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

Трибрат Вадим Дмитриевич

Курсовая работа по теме:

“Параллельная обработка данных в научных исследованиях”

Преподаватель: Бажанов Д.И.

Москва, 2022

Постановка задачи

В рамках курсовой работы требуется произвести параметрическую идентификацию потенциалов межатомного взаимодействия (A-A, B-B, A-B) для системы A/B(001). Потенциалы описываются следующими уравнениями:

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

$$E_R^i = \sum_{j \neq i} (A_{\alpha\beta}^1 (r_{ij} - r_0^{\alpha\beta}) - A_{\alpha\beta}^0) exp(-p_{\alpha\beta} (\frac{r_{ij}}{r_0^{\alpha\beta}} - 1))$$

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

, где

- E – полная энергия системы
- E_R^i – энергия отталкивания
- E_B^i – энергия притяжения
- r_{ij} – расстояние между атомами i и j

Для вычисления параметров $A_{\alpha\beta}^1, A_{\alpha\beta}^0, \xi_{\alpha\beta}, p_{\alpha\beta}, q_{\alpha\beta}, r_0^{\alpha\beta}$ каждой из рассматриваемых систем необходимо ввести функцию отклонения рассчитываемых величин от табличных и минимизировать ее. Все расчеты полных энергий проводятся для кристаллической решетки размером 3x3x3 в единицах элементарной ячейки ГЦК структуры. Все расчеты статические, проводятся без релаксации атомных позиций. Атомы располагаются в узлах «идеальной» ГЦК решетки.

Для ускорения работы программы требуется применить технологию распараллеливания OpenMP или MPI.

Ход работы

ГЦК структура представляет собой решетку, в которой атомы расположены в вершинах и в центрах граней куба:

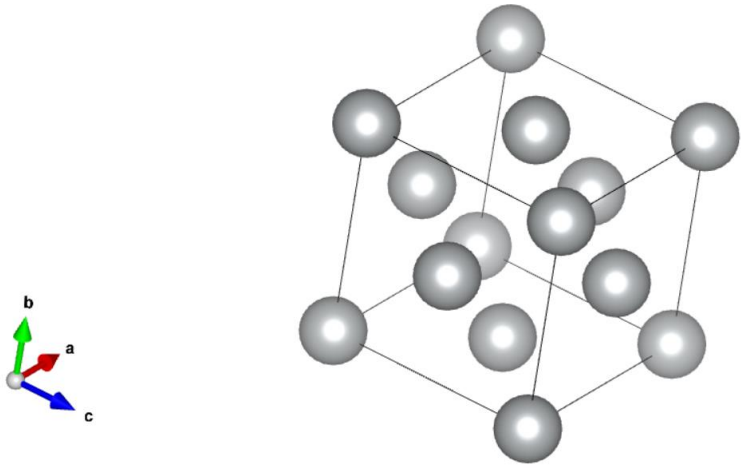


Рис. 1. Вид гранецентрированной кубической решетки.

Для оптимизации параметров потенциалов межатомного взаимодействия была выбрана следующая функция потерь:

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

, где

- n – количество оптимизируемых параметров
- $value_i$ – значение, вычисляемое при текущих параметрах
- $tableValue_i$ – табличные параметры

Для системы Co-Ag использовались следующие табличные параметры:

	Ag(001)
α	4.085
E_c	-2.960
B	1.08
C_{11}	1.32
C_{12}	0.97
C_{44}	0.51
	Co

E_{sol}	0.63
E_{dim}^{in}	-0.08
E_{dim}^{on}	-0.69

Описания данных и формулы расчётов:

- α – параметр решетки. Отвечает за объем элементарной ячейки.
- $E_c = \frac{E}{N}$ – когезионная энергия (полная энергия системы, деленная на количество атомов в ней).
- B - модуль всестороннего растяжения (сжатия). Если матрица деформации имеет вид $D = \begin{pmatrix} 1+\alpha & 0 & 0 \\ 0 & 1+\alpha & 0 \\ 0 & 0 & 1+\alpha \end{pmatrix}$, т.е. растяжение(сжатие) равномерное вдоль каждой из осей, то $B = V_0 \frac{d^2E}{dV^2} = \frac{2}{9V_0}\frac{d^2E_B}{d\alpha^2}$.
- C_{11} - константа упругости. Матрица деформации имеет вид $D = \begin{pmatrix} 1+\alpha & 0 & 0 \\ 0 & 1+\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$, т.е. равномерное растяжение(сжатие) вдоль осей XY, тогда $C_{11} = \frac{\frac{\partial^2 E_{C_{11}}}{\partial\alpha^2} + \frac{\partial^2 E_{C_{12}}}{\partial\alpha^2}}{2V_0}$.
- C_{12} - константа упругости. Матрица деформации имеет вид $D = \begin{pmatrix} 1+\alpha & 0 & 0 \\ 0 & 1-\alpha & 0 \\ 0 & 0 & 1 \end{pmatrix}$, т.е. растяжение вдоль оси X и сжатие вдоль Y, тогда $C_{12} = \frac{\frac{\partial^2 E_{C_{11}}}{\partial\alpha^2} - \frac{\partial^2 E_{C_{12}}}{\partial\alpha^2}}{2V_0}$.
- C_{44} - константа упругости. Матрица деформации имеет вид $D = \begin{pmatrix} 1 & \alpha & 0 \\ \alpha & 1 & 0 \\ 0 & 0 & \frac{1}{1-\alpha^2} \end{pmatrix}$, тогда $C_{44} = \frac{2}{9V_0}\frac{d^2E_{C_{44}}}{d\alpha^2}$..
- E_{sol} - энергия растворимости. Представляет собой энергию растворимости атома Co в кристалле Ag. Для этого надо заменить один из атомов решетки 3*3*3 на Co. Определяется следующей формулой $E_{sol} = E^{AB} - E^B - E_{coh}^A + E_{coh}^B$, где E^{AB} – полная энергия системы с одним примесным элементом, а E^B – полная энергия B-B системы.

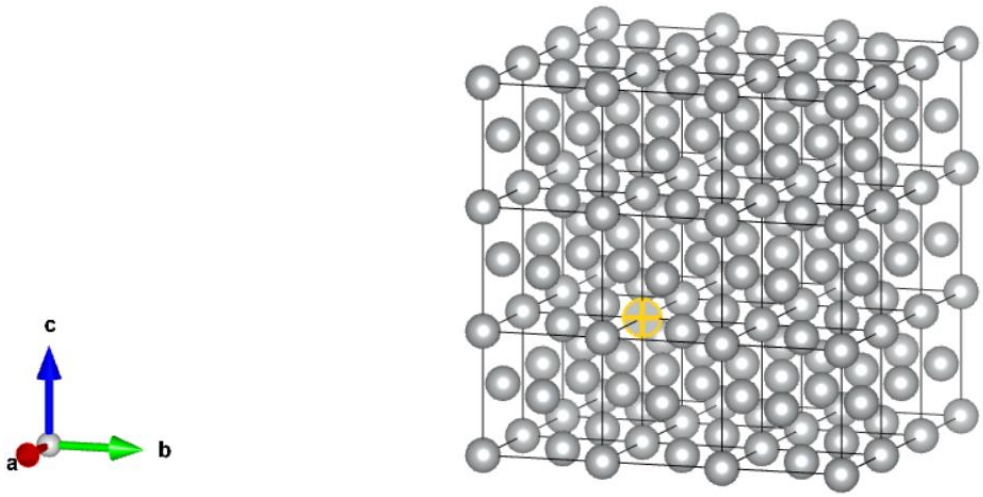


Рис. 2. ГЦК решетка с одним примесным элементом.

- E_{dim}^{in} – энергия связи димера в поверхностном слое. Для вычисления необходимо увеличить решетку до размеров 3*3*6 и убрать половину атомов, а затем заменить нужные атомы в поверхностном слое (расстояние между атомами $\frac{a}{\sqrt{2}}$). $E_{dim}^{in} = E^{dim+surf} - E^{surf} - 2(E^{adatom+surf} - E^{surf})$, где $E^{dim+surf}$ – полная энергия поверхности с димером в верхнем слое, E^{surf} - полная энергия поверхности, $E^{adatom+surf}$ – полная энергия поверхности с одним атомом в верхнем слое.
- E_{dim}^{on} – энергия связи димера на поверхностном слое. Для вычисления необходимо увеличить решетку до размеров 3*3*6 и убрать половину атомов, а затем поставить нужные атомы на поверхностный слой (расстояние между атомами $\frac{a}{\sqrt{2}}$ и $\frac{1}{2}$ до нижнего атома). $E_{dim}^{in} = E^{dim+surf} - E^{surf} - 2(E^{adatom+surf} - E^{surf})$, где $E^{dim+surf}$ – полная энергия структуры из димера на поверхности, E^{surf} - полная энергия поверхности, $E^{adatom+surf}$ – полная энергия структуры из одного атома.

Для решения задачи необходимо выполнить следующие пункты:

1) Реализовать классы для хранения решетки и обработки ее параметров.
2) Реализовать метод оптимизации нулевого порядка для вычисления параметров потенциалов межатомного

взаимодействия с поддержкой диапазонов значений параметров.

3) Распараллелить вычисление полной энергии системы.
4) Визуализировать полученные результаты.

Метод оптимизации

В работе для решения задачи минимизации функционала ошибки применялся метод Нелдера-Мида или метод деформируемого многогранника. Он представляет собой безусловный метод оптимизации нулевого порядка.

Алгоритм метода:

1. Произвольным образом выбирается n+1 точка в n-мерном пространстве так, чтобы образовывался симплекс. В этих точках вычисляется значения минимизируемой функции.
2. Из вершин симплекса выбирается три точки: x_h, x_g и x_l, в которых достигается максимальное, следующее за максимальным и наименьшее значение, соответственно.
3. Находится центр тяжести всех точек за исключением x_h, $x_c = \frac{1}{n}\sum_i x_i$.
4. Отразим точку x_h относительно x_c с коэффициентом α (при $\alpha = 1$ это будет центральная симметрия, в общем случае — гомотетия), получим точку x_r и вычислим в ней функцию: $f_r = f(x_r)$. Координаты новой точки вычисляются по формуле $x_r = (1+\alpha)x_c - \alpha x_h$.
5. Далее сравниваем значение f_r со значениями f_h, f_g, f_l:
   a. Если $f_r < f_l$, то производим растяжение. Новая точка $x_e = (1-\gamma)x_c + \gamma x_r$ и значение функции $f_e = f(x_e)$.
      i. Если $f_e < f_l$, то заменяем точку x_h на x_e и заканчиваем итерацию (на шаг 9).
      ii. Если $f_e > f_l$, то заменяем точку x_h на x_r и заканчиваем итерацию (на шаг 9).
   b. Если $f_l < f_r < f_g$, то заменяем точку x_h на x_r и переходим на шаг 9.

c. Если $f_h > f_r > f_g$, то меняем обозначения x_r, x_h (и соответствующие значения функции) местами и переходим на шаг 6.
d. Если $f_r > f_h$, то переходим на шаг 6.

6. Строим точку $x_s = \beta x_h + (1-\beta)x_c$ и вычисляем в ней значение f_s.
7. Если $f_s < f_h$, то заменяем точку x_h на x_s и переходим на шаг 8.
8. Если $f_s > f_h$, то производим сжатие симплекса — гомотетию к точке с наименьшим значением x_0: $x_i \to x_0 + (x_i - x_0)/2$ для всех требуемых точек x_i.
9. Последний шаг — проверка сходимости. Может выполняться по-разному, например, оценкой дисперсии набора точек. Суть проверки заключается в том, чтобы проверить взаимную близость полученных вершин симплекса, что предполагает и близость их к искомому минимуму. Если требуемая точность ещё не достигнута, можно продолжить итерации с шага 1.

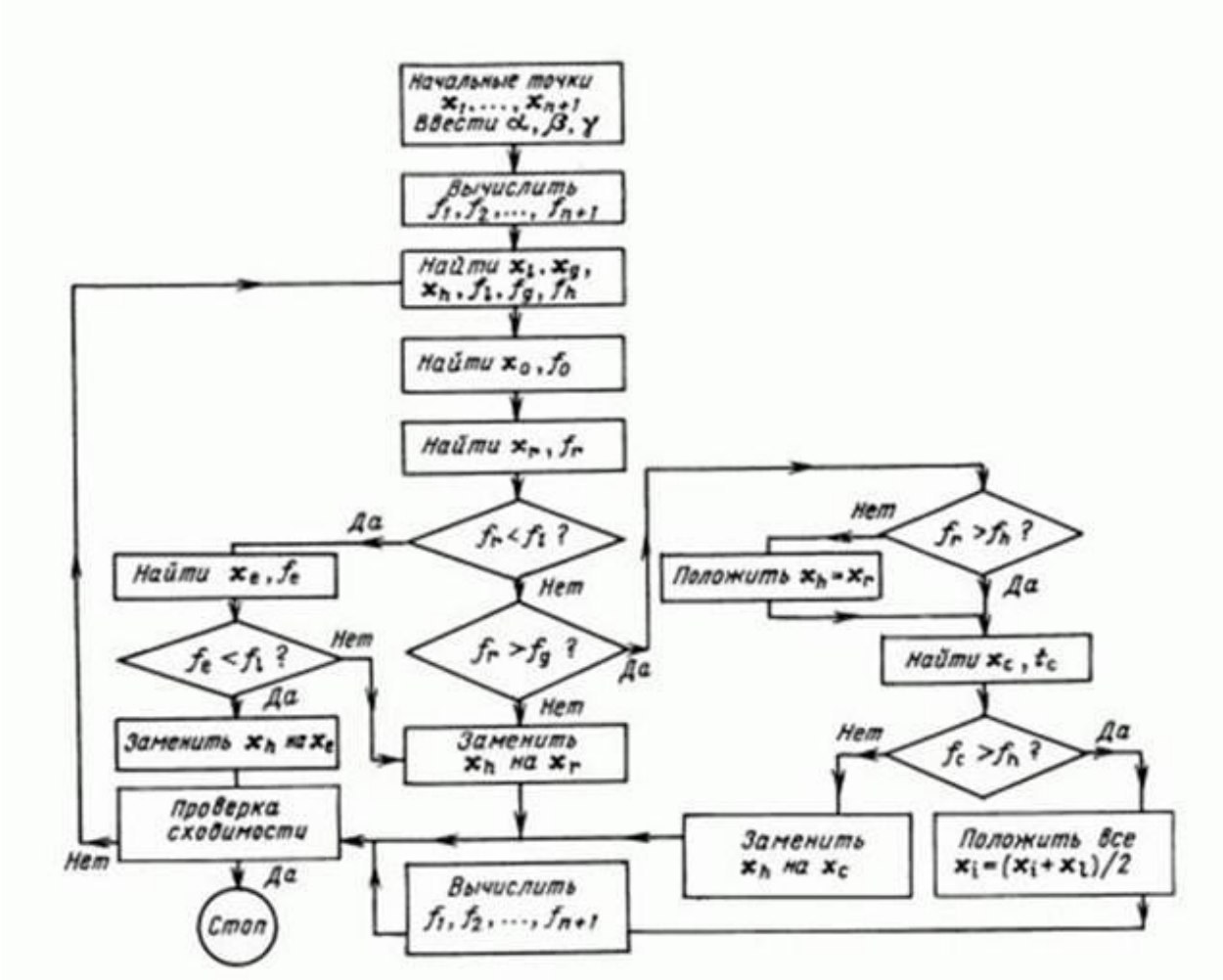


Рис. 3. Блок схема алгоритма Нелдера-Мида.

OpenMP

OpenMP – механизм написания параллельных программ для систем с общей памятью. Для распараллеливания программы необходимо выделить её части, выполнение которых может проводиться одновременно и независимо в нескольких потоках. В основном такими частями являются: (1) различного рода циклы, выполняющие на каждой итерации операции над небольшим набором независимых данных, либо (2) полностью независимые блоки кода, результаты независимой работы которых комбинируются по завершении всех вычислений. Именно на упрощении распараллеливания подобных конструкций и сконцентрирован стандарт OpenMP.

Для реализации параллелизма OpenMP использует модель 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, однако его можно настраивать как во время выполнения программы, так и с помощью переменных окружения во время её запуска.

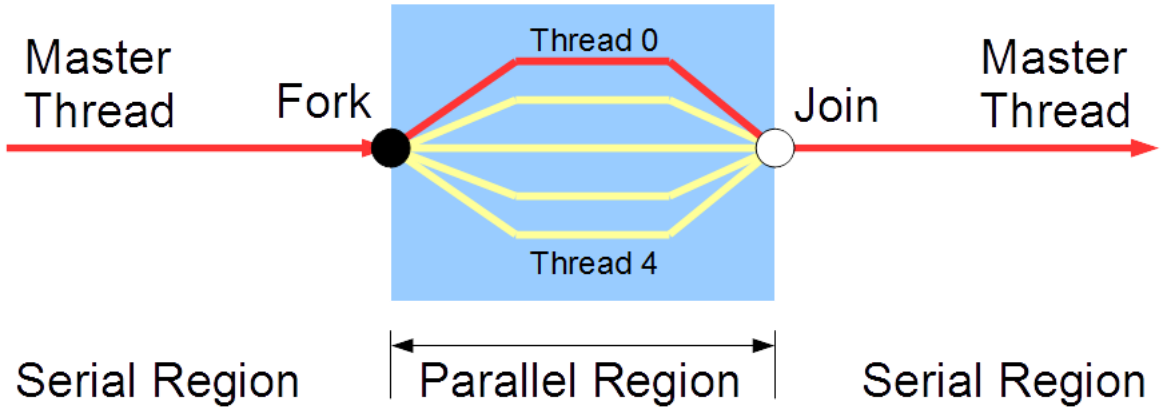


Рис. 4. Схема работы механизма Fork-join.

Результаты работы

Программа реализована на языке C++ стандарта версии 14. Она состоит из следующих классов: Optimizer, Point, Lattice и Solver. Optimizer реализует метод Нелдера-Мида.

Описание класса:

```
struct Optimizer
{
	using func = std::function<double(Solver&, std::vector<double>&)>;
	func f;
	Solver sol;
	std::vector<std::pair<double, double>> constraints;
	std::vector<std::vector<double>> coors;
	unsigned dims;
	double reflect_coeff, compress_coeff, stretch_coeff, global_compress;
	Optimizer(func f, std::vector<std::vector<double>>& init_coors, Solver& sol, const
std::vector<std::pair<double, double>>&,
		double rc = 1, double cc = 0.5, double sc = 2, double gc = 0.5);
	std::tuple<size_t, size_t, size_t> points();
	void reflect();
	void compress(const std::vector<double>& x_c, size_t h, size_t l);
	std::vector<double> find_min(int num_iter);
	void fixParams(double&, int);
};
```

Point – структура для описания трехмерных векторов и работы с ними.

Описание класса:

```
struct Point
{
	double x, y, z;
	Point(double x, double y, double z);
	Point(const std::vector<double>& vec);
	Point(const Point& p) = default;
	Point& operator=(const Point& p) = default;
	Point();
	double& operator[](size_t i);
	static double norm(const Point& p);
	friend Point operator*(const Point& p1, const Point& p2);
	friend Point operator+(const Point& p1, const Point& p2);
	friend Point operator-(const Point& p1, const Point& p2);
	friend std::ostream& operator<< (std::ostream& out, const Point& p);
};
```

Lattice – структура для хранения решетки. Атомы представлены парами <тип атома, координата>. Создать решетку можно двумя способами: считав ее из файла или задав через элементарную ячейку. Также реализовано вычисление расстояния между атомами с учетом периодических условий, растяжения решетки в различных направлениях и границы отсечения. Для возможности манипулирования параметром решетки предусмотрен метод устанавливающий ее размер.

Описание класса:

```
struct Lattice
{
	enum class ElementType
	{
		A, B
	};
```

```
        std::vector<std::pair<ElementType, Point>> atoms;
        Point period;
        double mul;
        Lattice(const std::string& file_name, const Point& period, double m);
        Lattice(const Lattice&) = default;
        Lattice(const Point&, double m, bool cutOff = false);
        Lattice& operator=(const Lattice&) = default;
        std::pair<ElementType, Point>& operator[](int i);
        double distance(size_t i, size_t j, const std::vector<double>& transform =
std::vector<double>());
        void setMul(double m);
        void print();
};
```

Solver – структура для хранения решетки и вычисления энергии системы в различных конфигурациях. После получения каждой из шестерки параметров имеется возможность сохранить их в классе. Метод derivative реализует вычисление второй производной.

Описание класса:

```
struct Solver
{
        std::vector<double> fittedBB;
        std::vector<double> fittedAB;
        std::vector<double> fittedAA;
        Lattice lattice;
        double E_cohA, E_cohB;
        Solver(const Lattice& lat, double e);
        std::vector<double> calculateBB(const std::vector<double>& params);
        std::vector<double> calculateAB(const std::vector<double>& params);
        std::vector<double> calculateAA(const std::vector<double>& params);
        double energy(const std::vector<double>& params, const std::vector<double>&
transform = {});
        double derivative(double energy, const std::vector<double>& params, const
std::vector<double>& pos, const std::vector<double>& neg);
};
```

После оптимизации получены следующие параметры:

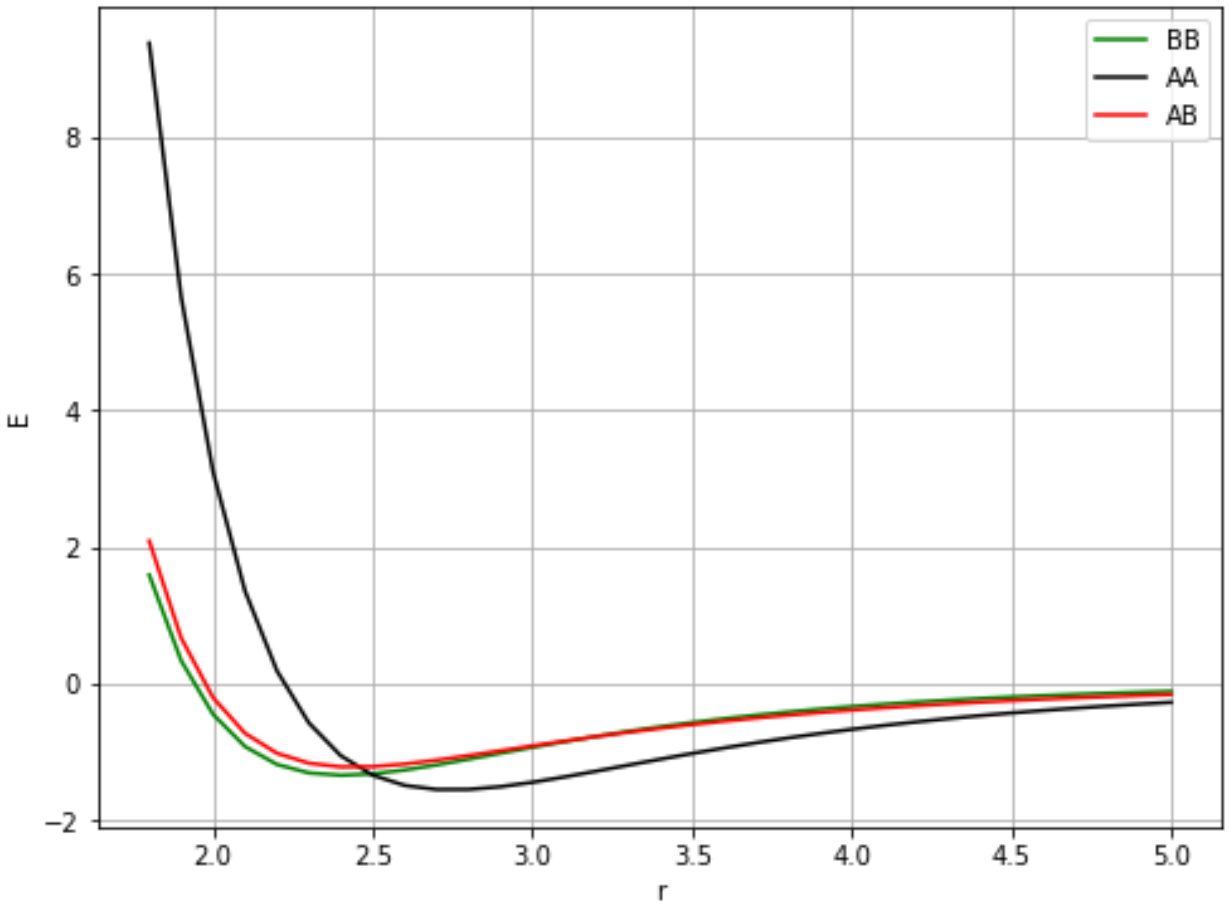
	Ag-Ag	**Co-Ag**	**Co-Co**
A^0	-0.0159319	-0.057879	-0.0580274
A^1	0.111505	0.102937	0.060168
p	10.2265	10.3437	10.0452
q	3.0267	2.36513	2.93416
ξ	1.20272	1.23069	1.25747
r_0	2.82208	2.69373	3.29989

С учетом полученных параметров и подобранного параметра решетки вычислены следующие значения

	Ag(001) (табличные)	**Ag(001) (вычисленные)**
α	4.085	4.02547

E_c	-2.960	-2.96104
B	1.08	1.08177
C_{11}	1.32	1.31352
C_{12}	0.97	0.965387
C_{44}	0.51	0.509472
	Co (табличные)	**Co (вычисленные)**
E_{sol}	0.63	0.629979
E_{dim}^{in}	-0.08	-0.0806687
E_{dim}^{on}	-0.69	-0.651155

Графики потенциалов межатомного взаимодействия:



Распараллеливание применялось для вычисления энергии полной системы. Далее представлена зависимость скорости вычисления параметров системы B-B:

Кол-во потоков	Время работы
1	129.013s
2	57.2638s
4	35.6225s

8	26.5735s

Вывод

В рамках курсовой работы была создана программная система для вычисления параметров межатомного взаимодействия Ag и Co. Для ускорения работы применялась технология распараллеливания OpenMP. Для оптимизации применялся метод Нелдера-Мида с учетом ограничений на подбираемые параметры.

Код программы:

Optimizer:

```
Optimizer::Optimizer(func f, std::vector<std::vector<double>>& init_coors,
	Solver& sol, const std::vector<std::pair<double, double>>& cons,
	double rc, double cc, double sc, double gc) : sol(sol), constraints(cons), f(f)
{
	reflect_coeff = rc;
	compress_coeff = cc;
	stretch_coeff = sc;
	global_compress = gc;
	coors = init_coors;
	dims = init_coors[0].size();
}

std::tuple<size_t, size_t, size_t> Optimizer::points()
{
	std::vector<std::pair<int, double>> values(dims + 1, { 0, 0 });
	size_t h, g, l;
	for (size_t i = 0; i < coors.size(); ++i)
		values[i] = { i, f(sol, coors[i]) };
	std::stable_sort(values.begin(), values.end(),
		[](const std::pair<int, double >& a, const std::pair<int, double >& b)
		{return a.second > b.second; });
	h = values[0].first;
	g = values[1].first;
	l = values[values.size() - 1].first;
	return std::make_tuple(h, g, l);
}

void Optimizer::reflect()
{
	size_t size = dims + 1, h, g, l;
	auto res = points();
	std::vector<double> x_c(dims, 0);
	std::vector<double> x_r(dims, 0);
	h = std::get<0>(res), g = std::get<1>(res), l = std::get<2>(res);
	for (size_t i = 0; i < size; i++)
		for (size_t j = 0; j < size - 1; j++)
			x_c[j] += coors[i][j];
	for (size_t i = 0; i < size - 1; ++i)
	{
		x_c[i] -= coors[h][i];
		x_c[i] /= size - 1;
		fixParams(x_c[i], i);
	}
	for (size_t i = 0; i < dims; ++i)
	{
		x_r[i] = (1 + reflect_coeff) * x_c[i] - reflect_coeff * coors[h][i];
		fixParams(x_r[i], i);
	}
	auto result = f(sol, x_r);
	if (result < f(sol, coors[l]))
	{
		std::vector<double> x_e(dims, 0);
		for (size_t i = 0; i < size - 1; ++i)
		{
			x_e[i] = (1 - stretch_coeff) * x_c[i] + stretch_coeff * x_r[i];
			fixParams(x_e[i], i);
		}
		if (f(sol, x_e) < f(sol, x_r))
			coors[h] = x_e;
```

```
                else
                        coors[h] = x_r;
        }
        else if ((result >= f(sol, coors[l])) and (result < f(sol, coors[g])))
                coors[h] = x_r;
        else if ((result >= f(sol, coors[g])) and (result < f(sol, coors[h])))
        {
                coors[h] = x_r;
                compress(x_c, h, l);
        }
        else if (f(sol, coors[h]) <= result)
                compress(x_c, h, l);
        else
        {
                compress(x_c, h, l);
        }
}

void Optimizer::compress(const std::vector<double>& x_c, size_t h, size_t l)
{
        std::vector<double> x_s(dims, 0);
        auto x_h = coors[h];
        for (size_t i = 0; i < dims; ++i)
        {
                x_s[i] = (1 - compress_coeff) * x_c[i] + compress_coeff * x_h[i];
                fixParams(x_s[i], i);
        }
        if (f(sol, x_s) < f(sol, x_h))
                coors[h] = x_s;
        else
        {
                for (size_t i = 0; i < dims + 1; ++i)
                {
                        for (size_t j = 0; j < dims; ++j)
                        {
                                coors[i][j] = coors[l][j] + global_compress * (coors[i][j] -
coors[l][j]);
                                fixParams(coors[i][j], j);
                        }
                }
        }
}

std::vector<double> Optimizer::find_min(int num_iter)
{
        int i = 0;
        double error = 0;
        do
        {
                if (i == num_iter)
                        break;
                reflect();
                auto poin = coors[std::get<0>(points())];
                error= f(sol, poin);
                for (auto& val : coors)
                        for (size_t i = 0; i < val.size(); ++i)
                                fixParams(val[i], i);
                i++;
        } while (error>=1e-2);
        return coors[std::get<0>(points())];
}

void Optimizer::fixParams(double& x, int i)
{
```

```
	if (x <= constraints[i].first || x >= constraints[i].second)
	{
		if (x <= constraints[i].first)
			x = constraints[i].first + (constraints[i].second -
constraints[i].first) / 5 * ((rand() % 10'000) / 10'000.0);
		if (x >= constraints[i].second)
			x = constraints[i].second - (constraints[i].second -
constraints[i].first) / 5 * ((rand() % 10'000) / 10'000.0);
	}
}
```

Point:

```
struct Point
{
	double x, y, z;
	Point(double x, double y, double z);
	Point(const std::vector<double>& vec);
	Point(const Point& p) = default;
	Point& operator=(const Point& p) = default;
	Point();
	double& operator[](size_t i);
	static double norm(const Point& p);
	friend Point operator*(const Point& p1, const Point& p2)
	{
		return Point(p1.x * p2.x, p1.y * p2.y, p1.z * p2.z);
	}
	friend Point operator+(const Point& p1, const Point& p2)
	{
		return Point(p1.x + p2.x, p1.y + p2.y, p1.z + p2.z);
	}
	friend Point operator-(const Point& p1, const Point& p2)
	{
		return Point(p1.x - p2.x, p1.y - p2.y, p1.z - p2.z);
	}
	friend std::ostream& operator<< (std::ostream& out, const Point& p)
	{
		out << p.x << " " << p.y << " " << p.z;
		return out;
	}
};

Point::Point(double x, double y, double z): x(x), y(y), z(z)
{}

Point::Point(const std::vector<double>& vec)
{
	if (vec.size() == 3)
		x = vec[0], y = vec[1], z = vec[2];
	else
		x = y = z = 0;
}

double& Point::operator[](size_t i)
{
	switch (i)
	{
	case 0: return x;
	case 1: return y;
	case 2: return z;
	}
}

double Point::norm(const Point& p)
```

```
{
	return sqrt(p.x * p.x + p.y * p.y + p.z * p.z);
}

Point::Point()
{
	x = y = z = 0;
}
```

Lattice:

```
Lattice::Lattice(const std::string& file_name, const Point& period, double m) :
period(period), mul(m)
{
    std::ifstream in(file_name);
    std::string line;
    if (in.is_open())
    {
        while (getline(in, line))
        {
            std::vector<std::string> temp;
            std::istringstream input(line);
            while (!input.eof())
            {
                std::string substring;
                input >> substring;
                temp.push_back(substring);
            }
            if (temp.size() == 4)
            {
                Point p(std::stod(temp[1]),std::stod(temp[2]), std::stod(temp[3]));
                if (p.x == period.x || p.y == period.y || p.z == period.z)
                    continue;
                if (temp[0] == "Ag")
                    atoms.push_back(std::make_pair(Lattice::ElementType::B,
                        Point(std::stod(temp[1]), std::stod(temp[2]),
std::stod(temp[3]))));
                else
                    atoms.emplace_back(std::make_pair(Lattice::ElementType::A,
                        Point(std::stod(temp[1]), std::stod(temp[2]),
std::stod(temp[3]))));
            }
        }
        in.close();
    }
    else
        throw std::exception("");
}

Lattice::Lattice(const Point& p, double m, bool cutOff)
{
    period = p;
    mul = m;
    std::vector<Point> basis = {
        Point(0, 0, 0),
        Point(0.5, 0, 0.5),
        Point(0.5, 0.5, 0),
        Point(0, 0.5, 0.5)
    };

    for (int i = 0; i < p.x; ++i) {
        for (int j = 0; j < p.y; ++j) {
            for (int k = 0; k < p.z; ++k) {
                for (auto& val : basis) {
```

```
                    if (!cutOff || (val + Point(i, j, k)).z <= p.z/2)
                        atoms.push_back(std::make_pair(Lattice::ElementType::B, val +
Point(i, j, k)));
                }
            }
        }
    }
}

void Lattice::print()
{
    for (size_t i = 0; i < atoms.size(); ++i)
    {
        std::cout << i << " " << atoms[i].second << "\n";
    }
    std::cout << "\n";
}

std::pair<Lattice::ElementType, Point>& Lattice::operator[](int i)
{
    return atoms[i];
}

void Lattice::setMul(double m)
{
    mul = m;
}

double Lattice::distance(size_t i, size_t j, const std::vector<double>& transform)
{
    auto atom1 = atoms[i].second, atom2 = atoms[j].second;
    auto dis = atom2 - atom1;

    for (size_t i = 0; i < 3; ++i)
        if (dis[i] > period[i] / 2)
            dis[i] -= period[i];
        else if (dis[i] < -period[i] / 2)
            dis[i] += period[i];

    if (transform.size() == 3)
        dis = dis * Point(transform);
    else if (transform.size() == 5)
        dis = Point(dis.x * transform[0] + dis.y * transform[1],
            dis.x * transform[2] + dis.y * transform[3],
            dis.z * transform[4]);

    double norm = Point::norm(dis);

    if (norm > 1.7)
        return 0;
    return norm * mul;
}
```

Solver:

```
#include <omp.h>
#include "Solver.h"

Solver::Solver(const Lattice& lat, double e):lattice(lat), E_cohA(e), E_cohB(0)
{
       fittedAB = fittedBB = fittedAA = std::vector<double>();
}
```

```
std::vector<double> Solver::calculateBB(const std::vector<double>& params)
{
	std::vector<double> res;
	if (fittedBB.empty())
	{
		double a0_min = 4.0, a_max = 4.10;
		double m = 1e30, p = 0;
		for (double x = a0_min; x <= 4.15; x += (rand() % 10'000 )/500'000.0)
		{
			lattice.setMul(x);
			auto val = energy(params);
			if (val < m)
			{
				m = val;
				p = x;
			}
		}
		lattice.setMul(p);
	}
	double E_0 = energy(params);
	res.push_back(E_0 / lattice.atoms.size());

	double V_0 = std::pow(lattice.mul, 3) * lattice.period[0] * lattice.period[1] *
lattice.period[2];

	double C_11Deriv = derivative(E_0, params, { 1 + 1e-2, 1 + 1e-2, 1 }, { 1 - 1e-2,
1 - 1e-2, 1 });
	double C_12Deriv = derivative(E_0, params, { 1 + 1e-2, 1 - 1e-2, 1 }, { 1 - 1e-2,
1 + 1e-2, 1 });
	double BDeriv = derivative(E_0, params, { 1 + 1e-2, 1 + 1e-2, 1 + 1e-2 }, { 1 -
1e-2, 1 - 1e-2, 1 - 1e-2 });
	res.push_back(2 * (BDeriv / (9 * V_0)) * 0.8019);

	res.push_back(((C_11Deriv + C_12Deriv) / (2*V_0))*0.8019);
	res.push_back(((C_11Deriv - C_12Deriv) / (2 * V_0))*0.8019);


	double C_44Deriv = derivative(E_0, params, { 1, 1e-2, 1e-2, 1, 1 / (1 - 1e-2 * 1e-
2) },
		{ 1, -1e-2, -1e-2, 1, 1 / (1 - 1e-2 * 1e-2) });
	double C_44 = (C_44Deriv / (2 * V_0))*0.8019;
	res.push_back(C_44);
	return res;
}


double Solver::energy(const std::vector<double>& params, const std::vector<double>&
transform)
{
	double sum_ = 0;
	#pragma omp parallel num_threads(8)
	{
		#pragma omp for
		for (int i = 0; i < lattice.atoms.size(); ++i)
		{
			double sum_b = 0, sum_r = 0;
			for (int j = 0; j < lattice.atoms.size(); ++j)
			{
				if (i != j)
				{
					double distance = lattice.distance(i, j, transform);
					if (distance != 0)
					{
						if (lattice.atoms[i].first ==
Lattice::ElementType::B && lattice.atoms[i].first == lattice.atoms[j].first)
```

```
                                        {
                                                if (fittedBB.empty())
                                                {
                                                        sum_r += ((params[0] * (distance -
params[2]) + params[1]) * exp(-params[3] * (distance / params[2] - 1)));
                                                        sum_b += params[5] * params[5] *
exp(-2 * params[4] * (distance / params[2] - 1));
                                                }
                                                else
                                                {
                                                        sum_r += ((fittedBB[0] * (distance
- fittedBB[2]) + fittedBB[1]) * exp(-fittedBB[3] * (distance / fittedBB[2] - 1)));
                                                        sum_b += fittedBB[5] * fittedBB[5]
* exp(-2 * fittedBB[4] * (distance / fittedBB[2] - 1));
                                                }
                                        }
                                        else if (lattice.atoms[i].first !=
lattice.atoms[j].first)
                                        {
                                                if (fittedAB.empty())
                                                {
                                                        sum_r += ((params[0] * (distance -
params[2]) + params[1]) * exp(-params[3] * (distance / params[2] - 1)));
                                                        sum_b += params[5] * params[5] *
exp(-2 * params[4] * (distance / params[2] - 1));
                                                }
                                                else
                                                {
                                                        sum_r += ((fittedAB[0] * (distance
- fittedAB[2]) + fittedAB[1]) * exp(-fittedAB[3] * (distance / fittedAB[2] - 1)));
                                                        sum_b += fittedAB[5] * fittedAB[5]
* exp(-2 * fittedAB[4] * (distance / fittedAB[2] - 1));
                                                }
                                        }
                                        else
                                        {
                                                if (fittedAA.empty())
                                                {
                                                        sum_r += ((params[0] * (distance -
params[2]) + params[1]) * exp(-params[3] * (distance / params[2] - 1)));
                                                        sum_b += params[5] * params[5] *
exp(-2 * params[4] * (distance / params[2] - 1));
                                                }
                                                else
                                                {
                                                        sum_r += ((fittedAA[0] * (distance
- fittedAA[2]) + fittedAA[1]) * exp(-fittedAA[3] * (distance / fittedAA[2] - 1)));
                                                        sum_b += fittedAA[5] * fittedAA[5]
* exp(-2 * fittedAA[4] * (distance / fittedAA[2] - 1));
                                                }
                                        }
                                }
                        }
                        #pragma omp atomic
                        sum_ += sum_r - sqrt(sum_b);
                }
        }
        return sum_;
}

double Solver::derivative(double energy, const std::vector<double>& params, const
std::vector<double>& pos, const std::vector<double>& neg)
{
        double positiveEnergy = this->energy(params, pos);
```

```
	double negativeEnergy = this->energy(params, neg);

	return (positiveEnergy - 2 * energy + negativeEnergy) / (1e-2 * 1e-2);
}

std::vector<double> Solver::calculateAB(const std::vector<double>& params)
{
	double E_B = energy(params);
	lattice[0].first = Lattice::ElementType::A;
	auto E_AB = energy(params);
	lattice[0].first = Lattice::ElementType::B;
	return std::vector<double>{E_AB - E_B - E_cohA + E_cohB};
}

std::vector<double> Solver::calculateAA(const std::vector<double>& params)
{
	auto lat = lattice;
	lattice = Lattice(Point(3, 3, 6), lat.mul, true);
	double E_surf = energy(params);
	//lat.print();
	lattice[12].first = Lattice::ElementType::A;
	double E_adaatom = energy(params);
	//std::cout << E_surf << " " << E_adaatom << "\n";
	lattice[13].first = Lattice::ElementType::A;
	double E_dim = energy(params);
	lattice[12].first = lattice[13].first = Lattice::ElementType::B;
	double E_indim = E_dim - E_surf - 2 * (E_adaatom - E_surf);
	lattice.atoms.push_back(std::make_pair(Lattice::ElementType::A, Point(0.5, 0,
3.5)));
	E_adaatom = energy(params);
	lattice.atoms.push_back(std::make_pair(Lattice::ElementType::A, Point(0, 0.5,
3.5)));
	E_dim = energy(params);
	double E_ondim = E_dim - E_surf - 2 * (E_adaatom - E_surf);
	lattice = lat;
	return std::vector<double>{E_indim, E_ondim};
}
```

Main:

```
#include <iostream>
#include <chrono>
#include <functional>
#include "Point.h"
#include "Lattice.h"
#include "Solver.h"
#include "Optimizer.h"

using namespace std::placeholders;

double func(Solver& sol, std::vector<double>& params, const std::vector<double> target,
std::vector<double>(Solver::* func)(const std::vector<double>&));
std::vector<double> GenerateVec(int n, std::vector<std::pair<double, double>>& mid);
void printVec(const std::vector<double>&);
double energy(const std::vector<double>& params, double dist);

int main()
{
	srand(time(0));
	int n = 6;
	//Lattice lattice("Ag.xyz", Point(3, 3, 3), 4.085);
	std::vector<std::vector<double>> init_coors;
	std::vector<std::pair<double, double>> constraints{
```

```
                std::make_pair(-0.2, 0.1),
                std::make_pair(0, 0.2),
                std::make_pair(1.9, 3.0),
                std::make_pair(10, 12),
                std::make_pair(2, 5),
                std::make_pair(0.9, 1.3)
                //std::make_pair(-0.3, 0.1),
                //std::make_pair(0, 0.2),
                //std::make_pair(1.9, 3.3),
                //std::make_pair(9, 13),
                //std::make_pair(2, 5),
                //std::make_pair(0.9, 2)
        };
        std::vector<std::pair<double, double>> constraints1
        {
                std::make_pair(-0.3, 0.1),
                std::make_pair(0, 0.2),
                std::make_pair(1.9, 3.3),
                std::make_pair(9, 13),
                std::make_pair(2, 5),
                std::make_pair(0.9, 2)
        };

        for (size_t i = 0; i < n+1; ++i)
                init_coors.push_back(GenerateVec(n, constraints));
        Lattice lattice(Point(3, 3, 3),  4.085);
        Solver sol(lattice, -4.32);


        std::cout << "\nStart:\n";
        auto start = std::chrono::high_resolution_clock::now();
        auto f = std::bind(func, _1, _2, std::vector<double>{ -2.960, 1.08, 1.32, 0.97,
0.51 }, &Solver::calculateBB);
        Optimizer opt(f, init_coors, sol, constraints);
        auto p = opt.find_min(251);
        auto r = sol.calculateBB(p);
        auto finish = std::chrono::high_resolution_clock::now();
        std::chrono::duration<double, std::chrono::seconds::period> elapsedtime =
                finish - start;
        std::cout << "elapsed time " << elapsedtime.count() << 's' << std::endl;
        printVec(p);
        std::cout << sol.lattice.mul << " ";
        printVec(r);

        std::cout << "\nStart:\n";
        start = std::chrono::high_resolution_clock::now();
        sol.E_cohB = r[0];
        sol.lattice.setMul(sol.lattice.mul);
        sol.fittedBB = p;
        std::cout << " " << sol.lattice.mul << " ";
        Optimizer opt1(std::bind(func, _1, _2, std::vector<double>{0.63}, &
Solver::calculateAB), init_coors, sol, constraints);
        auto p1 = opt1.find_min(501);
        auto r1 = sol.calculateAB(p1);
        finish = std::chrono::high_resolution_clock::now();
        elapsedtime =
                finish - start;
        std::cout << "elapsed time " << elapsedtime.count() << 's' << std::endl;
        printVec(p1);
        printVec(r1);
        sol.fittedBB = std::vector<double>{};

        std::cout << "\nStart:\n";
        start = std::chrono::high_resolution_clock::now();
        sol.fittedBB = p;
```

```
	sol.fittedAB = p1;
	sol.E_cohB = r[0];
	sol.lattice.setMul(sol.lattice.mul);
	Optimizer opt2(std::bind(func, _1, _2, std::vector<double>{ -0.08, -0.69 }, &
Solver::calculateAA), init_coors, sol, constraints1);
	auto p2 = opt2.find_min(501);
	auto r2 = sol.calculateAA(p2);
	finish = std::chrono::high_resolution_clock::now();
	elapsedtime =
		finish - start;
	std::cout << "elapsed time " << elapsedtime.count() << 's' << std::endl;
	sol.fittedBB = std::vector<double>{};
	sol.fittedAB = std::vector<double>{};
	printVec(p2);
	printVec(r2);

	//sol.fittedBB = std::vector<double>{ -0.0159319, 0.111505, 2.82208, 10.2265,
3.0267, 1.20272 };
	//sol.lattice.setMul(4.02547);
	//printVec(sol.calculateBB({}));

	//sol.E_cohB = -2.96104;
	//sol.lattice.setMul(4.02547);
	//sol.fittedBB = std::vector<double>{ -0.0159319, 0.111505, 2.82208, 10.2265,
3.0267, 1.20272 };
	//printVec(sol.calculateAB({ -0.057879, 0.102937, 2.69373, 10.3437, 2.36513,
1.23069, 0.629993 }));
	//sol.fittedBB = std::vector<double>{};

	//sol.E_cohB = -2.96104;
	//sol.lattice.setMul(4.02547);
	//sol.fittedBB = std::vector<double>{ -0.0159319, 0.111505, 2.82208, 10.2265,
3.0267, 1.20272 };
	//sol.fittedAB = std::vector<double>{ -0.057879, 0.102937, 2.69373, 10.3437,
2.36513, 1.23069, 0.629993 };
	//printVec(sol.calculateAA({ -0.0580274, 0.060168, 3.29989, 10.0452, 2.93416,
1.25747 }));
	//sol.fittedBB = std::vector<double>{};
	//sol.fittedAB = std::vector<double>{};
}


//Params BB: -0.0159319, 0.111505, 2.82208, 10.2265, 3.0267, 1.20272
//Params AB: -0.057879, 0.102937, 2.69373, 10.3437, 2.36513, 1.23069
//Params AA: -0.0580274, 0.060168, 3.29989, 10.0452, 2.93416, 1.25747

// BB_target: 0.599535, 0.289139, 2.71249, 11.8106, 0.532275, 0.881892
// AB_target: 0.63
// AA_target: -0.08, -0.69

double func(Solver& sol, std::vector<double>& params, const std::vector<double> target,
std::vector<double> (Solver::*func)(const std::vector<double>& ))
{
	double sum_ = 0;
	auto res = (sol.*func)(params);
	for (size_t i = 0; i < res.size(); ++i)
	{
		sum_ += pow(res[i]/target[i] - 1, 2) / (target.size());
	}
	return sum_;
}

std::vector<double> GenerateVec(int n, std::vector<std::pair<double, double>>& mid)
{
	std::vector<double> vec(n, 0);
```

```
	for (size_t i = 0; i < n; ++i)
	{
		double middle = (mid[i].second + mid[i].first) / 2;
		double num = middle + (mid[i].second - mid[i].first) / 2 * ((rand() %
10'000) / 5'000.0 - 1.0);
		vec[i] = num;
	}
	return vec;
}

void printVec(const std::vector<double>& vec)
{
	for (size_t i = 0; i < vec.size(); ++i)
		std::cout << vec[i] << " ";
	std::cout << "\n";
}

double energy(const std::vector<double>& params, double distance)
{
	return ((params[0] * (distance - params[2]) + params[1]) * exp(-params[3] *
(distance / params[2] - 1))) -
		sqrt(params[5] * params[5] * exp(-2 * params[4] * (distance / params[2] -
1)));
}
```