, solid electrolytes that conduct current through the motion of charged point defects, etc.). The multiscale approach presented in [1] uses a lot of the concepts that we have covered in class such as short-range, pairwise, Lennard-Jones potentials, but also addresses other topics such as how to efficiently model long-range potentials such as electrostatic potentials.

My interest in [2], and the Kinetic Monte Carlo in general, stems from the fact that I find it interesting that the evolution of a physical system can be approximated from random sampling and a few additional constraints. What is presented in [2] is an overview of different Kinetic Monte Carlo methods with the goal of helping the reader to assess other papers using Kinetic Monte Carlo and to give the reader the basic tools necessary to develop their own Kinetic Monte Carlo application. As discussed in class, molecular dynamics simulations are typically limited to about a microsecond of simulation time. It is interesting how Kinetic Monte Carlo methods try to overcome this limitation in time-scale by (often) no longer attempting to follow the exact trajectory of particles through their every vibrational period, but instead focus on dynamics of a coarser time-scale.

## 3 Problem 3

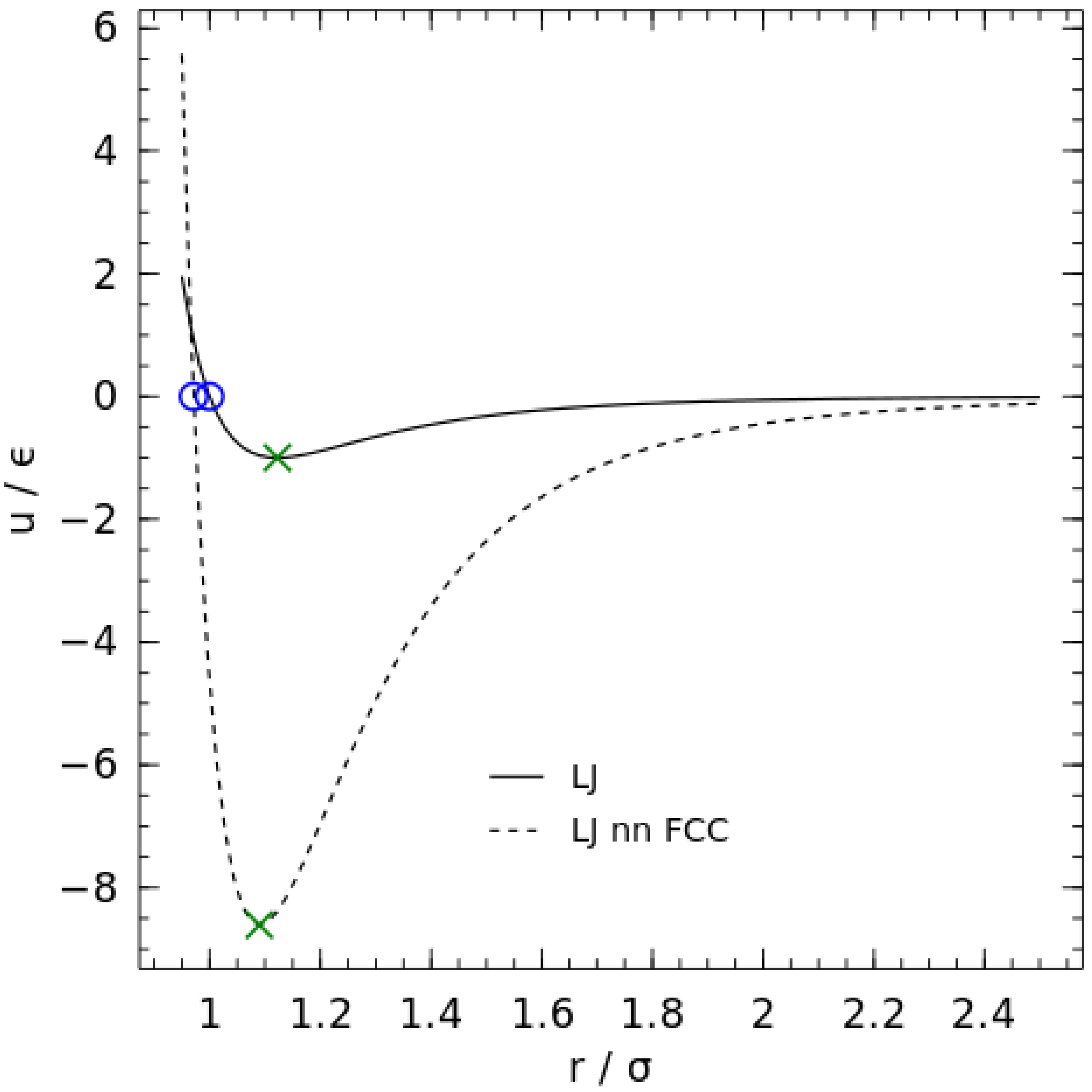
a) If we assume the water droplets are at room temperature, 25 C, then the density is approximately 0.997044 g/cm$^3$. In addition, water molecules are approximately 10 g/mol, or 10 g per $6.022 \times 10^{23}$ molecules. Using both the density and atomic mass of water, we can obtain:

$$\frac{\rho}{10 \text{ g}/6.022 \times 10^{23}} = \frac{0.997044 \text{ g/cm}^3}{10 \text{ g}/6.022 \times 10^{23}} = 6.004 \times 10^{23} \text{cm}^{-3} = 6.004 \times 10^{2} \text{ nm}^{-3}. \tag{1}$$

For spherical droplets of diameter 1 nm, 10 nm, and 100 nm, the volume is $\pi/4$ nm$^3$, $25\pi$ nm$^3$, and $2500\pi$ nm$^3$, respectively. Plugging these volumes in (1) results in 471, 47155, and $4.7155 \times 10^6$ molecules, respectively.

b) For each pair of water molecules, there are 9 distinct pair interactions; and for $N$ molecules, we can choose $\frac{N(N-1)}{2}$ distinct pairs of molecules. The number of distinct pair interactions for $N$ molecules is therefore $9\frac{N(N-1)}{2}$. The number distinct pair interactions in a spherical water droplet with diameter 1 nm, 10 nm, and 100 nm are 996165, $10^{10}$, and $10^{14}$, respectively.

## 4 Problem 4



a)
The LJ potential for a stable crystal structure (FCC) has a deeper potential well, has a longer range attraction, and tends to infinity more quickly when $r \to 0$.

b) The LJ potential is given by:

$$u(r) = 4\epsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right], \tag{2}$$

and the LJ potential in a stable, FCC crystal is given by:

$$U(r_{nn}) = 2\epsilon\left[A_{12}\left(\frac{\sigma}{r_{nn}}\right)^{12} - A_6\left(\frac{\sigma}{r_{nn}}\right)^{6}\right], \tag{3}$$

where $A_{12} = 12.13$ and $A_6 = 14.45$. Clearly, (2) is equal to zero when $r = \sigma$. To find the zero of (3) we'll use a few steps of algebra:

$$\begin{aligned} 0 &= A_{12}\left(\frac{\sigma}{r_{nn}}\right)^{12} - A_6\left(\frac{\sigma}{r_{nn}}\right)^{6}, \\ 0 &= A_{12}^{1/6}\left(\frac{\sigma}{r_{nn}}\right)^{2} - A_6^{1/6}\left(\frac{\sigma}{r_{nn}}\right), \\ A_6^{1/6}\left(\frac{\sigma}{r_{nn}}\right) &= A_{12}^{1/6}\left(\frac{\sigma}{r_{nn}}\right)^{2}, \\ \frac{\sigma}{r_{nn}} &= \left(\frac{A_6}{A_{12}}\right)^{1/6}, \\ r_{nn} &= \sigma\left(\frac{A_6}{A_{12}}\right)^{-1/6}, \end{aligned}$$

To minimize (2) we'll take the derivative with respect to $r$ and set it equal to zero.

$$\begin{aligned} \frac{du}{dr} = 0 &= 4\epsilon\left[-12\sigma^{12}r^{-13} + 6\sigma^6 r^{-7}\right], \\ 0 &= \sigma^6 r^{-7} - 2\sigma^{12}r^{-13}, \\ 2\sigma^{12}r^{-13} &= \sigma^6 r^{-7}, \\ r^{-6} &= \frac{1}{2}\sigma^{-6}, \\ r &= \left(\frac{1}{2}\right)^{-1/6}\sigma. \end{aligned}$$

Similarly, minimizing (3) results from:

$$r = \left(\frac{1}{2}\frac{A_6}{A_{12}}\right)^{-1/6}\sigma.$$

c) $\frac{\sqrt{\epsilon}k_B}{\sqrt{m}\sigma^2}$.

d) For argon, the dimensionless temperature is 20 K$/(\epsilon/k_B) = 0.165$ and the dimensionless thermal conductivity is 1.4 W/mK$/\frac{\sqrt{\epsilon}k_B}{\sqrt{m}\sigma^2} = 73.9$. For krypton, the dimensionless temperature is 0.123 and the dimensionless thermal conductivity is 106.

## References

[1] J. Marshall and K. Dayal. Atomistic-to-continuum multiscale modeling with long-range electrostatic interactions in ionic solids. *Journal of the Mechanics and Physics of Solids*, 62:137–162, 2014.

[2] A. F. Voter. Introduction to the kinetic monte carlo method. In *Radiation Effects in Solids*, pages 1–23. Springer, 2007.