Московский государственный университет имени М.В.Ломоносова
Факультет вычислительной математики и кибернетики

Павлишин Кирилл Юрьевич

Определение потенциалов межатомного взаимодействия для системы V/Ag (001)

КУРСОВОЙ ПРОЕКТ

Москва
2022

Содержание

1 Постановка задачи

Определить потенциалы межатомного взаимодействия (A-A, A-B, B-B) для системы A/B (001)

Вид потенциалов:

$$E = \sum_i E_R^i + E_B^i$$

$$E_B^i = -\sqrt{(\sum_j \xi_{\alpha\beta}^2 \, exp\,(-2q_{\alpha\beta}\,(\frac{r_{ij}}{r_0^{\alpha\beta}} - 1)))}$$

$$E_R^i = \sum_j (A_{\alpha\beta}^1\,(r_{ij} - r_0^{\alpha\beta}) + A_{\alpha\beta}^0)\, exp\,(-p_{\alpha\beta}\,(\frac{r_{ij}}{r_0^{\alpha\beta}} - 1))$$

E - полная знергия системы

E_B^i - энергия притяжения

E_R^i - энергия отталкивания

r_{ij} - расстояние между атомами i и j

Параметры потенциалов: A^1, A^0, ξ, p, q, r_0 (для A-A, A-B и B-B взаимодействий 3 набора по 6 параметров) необходимо определить путём минимизации функции ошибки относительно известных табличных величин.

Расчёты полных энергий E проводятся для кристаллической решётки $3 \times 3 \times 3$ в единицах элементарной ячейки ГЦК структуры. Все расчёты статические, проводятся без релаксации атомных позиций. Атомы располагаются в узлах “идеальной” ГЦК решётки.

Полученное решение оптимизировалось с помощью технологий параллельного программирования для уменьшения времени расчётов.

2 Вычисление характиристик

На рисунке 1 представлена ГЦК.

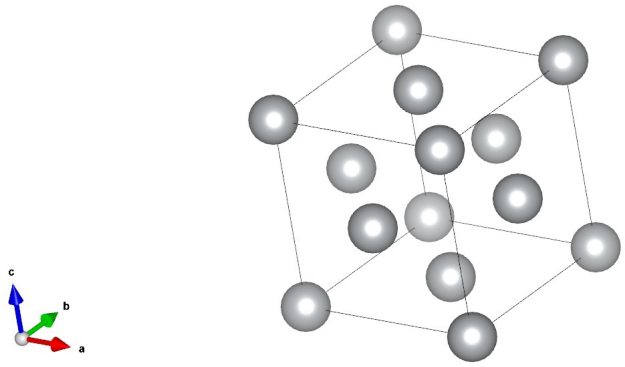

Рис. 1: ГЦК решётка

Для построения кристаллической решётки $3 \times 3 \times 3$ в единицах элементарной ячейки ГЦК структуры нужно выделить из ГЦК элементарную ячейку, которая изображена на рисунке 2, убрав из неё дублирующиеся (из-за периодических граничных условий) атомы, и размножить её по трём направлениям.

Рис. 2: Элементарная ячейка

Полученная кристаллическая решётка $3 \times 3 \times 3$ изображена на рисунке 3.

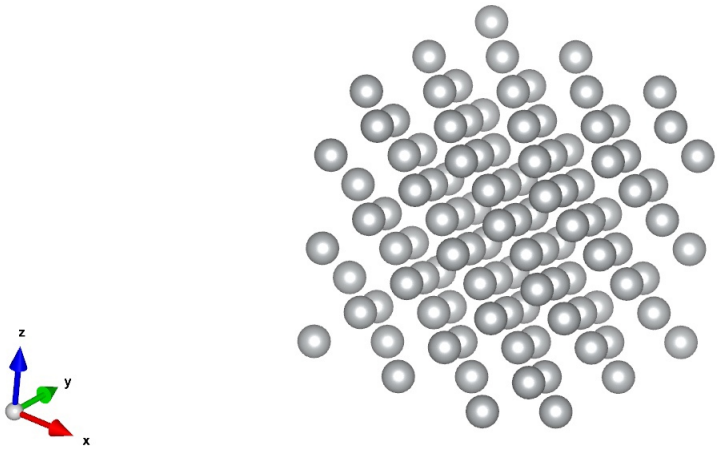

Рис. 3: Кристаллическая решётка $3 \times 3 \times 3$

Для оптимизации параметров потенциала в качестве целевого функционала минимизации был выбран следующий вид функционала:

$$f = \sqrt{\frac{1}{n} \sum_{i}^{n} \frac{(tableValue_i - value_i)^2}{tableValue_i^2}}$$

В данном целевом функционале $tableValue_i$ - известные табличные значения кристаллической решётки, $value_i$ - значения, посчитанные с текущим набором параметров потенциала.

Для оптимизации парметров потенциала для взаимодействия атомов сорта B-B в целевой функционал минимизации входят следующие значения кристаллической решётки:

1. a - параметр решётки, который находится как точка минимума полной энергии системы E по пяти точкам из возможного диапазона значений

2. E_{coh} - когезионная энергия:

$$E_{coh} = \frac{E}{N}$$

E - полная энергия

N - количество атомов в решётке

3. B - модуль всестороннего растяжения (сжатия):

$$B = \frac{2}{9\,V_0}\,\frac{\partial^2 E_B}{\partial \alpha^2}$$

V_0 - равновесный объём

Для вычисления этого значения была использована матрица деформации:

$$D = \begin{pmatrix} 1+\alpha & 0 & 0 \\ 0 & 1+\alpha & 0 \\ 0 & 0 & 1+\alpha \end{pmatrix}$$

4. C_{11}, C_{12}, C_{44} - константы упругости:

$$C_{11} = \frac{1}{V_0}\,\frac{(\frac{\partial^2 E_{C_{11}}}{\partial \alpha^2} + \frac{\partial^2 E_{C_{12}}}{\partial \alpha^2})}{2}$$

$$C_{12} = \frac{1}{V_0}\,\frac{(\frac{\partial^2 E_{C_{11}}}{\partial \alpha^2} - \frac{\partial^2 E_{C_{12}}}{\partial \alpha^2})}{2}$$

$$C_{44} = \frac{1}{V_0}\,\frac{\partial^2 E_{C_{44}}}{\partial \alpha^2}$$

Для вычисления этих значений были использованы матрицы деформации:

$$D_{C_{11}} = \begin{pmatrix} 1+\alpha & 0 & 0 \\ 0 & 1+\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$D_{C_{12}} = \begin{pmatrix} 1+\alpha & 0 & 0 \\ 0 & 1-\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

$$D_{C_{44}} = \begin{pmatrix} 1 & \alpha & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & \frac{1}{1-\alpha^2} \end{pmatrix}$$

В результате функционал ошибки принимает следующий вид:

$$f = \sqrt{\frac{1}{6}\left(\frac{(a-a^*)^2}{a^{*2}} + \frac{(E_{coh}-E^*_{coh})^2}{{E^*_{coh}}^2} + \frac{(B-B^*)^2}{B^{*2}} + \frac{(C_{11}-C^*_{11})^2}{{C^*_{11}}^2} + \frac{(C_{12}-C^*_{12})^2}{{C^*_{12}}^2} + \frac{(C_{44}-C^*_{44})^2}{{C^*_{44}}^2}\right)}$$

Для оптимизации парметров потенциала для взаимодействия атомов сорта A-B в целевой функционал минимизации входят следующие значения кристаллической решётки:

1. E_{sol} - энергия растворимости примеси сорта A в кристалле сорта B:

$$E_{sol} = E^{AB} - E^B - E^A_{coh} + E^B_{coh}$$

E^{AB} - полная энергия кристаллической решётки элемента сорта B с примесью замещения сорта A

E^B - полная энергия кристаллической решётки элемента сорта B

E^A_{coh} - когезионная энергия элемента сорта A (берётся из справочника)

E^B_{coh} - когезионная энергия элемента сорта B

В результате функционал ошибки принимает следующий вид:

$$f = \sqrt{\frac{(E_{sol}-E^*_{sol})^2}{{E^*_{sol}}^2}}$$

Для оптимизации парметров потенциала для взаимодействия атомов сорта A-A в целевой функционал минимизации входят следующие значения кристаллической решётки:

1. E^{in}_{dim} - энергия связи димера сорта A в поверхностном слое сорта B:

$$E^{in}_{dim} = E^{dim+surf} - E^{surf} - N\left(E^{adatom+surf} - E^{surf}\right)$$

$E^{dim+surf}$ - полная энергия структуры поверхности с димером сорта A в верхнем слое

E^{surf} - полная энергия структуры поверхности

$E^{adatom+surf}$ - полная энергия структуры поверхности с одним атомом сорта A в верхнем слое

2. E^{on}_{dim} - энергия связи димера сорта A на поверхностном слое сорта B:

$$E_{dim}^{on} = E^{dim+surf} - E^{surf} - N\,(E^{adatom+surf} - E^{surf})$$

$E^{dim+surf}$ - полная энергия структуры поверхности с димером сорта A над верхним слоем

E^{surf} - полная энергия структуры поверхности

$E^{adatom+surf}$ - полная энергия структуры поверхности с одним атомом сорта A над верхним слоем

В результате функционал ошибки принимает следующий вид:

$$f = \sqrt{\frac{1}{2}\,(\frac{(E_{dim}^{in} - {E_{dim}^{in}}^{*})^2}{{E_{dim}^{in}}^{*2}} + \frac{(E_{dim}^{on} - {E_{dim}^{on}}^{*})^2}{{E_{dim}^{on}}^{*2}})}$$

3 Ход работы

Расчёты проводились для системы A/B (001): V/Ag (001).

Параметры потенциалов для взаимодействий A-A, A-B, B-B оптимизировались с помощью алгоритма минимизации Нелдера-Мида с ограничениями параметров.

Для всех трёх типов взаимодействий A-A, A-B, B-B были выбраны следующие ограничения на параметры потенциала:

```
A1: 0 - 0.1
A0: 0.0685 - 0.137
ksi: 0.7853 - 1.57067
p: 7.2853 - 14.5706
q: 2.0927 - 4.1853
r0: 1.9257 - 3.8514
```

Начальные параметры потенциала для запуска оптимизатора для всех трёх типов взаимодействий A-A, A-B, B-B случайно генерируются из тех же промежутков, как и ограничения на параметры потенциала.

Для распараллеливания вычислений на потоки применялачь технология OpenMP.

Известные табличные значения кристаллической решётки представлены в таблице 1.

Таблица 1: Табличные значения кристаллической решётки

Название	Значение
a	4.085
E_{coh}	-2.960
B	1.08
C_{11}	1.32
C_{12}	0.97
C_{44}	0.51
E_{sol}	0.497
E_{dim}^{in}	0.22
E_{dim}^{on}	-0.36

Для вычисления E_{sol}, E_{dim}^{in}, E_{dim}^{on} примесные атомы сорта A помещались в определённые места кристаллической решётки $3 \times 3 \times 3$ в зависимости от типа считаемой энергии.

При расчёте E_{sol} энергия считается в бесконечном кристалле и расположение примесного атома сорта A внутри кристаллической решётки может быть любым, так как периодические граничные условия действуют по всем трём направлениям. Расположение примесного атома для этого случая представлено на рисунке 4.

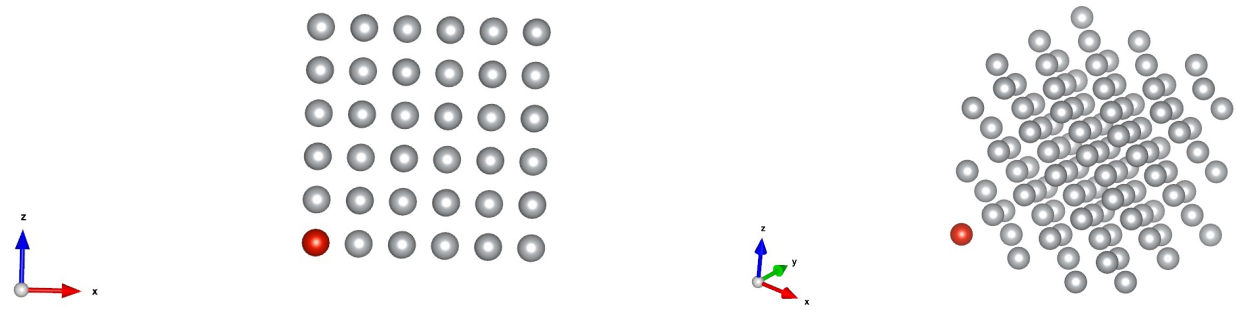

Рис. 4: Примесный атом сорта A внутри кристаллической решётки

При расчёте E_{dim}^{in} энергия считается уже не в бесконечном кристалле, а в поверхностном слое, и примесный димер или атом сорта A располагается внутри этого поверхностного слоя кристаллической решётки. Периодические граничные условия по оси Z в этом случае не действуют. Расположение примесного димера для этого случая представлено на рисунке 5.

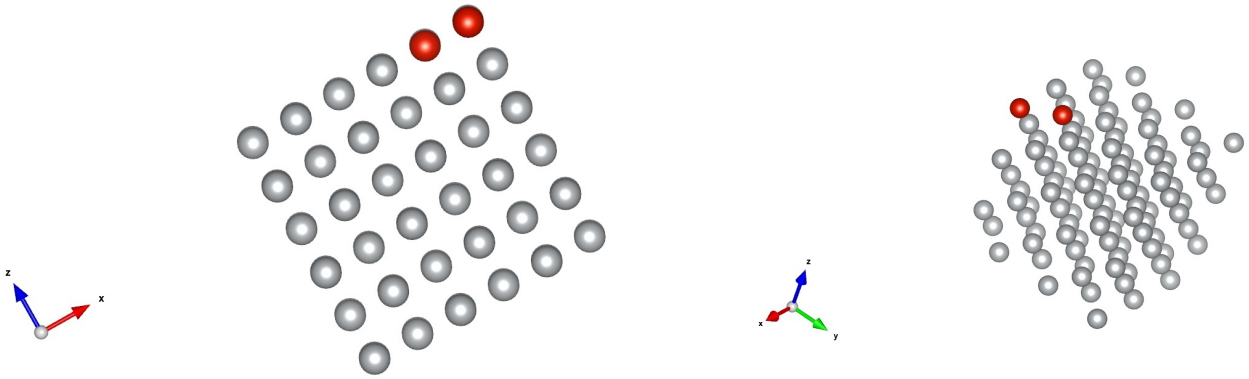

Рис. 5: Примесный димер сорта A в поверхностном слое кристаллической решётки

При расчёте E_{dim}^{on} энергия считается также не в бесконечном кристалле, а используя поверхностный слой, и примесный димер или атом сорта A располагается над поверхностным слоем кристаллической решётки. Периодические граничные условия по оси Z в этом случае также не действуют. Расположение примесного димера для этого случая представлено на рисунке 6.

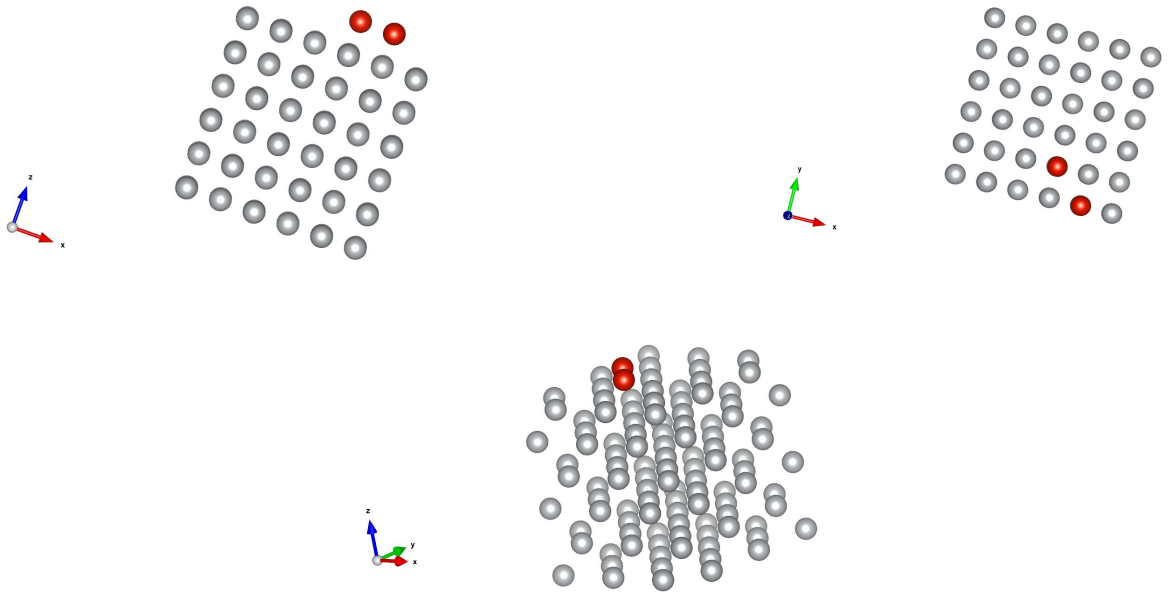

Рис. 6: Примесный димер сорта A над поверхностнымм слоем кристаллической решётки

4 Описание алгоритма минимизации Нелдера-Мида

Данный метод также известен как метод деформируемого многогранника. В основе этого метода лежит движение симплекса в пространстве параметров в сторону локального минимума целевой функции.

Параметрами метода являются:

1. Коэффициент отражения $\alpha > 0$, обычно выбирается равным 1
2. Коэффициент сжатия $\beta > 0$, обычно выбирается равным 0.5
3. Коэффициент растяжения $\gamma > 0$, обычно выбирается равным 2
4. Коэффициент глобального сжатия $\sigma > 0$, обычно выбирается равным 0.5

Алгоритм работы метода:

1. *Инициализация метода*. Случайным образом выбираются $n + 1$ точка в n -мерном пространстве параметров, образующие симплекс, и в них вычисляются значения целевого функционала: $x_i = (x_i^{(1)}, x_i^{(2)}, \dots, x^{(n)i}), F(x_i) = f_i$.
2. Выберем из вершин симплекса три: x_h , x_g и x_l , такие что f_h — наибольшее значение целевой функции из всех вершин симплекса, f_g — второе по величине значение и f_l — наименьшее.

   Для этого можно на каждой итерации сортировать все вершины по значению целевой функции в них, либо хранить в структуре данных `std::multimap`, которая умеет отвечать на запросы о наибольших и наименьших значениях, хранящихся в ней.
3. Найдём центр тяжести всех точек за исключением x_h : $x_c = \frac{1}{n}\sum_{i \neq h} x_i$.
4. Отразим точку x_h относительно точки x_c с коэффициентом α. Получим таким образом точку $x_r = (1 + \alpha)x_c - \alpha x_h$ и вычислим в ней значение целевого функционала f_r.

5. Проверим на сколько нам удалось улучшить значение функции:

   - $f_r < f_l$ — направление выбрано удачно, попробуем увеличить шаг. Вычислим новую точку $x_e = (1-\gamma)x_c + \gamma x_r$ и значение функции в ней f_e .

     Из точек x_r и x_e выбираем наилучшую и заменяем ею x_h.

   - $f_l \le f_r < f_g$ — новая точка улучшает ответ, заменим ею x_h.

   - $f_g \le f_r < f_h$ — новая точка улучшает ответ, но слабо, заменим ею x_h и проведём операцию сжатия.

   - $f_h \le f_r$ — новая точка не улучшает ответ, проведём операцию сжатия.

6. *Сжатие*. Если было решено провести операцию сжатия, то построим новую точку $x_s = (1-\beta)x_c + \beta x_h$ и вычислим значение функции в ней f_s.

   - $f_s < f_h$ — заменяем вершину x_h точкой x_s.

   - $f_h \le f_s$ — сжимаем весь симплекс к точке с наименьшим значением $x_i \leftarrow x_l + \sigma(x_i - x_l)$.

7. *Проверка сходимости*. Проверять сходимость можно разными способами. Одним из таких является падение дисперсии в результатах расчета функции ниже некоторого порога. Однако такой способ может привести к слишком ранней остановке оптимизации, если функция достаточно пологая. Другим вариантом является проверка размеров симплекса - расстояния между двумя самыми далёкими вершинами. Этот способ работает хуже в случае существенно различных масштабов переменных и требует нормализации. Пока критерий сходимости не выполнен, возвращаемся к первой итерации.

5 Технология распараллеливания

Для распараллеливания в программе использовалась технология OpenMP.

Для реализации параллелизма OpenMP использует модель fork-join, которая изображена на рисунке 7.

Рис. 7: Модель распараллеливания fork-join

Программист может пометить блок (секцию) кода для параллельного выполнения. Дойдя до этого блока, программа должна будет породить группу потоков, которые будут выполнять его сообща. Время работы каждого потока может быть различным, однако выполнение кода после параллельной секции будет продолжено только после завершения работы всех потоков и их синхронизации. По умолчанию количество потоков будет равно количеству доступных ядер процессора, но его можно настраивать как во время выполнения программы, так и с помощью переменных окружения во время её запуска. Память в OpenMP разделяется на локальную и общую. Локальная память доступна только для одного конкретного потока, а общая - для всех потоков.

6 Распараллеливание программы

В ходе работы приходится многократно считать полную энергию кристаллической решётки E, которая состоит из совокупности энергий взаимодействия каждого атома с его окружением. И расчёт полной энергии одного атома не зависим относительно остальных. Для этого уместно использование технологий параллельного программирования, которое даст неплохой прирост в сорости вычислений.

Как было написано выше, для распараллеливания вычислений используется технология OpenMP. В последовательной части кода создаётся массив, который хранит результирующую информацию для каждого потока. Количество потоков соответствует числу логических ядер процессора, что в моём случае равно 4.

В параллельной части кода каждому потоку раздаётся своя часть атомов кристаллической решётки, и поток считает энергии взаимодействия между атомами этой части. Посчитанную энергию своей части кристаллической решётки каждый поток кладёт в результирующий массив в ячейку, соответствующую этому потоку.

После завершения параллельной части кода программа ждёт завершения работы каждого из потоков. Далее считается полная энергия взаимодействия для решётки, равная сумме значений из результирующего массива, в котором лежат посчитанные энергии от каждого потока.

7 Результаты работы программы

В ходе работы программы были получены следующие результаты:

```
B-B:
A1 = 0
A0 = 0.0807582
ksi = 1.10041
p = 11.269
q = 3.21654
r0 = 2.96157

a0 = 4.10232
E_coh = -2.96268
B = 1.08331
c11 = 1.31945
c12 = 0.965229
c44 = 0.509639
```

```
A-B:
A1 = 0.099
A0 = 0.13563
ksi = 1.51224
p = 8.32672
q = 2.7878
r0 = 2.90354

E_sol = 0.497
```

```
A-A:
A1 = 0.064906
A0 = 0.112536
ksi = 1.43775
p = 9.70314
q = 2.16807
r0 = 3.0382

E_in = 0.22
E_on = -0.36
```

Для проверки корректности полученных параметров построим графики зависимости полной энергии от расстояния между атомами для A-A, A-B и B-B взаимодействий. Полученные графики представленны на рисунке 8.

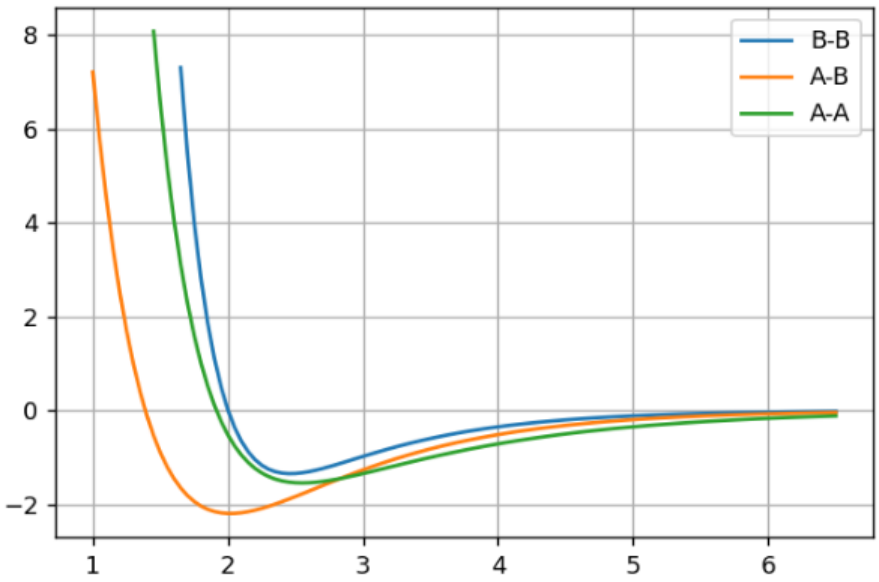

Рис. 8: Графики потенциалов

По графикам видно, что их поведение удовлетворяет каноническому поведению потенциала.

Для оптимизации скорости вычислений применялась компиляция с различными ключами компиляции, а также с применением технологии многопоточного программирования OpenMP.

Результат программы не зависит от ключей компиляции, количества потоков и использования технологий многопоточного программирования. Они влияют только на время работы программы. На значения оптимизируемых параметров влияет только зерно для генерации случайных чисел.

Время работы программы и результаты варьировались в зависимости от удачности выбранного зерна. Для сравнения времени выполнения программы в зависимости от ключей компиляции, количества потоков и использования технологий многопоточного программирования было выбранно зерно 4259707270, которое к тому же даёт наилучшие результаты по отношению к другим зёрнам, которые я брал. Время работы программы при выбранном зерне в зависимости от ключей компиляции, количества потоков и использования технологий многопоточного программирования приведено в таблице 2.

Таблица 2: Время работы программы

Ключи компиляции	Время выполнения
Без ключей компиляции	999 сек
-fopenmp (2 потока)	501 сек
-fopenmp (4 потока)	258 сек
-O3	235 сек
-march=native	1005 сек
-O3 -march=native	192 сек
-O3 -march=native -fopenmp (2 потока)	94 сек
-O3 -march=native -fopenmp (4 потока)	50 сек

По таблице видно, что добавление ключей компиляции и использование технологий параллельного программирования позволяет существенно уменьшить время выполнения программы.

8 Выводы

В ходе данного курсового проекта была реализована программа, которая позволяет определить потенциалы межатомного взаимодействия для системы A/B (001). Были получены результаты сопоставимые с действительностью. Применяемый метод оптимизации (метод Нелдера-Мида) конкретно для данной задачи очень сильно зависит от начальных параметров и граничных условий на параметры. Если не задавать границы изменения параметров, то метод находит минимум функционала, но зачастую этот минимум не соответствует действительным параметрам. Поэтому приходится задавать ограничения и делать случайные рестарты программы в надежде на то, что метод выдаст параметры, близкие к действительным.

Распараллеливание программы и добавление ключей компиляции даёт значительный прирост в скорости выполнения программы, что, несомненно, будет давать огромный выигрыш при увеличении кристаллической решётки. К тому же, за счёт того, что при распараллеливании программа работает быстрее, можно быстрее делать случайные рестарты, что позволит быстрее подобрать верные с точки зрения реальности параметры.

9 Код программы

```
#include <iostream>
#include <string>
#include <vector>
#include <cmath>
#include <ctime>
#include <fstream>
#include <omp.h>

using namespace std;

double d_edin[9] = { 1,0,0, 0,1,0, 0,0,1 };
double true_parameters[6] = {4.085, -2.960, 1.08, 1.32, 0.97, 0.51};
double true_params[2] = {0.22, -0.36};
double true_parameter = 0.497;
double Array[6] = {};
double Array2[12] = {};
double E_cohesion;
int ind_min_global;
double a0_global;
double h;
double Params_bounds[6][2] = {{0.0, 0.1}, {0.0685, 0.1370}, {0.7853, 1.570666},
    {7.2853, 14.5706}, {2.0927, 4.1853}, {1.9257, 3.8514}};

class Atom
{
    public:
        string name;
        double x, y, z;

        Atom(double _x, double _y, double _z, string _name)
        {
            setAtom(_x, _y, _z, _name);
        }

        void setAtom(double _x, double _y, double _z, string _name)
        {
            x = _x;
            y = _y;
            z = _z;
            name = _name;
        }

        void getAtom()
        {
            cout << name << endl;
            cout << x << endl;
            cout << y << endl;
```

```
            cout << z << endl;
            cout << endl;
        }

        void changeAtom(string _name)
        {
            name = _name;
        }
};

vector<Atom> GCK(string name, double a0)
{
    vector <Atom>Atoms_GCK;

    for(int i=0; i<3; i++)
    {
        for(int j=0; j<3; j++)
        {
            for(int k=0; k<3; k++)
            {
                Atoms_GCK.push_back(Atom((0+i)*a0, (0+j)*a0, (0+k)*a0, name));
                Atoms_GCK.push_back(Atom((0+i)*a0, (0.5+j)*a0, (0.5+k)*a0, name));
                Atoms_GCK.push_back(Atom((0.5+i)*a0, (0.5+j)*a0, (0+k)*a0, name));
                Atoms_GCK.push_back(Atom((0.5+i)*a0, (0+j)*a0, (0.5+k)*a0, name));
            }
        }
    }

    return Atoms_GCK;
}

double Energy(const vector <Atom>&Vect, double a0, double d_trans[9], double A1,
    double A0, double ksi, double p, double q, double r0)
{
    double Energy = 0;

    for (int i=0; i<Vect.size(); i++)
    {
        double Er = 0, Eb = 0;

        double a0_x = a0 * d_trans[0] + a0 * d_trans[1] + a0 * d_trans[2];
        double a0_y = a0 * d_trans[3] + a0 * d_trans[4] + a0 * d_trans[5];
        double a0_z = a0 * d_trans[6] + a0 * d_trans[7] + a0 * d_trans[8];

        double cutoff = 1.7 * fmax(fmax(a0_x, a0_y), a0_z);

        for (int j=0; j<Vect.size(); j++)
        {
            for (int dx=-1; dx<2; dx++)
```

```
                for (int dy=-1; dy<2; dy++)
                    for (int dz=-1; dz<2; dz++)
                    {
                        if (i != j || dx !=0 || dy !=0 || dz !=0 )
                        {
                            double tmp_x = Vect[j].x + a0*3*dx;
                            double tmp_y = Vect[j].y + a0*3*dy;
                            double tmp_z = Vect[j].z + a0*3*dz;

                            tmp_x = tmp_x * d_trans[0] + tmp_y * d_trans[1] + tmp_z *
    d_trans[2];
                            tmp_y = tmp_x * d_trans[3] + tmp_y * d_trans[4] + tmp_z *
    d_trans[5];
                            tmp_z = tmp_x * d_trans[6] + tmp_y * d_trans[7] + tmp_z *
    d_trans[8];

                            tmp_x -= Vect[i].x * d_trans[0] + Vect[i].y * d_trans[1]
   + Vect[i].z * d_trans[2];
                            tmp_y -= Vect[i].x * d_trans[3] + Vect[i].y * d_trans[4]
   + Vect[i].z * d_trans[5];
                            tmp_z -= Vect[i].x * d_trans[6] + Vect[i].y * d_trans[7]
   + Vect[i].z * d_trans[8];

                            double rij = sqrt(tmp_x*tmp_x + tmp_y*tmp_y + tmp_z*tmp_z
   );

                            if (rij <= cutoff)
                            {
                                Er = Er + ((A1/r0)*(rij-r0)+A0)*exp(-p*(rij/r0-1));

                                Eb = Eb + ksi*ksi*exp(-2*q*(rij/r0-1));
                            }
                        }
                    }
        }

        Eb = -sqrt(Eb);
        Energy += Er + Eb;
    }

    return Energy;
}

double Energy_parallel(const vector <Atom>&Vect, double a0, double d_trans[9], double
    A1, double A0, double ksi, double p, double q, double r0, int i)
{
    double Energy = 0;

    double Er = 0, Eb = 0;
```

```
double a0_x = a0 * d_trans[0] + a0 * d_trans[1] + a0 * d_trans[2];
double a0_y = a0 * d_trans[3] + a0 * d_trans[4] + a0 * d_trans[5];
double a0_z = a0 * d_trans[6] + a0 * d_trans[7] + a0 * d_trans[8];

double cutoff = 1.7 * fmax(fmax(a0_x, a0_y), a0_z);

for (int j=0; j<Vect.size(); j++)
{
    for (int dx=-1; dx<2; dx++)
        for (int dy=-1; dy<2; dy++)
            for (int dz=-1; dz<2; dz++)
            {
                if (i != j || dx !=0 || dy !=0 || dz !=0 )
                {
                    double tmp_x = Vect[j].x + a0*3*dx;
                    double tmp_y = Vect[j].y + a0*3*dy;
                    double tmp_z = Vect[j].z + a0*3*dz;

                    tmp_x = tmp_x * d_trans[0] + tmp_y * d_trans[1] + tmp_z *
d_trans[2];
                    tmp_y = tmp_x * d_trans[3] + tmp_y * d_trans[4] + tmp_z *
d_trans[5];
                    tmp_z = tmp_x * d_trans[6] + tmp_y * d_trans[7] + tmp_z *
d_trans[8];

                    tmp_x -= Vect[i].x * d_trans[0] + Vect[i].y * d_trans[1] +
Vect[i].z * d_trans[2];
                    tmp_y -= Vect[i].x * d_trans[3] + Vect[i].y * d_trans[4] +
Vect[i].z * d_trans[5];
                    tmp_z -= Vect[i].x * d_trans[6] + Vect[i].y * d_trans[7] +
Vect[i].z * d_trans[8];

                    double rij = sqrt(tmp_x*tmp_x + tmp_y*tmp_y + tmp_z*tmp_z);

                    if (rij <= cutoff)
                    {
                        Er = Er + ((A1/r0)*(rij-r0)+A0)*exp(-p*(rij/r0-1));

                        Eb = Eb + ksi*ksi*exp(-2*q*(rij/r0-1));
                    }
                }
            }
}

Eb = -sqrt(Eb);
Energy += Er + Eb;

return Energy;
```

```
}

double Energy_AB(vector <Atom>&Vect, double a0, double d_trans[9], double A1, double
    A0, double ksi, double p, double q, double r0, double x[6])
{
    double Energy = 0;

    for (int i=0; i<Vect.size(); i++)
    {
        double Er = 0, Eb = 0;

        double a0_x = a0 * d_trans[0] + a0 * d_trans[1] + a0 * d_trans[2];
        double a0_y = a0 * d_trans[3] + a0 * d_trans[4] + a0 * d_trans[5];
        double a0_z = a0 * d_trans[6] + a0 * d_trans[7] + a0 * d_trans[8];

        double cutoff = 1.7 * fmax(fmax(a0_x, a0_y), a0_z);

        for (int j=0; j<Vect.size(); j++)
        {
            for (int dx=-1; dx<2; dx++)
                for (int dy=-1; dy<2; dy++)
                    for (int dz=-1; dz<2; dz++)
                    {
                        if (i != j || dx !=0 || dy !=0 || dz !=0 )
                        {
                            double tmp_x = Vect[j].x + a0*3*dx;
                            double tmp_y = Vect[j].y + a0*3*dy;
                            double tmp_z = Vect[j].z + a0*3*dz;

                            tmp_x = tmp_x * d_trans[0] + tmp_y * d_trans[1] + tmp_z *
    d_trans[2];
                            tmp_y = tmp_x * d_trans[3] + tmp_y * d_trans[4] + tmp_z *
    d_trans[5];
                            tmp_z = tmp_x * d_trans[6] + tmp_y * d_trans[7] + tmp_z *
    d_trans[8];

                            tmp_x -= Vect[i].x * d_trans[0] + Vect[i].y * d_trans[1]
   + Vect[i].z * d_trans[2];
                            tmp_y -= Vect[i].x * d_trans[3] + Vect[i].y * d_trans[4]
   + Vect[i].z * d_trans[5];
                            tmp_z -= Vect[i].x * d_trans[6] + Vect[i].y * d_trans[7]
   + Vect[i].z * d_trans[8];

                            double rij = sqrt(tmp_x*tmp_x + tmp_y*tmp_y + tmp_z*tmp_z
   );

                            if (rij <= cutoff)
                            {
                                if((Vect[i].name == "Ag") && (Vect[j].name == "Ag"))
```

```
                                {
                                    Er = Er + ((x[0]/x[5])*(rij-x[5])+x[1])*exp(-x
    [3]*(rij/x[5]-1));

                                    Eb = Eb + x[2]*x[2]*exp(-2*x[4]*(rij/x[5]-1));
                                }
                                else if(((Vect[i].name == "V") && (Vect[j].name == "
    Ag")) || ((Vect[i].name == "Ag") && (Vect[j].name == "V")))
                                {
                                    Er = Er + ((A1/r0)*(rij-r0)+A0)*exp(-p*(rij/r0-1)
    );

                                    Eb = Eb + ksi*ksi*exp(-2*q*(rij/r0-1));
                                }
                            }
                        }
                    }

        Eb = -sqrt(Eb);
        Energy += Er + Eb;
    }

    return Energy;
}

double Energy_AB_parallel(vector <Atom>&Vect, double a0, double d_trans[9], double A1
    , double A0, double ksi, double p, double q, double r0, double x[6], int i)
{
    double Energy = 0;

    double rij_min = 10 * a0;

    double Er = 0, Eb = 0;

    double a0_x = a0 * d_trans[0] + a0 * d_trans[1] + a0 * d_trans[2];
    double a0_y = a0 * d_trans[3] + a0 * d_trans[4] + a0 * d_trans[5];
    double a0_z = a0 * d_trans[6] + a0 * d_trans[7] + a0 * d_trans[8];

    double cutoff = 1.7 * fmax(fmax(a0_x, a0_y), a0_z);

    for (int j=0; j<Vect.size(); j++)
    {
        for (int dx=-1; dx<2; dx++)
            for (int dy=-1; dy<2; dy++)
                for (int dz=-1; dz<2; dz++)
                {
                    if (i != j || dx !=0 || dy !=0 || dz !=0 )
                    {
```

```
                        double tmp_x = Vect[j].x + a0*3*dx;
                        double tmp_y = Vect[j].y + a0*3*dy;
                        double tmp_z = Vect[j].z + a0*3*dz;

                        tmp_x = tmp_x * d_trans[0] + tmp_y * d_trans[1] + tmp_z *
    d_trans[2];
                        tmp_y = tmp_x * d_trans[3] + tmp_y * d_trans[4] + tmp_z *
    d_trans[5];
                        tmp_z = tmp_x * d_trans[6] + tmp_y * d_trans[7] + tmp_z *
    d_trans[8];

                        tmp_x -= Vect[i].x * d_trans[0] + Vect[i].y * d_trans[1] +
    Vect[i].z * d_trans[2];
                        tmp_y -= Vect[i].x * d_trans[3] + Vect[i].y * d_trans[4] +
    Vect[i].z * d_trans[5];
                        tmp_z -= Vect[i].x * d_trans[6] + Vect[i].y * d_trans[7] +
    Vect[i].z * d_trans[8];

                        double rij = sqrt(tmp_x*tmp_x + tmp_y*tmp_y + tmp_z*tmp_z);

                        if (rij <= cutoff)
                        {
                            if((Vect[i].name == "Ag") && (Vect[j].name == "Ag"))
                            {
                                Er = Er + ((x[0]/x[5])*(rij-x[5])+x[1])*exp(-x[3]*(
    rij/x[5]-1));

                                Eb = Eb + x[2]*x[2]*exp(-2*x[4]*(rij/x[5]-1));
                            }
                            else if(((Vect[i].name == "V") && (Vect[j].name == "Ag"))
     || ((Vect[i].name == "Ag") && (Vect[j].name == "V")))
                            {
                                Er = Er + ((A1/r0)*(rij-r0)+A0)*exp(-p*(rij/r0-1));

                                Eb = Eb + ksi*ksi*exp(-2*q*(rij/r0-1));
                            }
                        }
                    }
                }
    }

    Eb = -sqrt(Eb);
    Energy += Er + Eb;

    return Energy;
}

double Energy_last(vector <Atom>&Vect, double a0, double d_trans[9], double A1,
    double A0, double ksi, double p, double q, double r0, double x[12])
```

```
{
    double Energy = 0;

    for (int i=0; i<Vect.size(); i++)
    {
        double Er = 0, Eb = 0;

        double a0_x = a0 * d_trans[0] + a0 * d_trans[1] + a0 * d_trans[2];
        double a0_y = a0 * d_trans[3] + a0 * d_trans[4] + a0 * d_trans[5];
        double a0_z = a0 * d_trans[6] + a0 * d_trans[7] + a0 * d_trans[8];

        double cutoff = 1.7 * fmax(fmax(a0_x, a0_y), a0_z);

        for (int j=0; j<Vect.size(); j++)
        {
            for (int dx=-1; dx<2; dx++)
                for (int dy=-1; dy<2; dy++)
                    for (int dz=0; dz<1; dz++)
                    {
                        if (i != j || dx !=0 || dy !=0 || dz !=0 )
                        {
                            double tmp_x = Vect[j].x + a0*3*dx;
                            double tmp_y = Vect[j].y + a0*3*dy;
                            double tmp_z = Vect[j].z + a0*3*dz;

                            tmp_x = tmp_x * d_trans[0] + tmp_y * d_trans[1] + tmp_z *
    d_trans[2];
                            tmp_y = tmp_x * d_trans[3] + tmp_y * d_trans[4] + tmp_z *
    d_trans[5];
                            tmp_z = tmp_x * d_trans[6] + tmp_y * d_trans[7] + tmp_z *
    d_trans[8];

                            tmp_x -= Vect[i].x * d_trans[0] + Vect[i].y * d_trans[1]
   + Vect[i].z * d_trans[2];
                            tmp_y -= Vect[i].x * d_trans[3] + Vect[i].y * d_trans[4]
   + Vect[i].z * d_trans[5];
                            tmp_z -= Vect[i].x * d_trans[6] + Vect[i].y * d_trans[7]
   + Vect[i].z * d_trans[8];

                            double rij = sqrt(tmp_x*tmp_x + tmp_y*tmp_y + tmp_z*tmp_z
    );

                            if (rij <= cutoff)
                            {
                                if((Vect[i].name == "Ag") && (Vect[j].name == "Ag"))
                                {
                                    Er = Er + ((x[0]/x[5])*(rij-x[5])+x[1])*exp(-x
    [3]*(rij/x[5]-1));
```

```
                                    Eb = Eb + x[2]*x[2]*exp(-2*x[4]*(rij/x[5]-1));
                                }
                                else if(((Vect[i].name == "V") && (Vect[j].name == "
    Ag")) || ((Vect[i].name == "Ag") && (Vect[j].name == "V")))
                                {
                                    Er = Er + ((x[6]/x[11])*(rij-x[11])+x[7])*exp(-x
    [9]*(rij/x[11]-1));

                                    Eb = Eb + x[8]*x[8]*exp(-2*x[10]*(rij/x[11]-1));
                                }
                                else if((Vect[i].name == "V") && (Vect[j].name == "V"
    ))
                                {
                                    Er = Er + ((A1/r0)*(rij-r0)+A0)*exp(-p*(rij/r0-1)
    );

                                    Eb = Eb + ksi*ksi*exp(-2*q*(rij/r0-1));
                                }
                            }
                        }
                    }
                }

                Eb = -sqrt(Eb);
                Energy += Er + Eb;
            }

            return Energy;
}

double Energy_last_parallel(vector <Atom>&Vect, double a0, double d_trans[9], double
    A1, double A0, double ksi, double p, double q, double r0, double x[12], int i)
{
            double Energy = 0;

            double Er = 0, Eb = 0;

            double a0_x = a0 * d_trans[0] + a0 * d_trans[1] + a0 * d_trans[2];
            double a0_y = a0 * d_trans[3] + a0 * d_trans[4] + a0 * d_trans[5];
            double a0_z = a0 * d_trans[6] + a0 * d_trans[7] + a0 * d_trans[8];

            double cutoff = 1.7 * fmax(fmax(a0_x, a0_y), a0_z);

            for (int j=0; j<Vect.size(); j++)
            {
                for (int dx=-1; dx<2; dx++)
                    for (int dy=-1; dy<2; dy++)
                        for (int dz=0; dz<1; dz++)
                        {
```

```
                if (i != j || dx !=0 || dy !=0 || dz !=0 )
                {
                    double tmp_x = Vect[j].x + a0*3*dx;
                    double tmp_y = Vect[j].y + a0*3*dy;
                    double tmp_z = Vect[j].z + a0*3*dz;

                    tmp_x = tmp_x * d_trans[0] + tmp_y * d_trans[1] + tmp_z *
d_trans[2];
                    tmp_y = tmp_x * d_trans[3] + tmp_y * d_trans[4] + tmp_z *
d_trans[5];
                    tmp_z = tmp_x * d_trans[6] + tmp_y * d_trans[7] + tmp_z *
d_trans[8];

                    tmp_x -= Vect[i].x * d_trans[0] + Vect[i].y * d_trans[1] +
Vect[i].z * d_trans[2];
                    tmp_y -= Vect[i].x * d_trans[3] + Vect[i].y * d_trans[4] +
Vect[i].z * d_trans[5];
                    tmp_z -= Vect[i].x * d_trans[6] + Vect[i].y * d_trans[7] +
Vect[i].z * d_trans[8];

                    double rij = sqrt(tmp_x*tmp_x + tmp_y*tmp_y + tmp_z*tmp_z);

                    if (rij <= cutoff)
                    {
                        if((Vect[i].name == "Ag") && (Vect[j].name == "Ag"))
                        {
                            Er = Er + ((x[0]/x[5])*(rij-x[5])+x[1])*exp(-x[3]*(
rij/x[5]-1));

                            Eb = Eb + x[2]*x[2]*exp(-2*x[4]*(rij/x[5]-1));
                        }
                        else if(((Vect[i].name == "V") && (Vect[j].name == "Ag"))
 || ((Vect[i].name == "Ag") && (Vect[j].name == "V")))
                        {
                            Er = Er + ((x[6]/x[11])*(rij-x[11])+x[7])*exp(-x[9]*(
rij/x[11]-1));

                            Eb = Eb + x[8]*x[8]*exp(-2*x[10]*(rij/x[11]-1));
                        }
                        else if((Vect[i].name == "V") && (Vect[j].name == "V"))
                        {
                            Er = Er + ((A1/r0)*(rij-r0)+A0)*exp(-p*(rij/r0-1));

                            Eb = Eb + ksi*ksi*exp(-2*q*(rij/r0-1));
                        }
                    }
                }
            }
}
```

```
    Eb = -sqrt(Eb);
    Energy += Er + Eb;

    return Energy;
}

double EnergyFinal(double rij, double x[6])
{
    double Energy, Er, Eb;

    Er = ((x[0]/x[5])*(rij-x[5])+x[1])*exp(-x[3]*(rij/x[5]-1));

    Eb = x[2]*x[2]*exp(-2*x[4]*(rij/x[5]-1));

    Eb = -sqrt(Eb);

    Energy = Er + Eb;

    return Energy;
}

double EnergySol(double a0, double d_trans[9], double A1, double A0, double ksi,
    double p, double q, double r0, double x[6], double E_coh)
{
    vector <Atom> Vect = GCK("Ag", a0);

    Vect[0].changeAtom("V");

    //double E_AB = Energy_AB(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

    double Energy_per_thread[4] = {0.0, 0.0, 0.0, 0.0};
    omp_set_dynamic(0);

    #pragma omp parallel num_threads(4)
    {
        int threads = omp_get_num_threads();
        int thread_num = omp_get_thread_num();

        int k = Vect.size();

        int left_bound = (k*thread_num) / double(threads);
        int right_bound = (k*(thread_num + 1)) / double(threads);

        if(thread_num == threads - 1)
        {
            right_bound = k;
        }
```

```
        for(int i=left_bound; i<right_bound; i++)
        {
            Energy_per_thread[thread_num] += Energy_AB_parallel(Vect, a0, d_trans, A1
    , A0, ksi, p, q, r0, x, i);
        }
    }

    double E_AB = 0;

    for(int i=0; i<4; i++)
    {
        E_AB += Energy_per_thread[i];
    }

    double E_coh_A = -5.31;

    double E_coh_B = E_coh;

    double E_B = E_coh_B * Vect.size();

    Vect[0].changeAtom("Ag");

    return E_AB - E_B - E_coh_A + E_coh_B;
}

double EnergyIn(double a0, double d_trans[9], double A1, double A0, double ksi,
    double p, double q, double r0, double x[12])
{
    int N = 2;

    vector <Atom> Vect = GCK("Ag", a0);

    /*Vect[81].changeAtom("V");
    Vect[83].changeAtom("V");

    double E_dim_surf = Energy_last(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

    Vect[81].changeAtom("Ag");
    Vect[83].changeAtom("Ag");

    double E_surf = Energy_last(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

    Vect[81].changeAtom("V");

    double E_adatom_surf = Energy_last(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

    Vect[81].changeAtom("Ag");*/
```

```
double Energy_per_thread[4][3] = {{0.0, 0.0, 0.0}, {0.0, 0.0, 0.0}, {0.0, 0.0,
0.0}, {0.0, 0.0, 0.0}};
omp_set_dynamic(0);

Vect[81].changeAtom("V");
Vect[83].changeAtom("V");

#pragma omp parallel num_threads(4)
{
    int threads = omp_get_num_threads();
    int thread_num = omp_get_thread_num();

    int k = Vect.size();

    int left_bound = (k*thread_num) / double(threads);
    int right_bound = (k*(thread_num + 1)) / double(threads);

    if(thread_num == threads - 1)
    {
        right_bound = k;
    }

    for(int i=left_bound; i<right_bound; i++)
    {
        Energy_per_thread[thread_num][0] += Energy_last_parallel(Vect, a0,
d_trans, A1, A0, ksi, p, q, r0, x, i);
    }
}

Vect[81].changeAtom("Ag");
Vect[83].changeAtom("Ag");

#pragma omp parallel num_threads(4)
{
    int threads = omp_get_num_threads();
    int thread_num = omp_get_thread_num();

    int k = Vect.size();

    int left_bound = (k*thread_num) / double(threads);
    int right_bound = (k*(thread_num + 1)) / double(threads);

    if(thread_num == threads - 1)
    {
        right_bound = k;
    }

    for(int i=left_bound; i<right_bound; i++)
    {
```

```
            Energy_per_thread[thread_num][1] += Energy_last_parallel(Vect, a0,
    d_trans, A1, A0, ksi, p, q, r0, x, i);
        }
    }

    Vect[81].changeAtom("V");

    #pragma omp parallel num_threads(4)
    {
        int threads = omp_get_num_threads();
        int thread_num = omp_get_thread_num();

        int k = Vect.size();

        int left_bound = (k*thread_num) / double(threads);
        int right_bound = (k*(thread_num + 1)) / double(threads);

        if(thread_num == threads - 1)
        {
            right_bound = k;
        }

        for(int i=left_bound; i<right_bound; i++)
        {
            Energy_per_thread[thread_num][2] += Energy_last_parallel(Vect, a0,
    d_trans, A1, A0, ksi, p, q, r0, x, i);
        }
    }

    Vect[81].changeAtom("Ag");

    double E_dim_surf = 0, E_surf = 0, E_adatom_surf = 0;

    for(int i=0; i<4; i++)
    {
        E_dim_surf += Energy_per_thread[i][0];
        E_surf += Energy_per_thread[i][1];
        E_adatom_surf += Energy_per_thread[i][2];
    }

    return E_dim_surf - E_surf - N * (E_adatom_surf - E_surf);
}

double EnergyOn(double a0, double d_trans[9], double A1, double A0, double ksi,
    double p, double q, double r0, double x[12])
{
    int N = 2;

    vector <Atom> Vect = GCK("Ag", a0);
```

```
/*Vect.push_back(Atom(3*a0-1.5*a0, 0.5*a0, 3*a0, "V"));
Vect.push_back(Atom(2*a0, 0.0, 3*a0, "V"));

double E_dim_surf = Energy_last(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

Vect.pop_back();
Vect.pop_back();

double E_surf = Energy_last(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

Vect.push_back(Atom(3*a0-1.5*a0, 0.5*a0, 3*a0, "V"));

double E_adatom_surf = Energy_last(Vect, a0, d_trans, A1, A0, ksi, p, q, r0, x);

Vect.pop_back();*/

double Energy_per_thread[4][3] = {{0.0, 0.0, 0.0}, {0.0, 0.0, 0.0}, {0.0, 0.0,
0.0}, {0.0, 0.0, 0.0}};
omp_set_dynamic(0);

Vect.push_back(Atom(3*a0-1.5*a0, 0.5*a0, 3*a0, "V"));
Vect.push_back(Atom(2*a0, 0.0, 3*a0, "V"));

#pragma omp parallel num_threads(4)
{
    int threads = omp_get_num_threads();
    int thread_num = omp_get_thread_num();

    int k = Vect.size();

    int left_bound = (k*thread_num) / double(threads);
    int right_bound = (k*(thread_num + 1)) / double(threads);

    if(thread_num == threads - 1)
    {
        right_bound = k;
    }

    for(int i=left_bound; i<right_bound; i++)
    {
        Energy_per_thread[thread_num][0] += Energy_last_parallel(Vect, a0,
d_trans, A1, A0, ksi, p, q, r0, x, i);
    }
}

Vect.pop_back();
Vect.pop_back();
```

```
#pragma omp parallel num_threads(4)
{
    int threads = omp_get_num_threads();
    int thread_num = omp_get_thread_num();

    int k = Vect.size();

    int left_bound = (k*thread_num) / double(threads);
    int right_bound = (k*(thread_num + 1)) / double(threads);

    if(thread_num == threads - 1)
    {
        right_bound = k;
    }

    for(int i=left_bound; i<right_bound; i++)
    {
        Energy_per_thread[thread_num][1] += Energy_last_parallel(Vect, a0,
d_trans, A1, A0, ksi, p, q, r0, x, i);
    }
}

Vect.push_back(Atom(3*a0-1.5*a0, 0.5*a0, 3*a0, "V"));

#pragma omp parallel num_threads(4)
{
    int threads = omp_get_num_threads();
    int thread_num = omp_get_thread_num();

    int k = Vect.size();

    int left_bound = (k*thread_num) / double(threads);
    int right_bound = (k*(thread_num + 1)) / double(threads);

    if(thread_num == threads - 1)
    {
        right_bound = k;
    }

    for(int i=left_bound; i<right_bound; i++)
    {
        Energy_per_thread[thread_num][2] += Energy_last_parallel(Vect, a0,
d_trans, A1, A0, ksi, p, q, r0, x, i);
    }
}

Vect.pop_back();

double E_dim_surf = 0, E_surf = 0, E_adatom_surf = 0;
```

```
    for(int i=0; i<4; i++)
    {
        E_dim_surf += Energy_per_thread[i][0];
        E_surf += Energy_per_thread[i][1];
        E_adatom_surf += Energy_per_thread[i][2];
    }

    return E_dim_surf - E_surf - N * (E_adatom_surf - E_surf);
}

void Parameters(double &E, double a0, double &B, double &c11, double &c12, double &
    c44, double A1, double A0, double ksi, double p, double q, double r0)
{
    double v0 = a0*a0*a0/4, const_p = 0.8018993929636421, alpha = 0.001, alpha2 =
    0.000001;

    double d_b_plus[9]={1+alpha, 0,0,0, 1+alpha, 0,0,0, 1+alpha};

    double d_b_minus[9]={1-alpha, 0,0,0, 1-alpha, 0,0,0, 1-alpha};

    double d_c11_plus[9]={1+alpha, 0,0,0, 1+alpha, 0,0,0, 1};

    double d_c11_minus[9]={1-alpha, 0,0,0, 1-alpha, 0,0,0, 1};

    double d_c12_plus[9]={1+alpha, 0,0,0, 1-alpha, 0,0,0, 1};

    double d_c12_minus[9]={1-alpha, 0,0,0, 1+alpha, 0,0,0, 1};

    double d_c44_plus[9]={1, alpha, 0, alpha, 1, 0, 0,0, 1/(1-alpha2)};

    double d_c44_minus[9]={1, -alpha, 0, -alpha, 1, 0, 0,0, 1/(1-alpha2)};

    vector <Atom> Vect_p = GCK("Ag", a0);

    vector <Atom> Vect_B_plus = GCK("Ag", a0);

    vector <Atom> Vect_B_minus = GCK("Ag", a0);

    vector <Atom> Vect_c11_plus = GCK("Ag", a0);

    vector <Atom> Vect_c11_minus = GCK("Ag", a0);

    vector <Atom> Vect_c12_plus = GCK("Ag", a0);

    vector <Atom> Vect_c12_minus = GCK("Ag", a0);

    vector <Atom> Vect_c44_plus = GCK("Ag", a0);
```

```
vector <Atom> Vect_c44_minus = GCK("Ag", a0);

/*double E_p = Energy(Vect_p, a0, d_edin, A1, A0, ksi, p, q, r0) / Vect_p.size();

double E_B_plus = Energy(Vect_B_plus, a0, d_b_plus, A1, A0, ksi, p, q, r0) /
Vect_B_plus.size();

double E_B_minus = Energy(Vect_B_minus, a0, d_b_minus, A1, A0, ksi, p, q, r0) /
Vect_B_minus.size();

double E_c11_plus = Energy(Vect_c11_plus, a0, d_c11_plus, A1, A0, ksi, p, q, r0)
/ Vect_c11_plus.size();

double E_c11_minus = Energy(Vect_c11_minus, a0, d_c11_minus, A1, A0, ksi, p, q,
r0) / Vect_c11_minus.size();

double E_c12_plus = Energy(Vect_c12_plus, a0, d_c12_plus, A1, A0, ksi, p, q, r0)
/ Vect_c12_plus.size();

double E_c12_minus = Energy(Vect_c12_minus, a0, d_c12_minus, A1, A0, ksi, p, q,
r0) / Vect_c12_minus.size();

double E_c44_plus = Energy(Vect_c44_plus, a0, d_c44_plus, A1, A0, ksi, p, q, r0)
/ Vect_c44_plus.size();

double E_c44_minus = Energy(Vect_c44_minus, a0, d_c44_minus, A1, A0, ksi, p, q,
r0) / Vect_c44_minus.size();*/

double Energy_per_thread[4][9] = {{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0},
{0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}, {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
 0.0, 0.0}, {0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0}};
omp_set_dynamic(0);

#pragma omp parallel num_threads(4)
{
    int threads = omp_get_num_threads();
    int thread_num = omp_get_thread_num();

    int k = Vect_p.size();

    int left_bound = (k*thread_num) / double(threads);
    int right_bound = (k*(thread_num + 1)) / double(threads);

    if(thread_num == threads - 1)
    {
        right_bound = k;
    }

    for(int i=left_bound; i<right_bound; i++)
```

```
    {
        Energy_per_thread[thread_num][0] += Energy_parallel(Vect_p, a0, d_edin,
A1, A0, ksi, p, q, r0, i) / Vect_p.size();
        Energy_per_thread[thread_num][1] += Energy_parallel(Vect_B_plus, a0,
d_b_plus, A1, A0, ksi, p, q, r0, i) / Vect_B_plus.size();
        Energy_per_thread[thread_num][2] += Energy_parallel(Vect_B_minus, a0,
d_b_minus, A1, A0, ksi, p, q, r0, i) / Vect_B_minus.size();
        Energy_per_thread[thread_num][3] += Energy_parallel(Vect_c11_plus, a0,
d_c11_plus, A1, A0, ksi, p, q, r0, i) / Vect_c11_plus.size();
        Energy_per_thread[thread_num][4] += Energy_parallel(Vect_c11_minus, a0,
d_c11_minus, A1, A0, ksi, p, q, r0, i) / Vect_c11_minus.size();
        Energy_per_thread[thread_num][5] += Energy_parallel(Vect_c12_plus, a0,
d_c12_plus, A1, A0, ksi, p, q, r0, i) / Vect_c12_plus.size();
        Energy_per_thread[thread_num][6] += Energy_parallel(Vect_c12_minus, a0,
d_c12_minus, A1, A0, ksi, p, q, r0, i) / Vect_c12_minus.size();
        Energy_per_thread[thread_num][7] += Energy_parallel(Vect_c44_plus, a0,
d_c44_plus, A1, A0, ksi, p, q, r0, i) / Vect_c44_plus.size();
        Energy_per_thread[thread_num][8] += Energy_parallel(Vect_c44_minus, a0,
d_c44_minus, A1, A0, ksi, p, q, r0, i) / Vect_c44_minus.size();
    }
}

double E_p = 0, E_B_plus = 0, E_B_minus = 0, E_c11_plus = 0, E_c11_minus = 0,
E_c12_plus = 0, E_c12_minus = 0, E_c44_plus = 0, E_c44_minus = 0;

for(int i=0; i<4; i++)
{
    E_p += Energy_per_thread[i][0];
    E_B_plus += Energy_per_thread[i][1];
    E_B_minus += Energy_per_thread[i][2];
    E_c11_plus += Energy_per_thread[i][3];
    E_c11_minus += Energy_per_thread[i][4];
    E_c12_plus += Energy_per_thread[i][5];
    E_c12_minus += Energy_per_thread[i][6];
    E_c44_plus += Energy_per_thread[i][7];
    E_c44_minus += Energy_per_thread[i][8];
}

E = E_p;

double d2_E_B = (E_B_plus - 2*E_p +E_B_minus)/alpha2;

double d2_E_c11 = (E_c11_plus - 2*E_p +E_c11_minus)/alpha2;

double d2_E_c12 =(E_c12_plus - 2*E_p +E_c12_minus)/alpha2;

double d2_E_c44 =(E_c44_plus - 2*E_p +E_c44_minus)/alpha2;

B = (d2_E_B*2*const_p)/(9.0*v0);
```

```
    c11 = ((d2_E_c11 + d2_E_c12)*const_p)/(2.0*v0);

    c12 = ((d2_E_c11 - d2_E_c12)*const_p)/(2.0*v0);

    c44 = (d2_E_c44*const_p)/(2*v0);
}

double f(double x[], int j, int n)
{
    double E, B, c11, c12, c44;

    Parameters(E, a0_global, B, c11, c12, c44, x[0*(n+1)+j], x[1*(n+1)+j], x[2*(n+1)+
    j], x[3*(n+1)+j], x[4*(n+1)+j], x[5*(n+1)+j]);

    double sum = 0;

    sum += (a0_global - true_parameters[0])*(a0_global - true_parameters[0]) / (
    true_parameters[0]*true_parameters[0]);
    sum += (E - true_parameters[1])*(E - true_parameters[1]) / (true_parameters[1]*
    true_parameters[1]);
    sum += (B - true_parameters[2])*(B - true_parameters[2]) / (true_parameters[2]*
    true_parameters[2]);
    sum += (c11 - true_parameters[3])*(c11 - true_parameters[3]) / (true_parameters
    [3]*true_parameters[3]);
    sum += (c12 - true_parameters[4])*(c12 - true_parameters[4]) / (true_parameters
    [4]*true_parameters[4]);
    sum += (c44 - true_parameters[5])*(c44 - true_parameters[5]) / (true_parameters
    [5]*true_parameters[5]);

    sum = sqrt(sum/6);

        return sum;
}

double f1(double x[])
{
    double E, B, c11, c12, c44;

    Parameters(E, a0_global, B, c11, c12, c44, x[0], x[1], x[2], x[3], x[4], x[5]);

    double sum = 0;

    sum += (a0_global - true_parameters[0])*(a0_global - true_parameters[0]) / (
    true_parameters[0]*true_parameters[0]);
    sum += (E - true_parameters[1])*(E - true_parameters[1]) / (true_parameters[1]*
    true_parameters[1]);
    sum += (B - true_parameters[2])*(B - true_parameters[2]) / (true_parameters[2]*
    true_parameters[2]);
```

```
    sum += (c11 - true_parameters[3])*(c11 - true_parameters[3]) / (true_parameters
    [3]*true_parameters[3]);
    sum += (c12 - true_parameters[4])*(c12 - true_parameters[4]) / (true_parameters
    [4]*true_parameters[4]);
    sum += (c44 - true_parameters[5])*(c44 - true_parameters[5]) / (true_parameters
    [5]*true_parameters[5]);

    sum = sqrt(sum/6);

        return sum;
}

double f_2(double x[], int j, int n)
{
    double sum = 0;

    double E_sol = EnergySol(a0_global, d_edin, x[0*(n+1)+j], x[1*(n+1)+j], x[2*(n+1)
    +j], x[3*(n+1)+j], x[4*(n+1)+j], x[5*(n+1)+j], Array, E_cohesion);

    sum += (E_sol - true_parameter)*(E_sol - true_parameter) / (true_parameter*
    true_parameter);

    sum = sqrt(sum);

        return sum;
}

double f1_2(double x[])
{
    double sum = 0;

    double E_sol = EnergySol(a0_global, d_edin, x[0], x[1], x[2], x[3], x[4], x[5],
    Array, E_cohesion);

    sum += (E_sol - true_parameter)*(E_sol - true_parameter) / (true_parameter*
    true_parameter);

    sum = sqrt(sum);

        return sum;
}

double f_3(double x[], int j, int n)
{
    double sum = 0;

    double E_In = EnergyIn(a0_global, d_edin, x[0*(n+1)+j], x[1*(n+1)+j], x[2*(n+1)+j
    ], x[3*(n+1)+j], x[4*(n+1)+j], x[5*(n+1)+j], Array2);
```

```
    double E_On = EnergyOn(a0_global, d_edin, x[0*(n+1)+j], x[1*(n+1)+j], x[2*(n+1)+j
    ], x[3*(n+1)+j], x[4*(n+1)+j], x[5*(n+1)+j], Array2);

    sum += (E_In - true_params[0])*(E_In - true_params[0]) / (true_params[0]*
    true_params[0]);
    sum += (E_On - true_params[1])*(E_On - true_params[1]) / (true_params[1]*
    true_params[1]);

    sum = sqrt(sum/2);

        return sum;
}

double f1_3(double x[])
{
    double sum = 0;

    double E_In = EnergyIn(a0_global, d_edin, x[0], x[1], x[2], x[3], x[4], x[5],
    Array2);
    double E_On = EnergyOn(a0_global, d_edin, x[0], x[1], x[2], x[3], x[4], x[5],
    Array2);

    sum += (E_In - true_params[0])*(E_In - true_params[0]) / (true_params[0]*
    true_params[0]);
    sum += (E_On - true_params[1])*(E_On - true_params[1]) / (true_params[1]*
    true_params[1]);

    sum = sqrt(sum/2);

        return sum;
}

double* makeSimplex(double x[], int n, double L)
{
    double r1, r2;

    double* S = new double[n*(n+1)];

    r1 = L * ((n - 1.0 + sqrt(n + 1.0)) / (n * sqrt(2.0)));
    r2 = L * ((sqrt(n + 1.0) - 1.0) / (n * sqrt(2.0)));

    for(int i=0; i<n; i++)
    {
        S[i*(n+1)+0] = x[i];
    }

    for(int j=1; j<n+1; j++)
    {
        for(int i=0; i<n; i++)
```

```
        {
            if(i == j-1)
            {
                S[i*(n+1)+j] = x[i] + r1;
            }
            else
            {
                S[i*(n+1)+j] = x[i] + r2;
            }
        }
    }

    return S;
}

double* center_of_gravity(double S[], int k, int n)
{
    double* center_of_gravity = new double[n];

    for(int i=0; i<n; i++)
    {
        center_of_gravity[i] = 0.0;
    }

    for(int i=0; i<n; i++)
    {
        for(int j=0; j<n+1; j++)
        {
            if(j != k)
            {
                center_of_gravity[i] += S[i*(n+1)+j];
            }
        }

        center_of_gravity[i] = center_of_gravity[i]/n;
    }

    return center_of_gravity;
}

double reflection(double xc[], double xh[], int k, double alpha, int n, int i)
{
    double xr;

    xr = (1.0 + alpha) * xc[i] - alpha * xh[i];

    return xr;
}
```

```
double stretch(double xc[], double xr[], int k, double gamma, int n, int i)
{
    double xe;

    xe = (1.0 - gamma) * xc[i] + gamma * xr[i];

    return xe;
}

double compress(double xc[], double xh[], int k, double beta, int n, int i)
{
    double xs;

    xs = (1.0 - beta) * xc[i] + beta * xh[i];

    return xs;
}

bool notStopYet(double (*op1)(double*, int, int), double S[], double eps, int n)
{
    bool notStopYet = false;

    double F[n+1];

    for(int j=0; j<n+1; j++)
    {
        F[j] = op1(S, j, n);
    }

    double min_f = F[0];
    ind_min_global = 0;

    for(int j=0; j<n+1; j++)
    {
        if(F[j] < min_f)
        {
            min_f = F[j];
            ind_min_global = j;
        }
    }

    if(min_f > eps)
        notStopYet = true;

    cout << min_f << endl;

    return notStopYet;
}
```

```
double* nelMead(double (*op1)(double*, int, int), double (*op2)(double*), double x[],
    int n, double L, double eps, double alpha, double beta, double gamma, double
   sigma)
{
    double *S = makeSimplex(x, n, L);

    double *xc = new double[n];

    double F[n+1], Fr, Fe, Fs;

    double max_f, min_f, max2_f;

    double xh[n], xl[n], xr[n], xe[n], xs[n];

    int ind_max = 0, ind_min;

    bool flag;

    for(int j=0; j<n+1; j++)
    {
        F[j] = op1(S, j, n);
    }

    while(notStopYet(op1, S, eps, n))
    {
        for(int i=0; i<n; i++)
        {
            for(int j=0; j<n+1; j++)
            {
                if(S[i*(n+1)+j] < Params_bounds[i][0])
                {
                    S[i*(n+1)+j] = Params_bounds[i][0]*1.01;
                }
                else if(S[i*(n+1)+j] > Params_bounds[i][1])
                {
                    S[i*(n+1)+j] = Params_bounds[i][1]*0.99;
                }
            }
        }

        for(int j=0; j<n+1; j++)
        {
            F[j] = op1(S, j, n);
        }

        max_f = F[0];
        max2_f = F[0];
        min_f = F[0];
        ind_max = 0;
```

```
ind_min = 0;

for(int j=0; j<n+1; j++)
{
    if(F[j] <= min_f)
    {
        min_f = F[j];
        ind_min = j;
    }

    if(F[j] >= max_f)
    {
        max2_f = max_f;
        max_f = F[j];
        ind_max = j;
    }
    else if(F[j] >= max2_f)
    {
        max2_f = F[j];
    }
}

for(int i=0; i<n; i++)
{
    xh[i] = S[i*(n+1)+ind_max];
    xl[i] = S[i*(n+1)+ind_min];
}

xc = center_of_gravity(S, ind_max, n);

for(int i=0; i<n; i++)
{
    xr[i] = reflection(xc, xh, ind_max, alpha, n, i);
}

Fr = op2(xr);

if(Fr < min_f)
{
    for(int i=0; i<n; i++)
    {
        xe[i] = stretch(xc, xr, ind_max, gamma, n, i);
    }

    Fe = op2(xe);

    if(Fe < Fr)
    {
        for(int i=0; i<n; i++)
```

```
            {
                S[i*(n+1)+ind_max] = xe[i];
            }
            F[ind_max] = Fe;
        }
        else
        {
            for(int i=0; i<n; i++)
            {
                S[i*(n+1)+ind_max] = xr[i];
            }
            F[ind_max] = Fr;
        }
    }

    if((Fr >= min_f) && (Fr < max2_f))
    {
        for(int i=0; i<n; i++)
        {
            S[i*(n+1)+ind_max] = xr[i];
        }
        F[ind_max] = Fr;
    }

    flag = false;

    if((Fr >= max2_f) && (Fr < max_f))
    {
        flag = true;

        for(int i=0; i<n; i++)
        {
            S[i*(n+1)+ind_max] = xr[i];
        }
        F[ind_max] = Fr;

        for(int i=0; i<n; i++)
        {
            xh[i] = xr[i];
        }
    }

    if(Fr >= max_f)
    {
        flag = true;
    }

    if(flag)
    {
```

```
            for(int i=0; i<n; i++)
            {
                xs[i] = compress(xc, xh, ind_max, beta, n, i);
            }
            Fs = op2(xs);

            if(Fs < max_f)
            {
                for(int i=0; i<n; i++)
                {
                    S[i*(n+1)+ind_max] = xs[i];
                }
                F[ind_max] = Fs;
            }
            else
            {
                for(int j=0; j<n+1; j++)
                {
                    for(int i=0; i<n; i++)
                    {
                        S[i*(n+1)+j] = xl[i] + sigma*(S[i*(n+1)+j] - xl[i]);
                    }
                }

                for(int j=0; j<n+1; j++)
                {
                    F[j] = op1(S, j, n);
                }
            }
        }
    }

    for(int j=0; j<n+1; j++)
    {
        F[j] = op1(S, j, n);
    }

    double min_f_global = F[0];
    ind_min_global = 0;

    for(int j=0; j<n+1; j++)
    {
        if(F[j] < min_f_global)
        {
            min_f_global = F[j];
            ind_min_global = j;
        }
    }
```

```
    double* Params = new double[n];

    for(int i=0; i<n; i++)
    {
        Params[i] = S[i*(n+1)+ind_min_global];
    }

    return Params;
}

double GetRandomNumber(double min, double max, int precision)
{
  double value;

  value = rand() % (int)pow(10, precision);

  value = min + (value / pow(10, precision)) * (max - min);

  return value;
}

int main()
{
    srand(4259707270);

    int n = 6;

    cout << endl;
    cout << "    A1: " << Params_bounds[0][0] << " - " << Params_bounds[0][1] << endl;
    cout << "    A0: " << Params_bounds[1][0] << " - " << Params_bounds[1][1] << endl;
    cout << "    ksi: " << Params_bounds[2][0] << " - " << Params_bounds[2][1] << endl
   ;
    cout << "    p: " << Params_bounds[3][0] << " - " << Params_bounds[3][1] << endl;
    cout << "    q: " << Params_bounds[4][0] << " - " << Params_bounds[4][1] << endl;
    cout << "    r0: " << Params_bounds[5][0] << " - " << Params_bounds[5][1] << endl;

    ofstream fout1;
    ofstream fout2;
    ofstream fout3;
    ofstream fout4;

        fout1.open("x.txt");
        fout2.open("y1.txt");
        fout3.open("y2.txt");
        fout4.open("y3.txt");

        double* Params1 = new double[n];
        double* Params2 = new double[n];
        double* Params3 = new double[n];
```

```
a0_global = rand()%11+2;

h = 1.0;

    double x[n];

    unsigned int start_time =  clock();

    x[0] = 0.0280413;
    x[1] = 0.110633;
    x[2] = 1.19174;
    x[3] = 11.1364;
    x[4] = 3.03011;
    x[5] = 2.86881;

    while(h > 1.0e-6)
{
    double a_left, a_right;
    double Energy_left, Energy_right, Energy_cent;

    a_left = a0_global - h;
    a_right = a0_global + h;

    vector <Atom> Vect_left = GCK("Ag", a_left);
    vector <Atom> Vect_right = GCK("Ag", a_right);
    vector <Atom> Vect_cent = GCK("Ag", a0_global);

    Energy_left = Energy(Vect_left, a_left, d_edin, x[0], x[1], x[2], x[3], x[4],
x[5]);
    Energy_right = Energy(Vect_right, a_right, d_edin, x[0], x[1], x[2], x[3], x
[4], x[5]);
    Energy_cent = Energy(Vect_cent, a0_global, d_edin, x[0], x[1], x[2], x[3], x
[4], x[5]);

    double Energy_min = min(min(Energy_left, Energy_right), Energy_cent);

    if(Energy_min == Energy_cent)
    {
        a0_global = a0_global;
        h /= 10;
    }
    else if(Energy_min == Energy_left)
    {
        a0_global = a_left;
    }
    else if(Energy_min == Energy_right)
    {
        a0_global = a_right;
```

```
    }
}

    x[0] = GetRandomNumber(0.0, 0.1, 6);
    x[1] = GetRandomNumber(0.0685, 0.1370, 6);
    x[2] = GetRandomNumber(0.7853, 1.570666, 6);
    x[3] = GetRandomNumber(7.2853, 14.5706, 6);
    x[4] = GetRandomNumber(2.0927, 4.1853, 6);
    x[5] = GetRandomNumber(1.9257, 3.8514, 6);

Params1 = nelMead(f, f1, x, n, 1.0, 0.003, 1.0, 0.5, 2.0, 0.5);

cout << endl;
cout << "    B-B:" << endl;
cout << "    A1 = " << Params1[0] << endl;
cout << "    A0 = " << Params1[1] << endl;
cout << "    ksi = " << Params1[2] << endl;
cout << "    p = " << Params1[3] << endl;
cout << "    q = " << Params1[4] << endl;
cout << "    r0 = " << Params1[5] << endl;

double E, a0, B, c11, c12, c44;

a0 = a0_global;

Parameters(E, a0_global, B, c11, c12, c44, Params1[0], Params1[1], Params1[2],
Params1[3], Params1[4], Params1[5]);

cout << endl;
cout << "    a0 = " << a0 << endl  << "    E_coh = " << E << endl << "    B = " << B
<< endl << "    c11 = " << c11 << endl << "    c12 = " << c12 << endl << "    c44 =
" << c44 << endl;
cout << endl;

for(int i=0; i<n; i++)
{
    Array[i] = Params1[i];
}
E_cohesion = E;

x[0] = GetRandomNumber(0.0, 0.1, 6);
    x[1] = GetRandomNumber(0.0685, 0.1370, 6);
    x[2] = GetRandomNumber(0.7853, 1.570666, 6);
    x[3] = GetRandomNumber(7.2853, 14.5706, 6);
    x[4] = GetRandomNumber(2.0927, 4.1853, 6);
    x[5] = GetRandomNumber(1.9257, 3.8514, 6);

Params2 = nelMead(f_2, f1_2, x, n, 1.0, 0.000001, 1.0, 0.5, 2.0, 0.5);
```

```
 cout << "    A-B:" << endl;
 cout << "    A1 = " << Params2[0] << endl;
 cout << "    A0 = " << Params2[1] << endl;
 cout << "    ksi = " << Params2[2] << endl;
 cout << "    p = " << Params2[3] << endl;
 cout << "    q = " << Params2[4] << endl;
 cout << "    r0 = " << Params2[5] << endl;

 cout << endl;
 cout << "    E_sol = " << EnergySol(a0_global, d_edin, Params2[0], Params2[1],
Params2[2], Params2[3], Params2[4], Params2[5], Array, E_cohesion) << endl;
 cout << endl;

 for(int i=0; i<n; i++)
 {
     Array2[i] = Params1[i];
 }

 for(int i=n; i<2*n; i++)
 {
     Array2[i] = Params2[i-n];
 }

 x[0] = GetRandomNumber(0.0, 0.1, 6);
     x[1] = GetRandomNumber(0.0685, 0.1370, 6);
     x[2] = GetRandomNumber(0.7853, 1.570666, 6);
     x[3] = GetRandomNumber(7.2853, 14.5706, 6);
     x[4] = GetRandomNumber(2.0927, 4.1853, 6);
     x[5] = GetRandomNumber(1.9257, 3.8514, 6);

 Params3 = nelMead(f_3, f1_3, x, n, 1.0, 0.000001, 1.0, 0.5, 2.0, 0.5);

 cout << "    A-A:" << endl;
 cout << "    A1 = " << Params3[0] << endl;
 cout << "    A0 = " << Params3[1] << endl;
 cout << "    ksi = " << Params3[2] << endl;
 cout << "    p = " << Params3[3] << endl;
 cout << "    q = " << Params3[4] << endl;
 cout << "    r0 = " << Params3[5] << endl;

 cout << endl;
 cout << "    E_in = " << EnergyIn(a0_global, d_edin, Params3[0], Params3[1],
Params3[2], Params3[3], Params3[4], Params3[5], Array2) << endl;
 cout << "    E_on = " << EnergyOn(a0_global, d_edin, Params3[0], Params3[1],
Params3[2], Params3[3], Params3[4], Params3[5], Array2) << endl;
 cout << endl;

 unsigned int end_time = clock();
```

```
    unsigned int search_time = end_time - start_time;
    search_time = search_time / 1000.0;

    cout << "   Execution time: " << search_time << " seconds" << endl;
    cout << endl;

    double rij = 0.25;

    while(rij <= 6.5)
    {
        fout1 << rij << endl;

        fout2 << EnergyFinal(rij, Params1) << endl;
        fout3 << EnergyFinal(rij, Params2) << endl;
        fout4 << EnergyFinal(rij, Params3) << endl;

        rij += 0.05;
    }

    fout1.close();
    fout2.close();
    fout3.close();
    fout4.close();
}
```