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

Мирпулатов Исломбек Пулат-угли

Определение потенциалов межатомного взаимодействия системы Ag - Cr

КУРСОВАЯ РАБОТА

Преподаватель:
к.ф-м.н.
Д.И. Бажанов

Москва, 2022

Оглавление

1. Описание задачи

В рамках задачи требуется определить потенциалы межатомного взаимодействия
(A-A, A-B, B-B) для системы A/B, где A - Cr (Хром), B - Ag (Серебро).

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

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

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

E - полная энергия системы
E_B^i - энергии притяжения
E_R^i - энергия отталкивания
r_{ij} - расстояние между атомами i (Сорт A) и j (Сорт B)

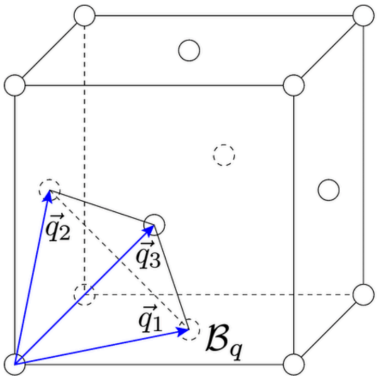
Параметры потенциалов: A^1, A^0, r_0, p, q, ξ (для A-A, A-B и B-B взаимодействий 3 набора по 6 параметров) необходимо определить с помощью алгоритма минимизации функции ошибки относительно известных табличных параметров.

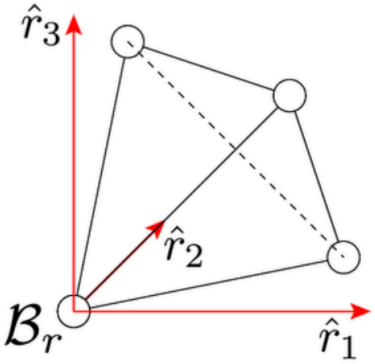
Все расчёты полных энергий E проводятся для кристаллической решётки размером 3 × 3 × 3 в единицах элементарной ячейки ГЦК структуры. Все расчёты статические, проводятся без релаксации атомных позиций. Атомы располагаются в узлах “идеальной” ГЦК решётки.

Полученное решение оптимизировать с помощью технологии распараллеливания.

2. Использованные формулы для расчета

Далее изображены ГЦК решетка Бравэ и элементарная ячейка из которой образуется кристаллическая решетка:





Для оптимизации параметров в качестве целевой функции минимизации был выбран следующий тип функции:

$$f = \frac{1}{n}\sum_{i}^{n}\frac{(TrueValue_i - Value_i)^2}{TrueValue_i^2}, \text{ где}$$

TrueValue - основные значения кристаллической решетки, значения взятые из справочника
Value - значения, посчитанные с текущим набором параметров потенциалов

Для описания взаимодействия атомов сорта B-B в целевом функционале необходимо использовать следующие значения кристаллической решетки:

- a - параметр решетки
- E_{coh} - когезионная энергия
- B - модуль всестороннего растяжения, сжатия
- C_{11}, C_{12}, C_{44} - константы упругости

В результате функция ошибки принимает следующий вид:

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

Для описания взаимодействия атомов сорта A-B в целевом функционале необходимо использовать следующие значения кристаллической решетки:

- E_{sol} - энергия растворимости

В результате функция ошибки принимает следующий вид:

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

Для описания взаимодействия атомов сорта A-A в целевом функционале необходимо использовать следующие значения кристаллической решетки:

- E_{dim}^{in} - энергия связи димера в поверхностном слое

- E_{dim}^{out} - энергия связи димера на поверхности

В результате функция ошибки принимает следующий вид:

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

Сопутствующие формулы для предыдущих:

$$r_0 = \frac{a}{\sqrt{2}}$$

$$cutoff = 1.7a$$

$$V_0 = a^3 N_x N_y N_z$$

$$E = E_{coh} n$$

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

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

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

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

$$E_{sol} = E^{AB} - E^{B} - E^{A}_{coh} + E^{B}_{coh}$$

$$E^{in}_{dim} = (E^{dim+surf} - E^{surf}) * 2(E^{adatom+surf} - E^{surf}),$$

$$E^{on}_{dim} = (E^{dim+surf} - E^{surf}) * 2(E^{adatom+surf} - E^{surf}),$$

$$D_B = \begin{bmatrix} 1+\alpha & 0 & 0 \\ 0 & 1+\alpha & 0 \\ 0 & 0 & 1+\alpha \end{bmatrix},$$

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

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

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

- r_0 - растояние для ближайшего соседа

- a - параметр решетки

- $cutoff$ - радиус взаимодействия атомов

- V_0 - равновесный объем

- $N_x N_y N_z$ - количество ячеек x, y, z

- E - полная энергия

- E_{coh} - когезионная энергия

- n - количество атомов

- B - модуль всестороннего растяжения (сжатия). Равномерное растяжение (сжатие) происходит по трем эквивалентным направлениям кубической решетки

- C_{11} - константа упругости. Равномерное растяжение в данном случае происходит в плоскости xy

- C_{12} - константа упругости. Растяжение в данном случае происходит вдоль оси x и сжатие вдоль оси y

- C_{44} - константа упругости

- E_{dim}^{in} - энергия связи димера в поверхностном слое

- E_{dim}^{out} - энергия связи димера на поверхности

- Матрицы D - матрицы деформации узлов, в которых располагаются атомы

3. Реализация

1. Реализован подсчет энергии взаимодействия между атомами учитывающих их тип.

2. Найдено значение a для B-B, исходя из параметров потенциала взятых из таблицы. Значение найдено при помощи алгоритма тернарнарного поиска. (Описание в главе 4)

3. Реализован подсчет основных характеристик и параметров решетки.

4. Реализован алгоритм оптимизации Неледера Мида. (Описание в главе 5)

5. Найдены параметры для двух типов взаимодействий: B-B, A-B.

6. Построены графики зависимости полной энергии от расстояния.

4. Алгоритм тернарного поиска

Возьмём любые две точки m_1, m_2 в этом отрезке: $l < m_1 < m_2 < r$. Посчитаем значения функции $f(m_1)$ и $f(m_2)$. Дальше у нас получается три варианта:

- Если окажется, что $f(m_1) < f(m_2)$, то искомый максимум не может находиться в левой части, т.е. в части $[l; m_1]$. В этом легко убедиться: если в левой точке функция меньше, чем в правой, то либо эти две точки находятся в области "подъёма" функции, либо только левая точка находится там. В любом случае, это означает, что максимум дальше имеет смысл искать только в отрезке $[m_1; r]$.

- Если, наоборот, $f(m_1) > f(m_2)$, то ситуация аналогична предыдущей с точностью до симметрии. Теперь искомый максимум не может находиться в правой части, т.е. в части $[m_2; r]$, поэтому переходим к отрезку $[l; m_2]$.

- Если $f(m_1) = f(m_2)$, то либо обе эти точки находятся в области максимума, либо левая точка находится в области возрастания, а правая — в области убывания (здесь существенно используется то, что возрастание/убывание строгие). Таким образом, в дальнейшем поиск имеет смысл производить в отрезке $[m_1; m_2]$, но (в целях упрощения кода) этот случай можно отнести к любому из двух предыдущих.

Таким образом, по результату сравнения значений функции в двух внутренних точках мы вместо текущего отрезка поиска $[l; r]$ находим новый отрезок $[l'; r']$. Повторим теперь все действия для этого нового отрезка, снова получим новый, строго меньший, отрезок, и т.д.

Рано или поздно длина отрезка станет маленькой, меньшей заранее определённой константы-точности, и процесс можно останавливать. Этот метод численный, поэтому после остановки алгоритма можно приближённо считать, что во всех точках отрезка $[l; r]$ достигается максимум; в качестве ответа можно взять, например, точку l.

5. Метод оптимизации Нелдера Мида

Метод Нелдера — Мида — метод оптимизации (поиска минимума) функции от нескольких переменных. Простой и в тоже время эффективный метод, позволяющий оптимизировать функции без использования градиентов. В рамках моей курсовой работы использовалась функция optimize из модуля scipy.optimize популярной библиотеки для языка python, которая используется для математических расчетов. []

Алгоритм заключается в формировании симплекса (**simplex**) и последующего его деформирования в направлении минимума, посредством трех операций:

1) **Отражение** (reflection);
2) **Растяжение** (expansion);
3) **Сжатие** (contract);

Симплекс представляет из себя геометрическую фигуру, являющуюся n — мерным обобщением треугольника. Для одномерного пространства — это отрезок, для двумерного — треугольник. Таким образом n — мерный симплекс имеет n + 1 вершину.

1) Пусть $f(x, y)$ функция , которую необходимо оптимизировать. На первом шаге выбираем три случайные точки (об этом чуть позже) и формируем симплекс (треугольник). Вычисляем значение функции в каждой точке: $f(V_1), f(V_2), f(V_3)$. Сортируем точки по значениям функции $f(x, y)$ в этих точках, таким образом получаем двойное неравенство:

$$f(V_2) \leq f(V_1) \leq f(V_3)$$

Мы ищем минимум функции, а следовательно, на данном шаге лучшей будет та точка, в которой значение функции минимально. Для удобства переобозначим точки следующим образом:

b =V_2 , g = V_1, w = V_3, где best, good, worst — соответственно.

2) На следующем шаге находим середину отрезка, точками которого являются g и b. Т.к. координаты середины отрезка равны полусумме координат его концов, получаем:

$$mid = (\frac{x_1 + x_2}{2}; \frac{y_1 + y_2}{2})$$

В более общем виде можно записать:

$$mid = \frac{1}{n}\sum_{i=1}^{n} x_i$$

3) Применяем операцию отражения:
Находим точку x_r, следующим образом:

$$x_r = mid + (mid - w)$$

Т.е. фактически отражаем точку w относительно mid. В качестве коэффициента берут как правило 1. Проверяем нашу точку: если $f(x_r) < f(g)$, то это хорошая точка. А теперь попробуем расстояние увеличить в 2 раза, вдруг нам повезет и мы найдем точку еще лучше.

4) Применяем операцию растяжения:
Находим точку x_e следующим образом:

$$x_e = mid + (x_r - mid)$$

В качестве γ принимаем γ = 2, т.е. расстояние увеличиваем в 2 раза.
Проверяем точку x_e:
Если $f(x_e) < f(b)$, то нам повезло и мы нашли точку лучше, чем есть на данный момент, если бы этого не произошло, мы бы остановились на точке x_r.
Далее заменяем точку w на x_e.

5) Если же нам совсем не повезло и мы не нашли хороших точек, пробуем операцию сжатия.
Как следует из названия операции мы будем уменьшать наш отрезок и искать хорошие точки внутри треугольника.
Пробуем найти хорошую точку x_c:

$$x_c = mid + (w - mid)$$

Коэффициент β принимаем равным 0.5, т.е. точка x_c на середине отрезка $wmid$.

Алгоритм заканчивается, когда:
1) Было выполнено необходимое количество итераций.
2) Площадь симплекса достигла определенной величины.
3) Текущее лучшее решение достигло необходимой точности.

6. Ускорение кода

6.1 Ускорение кода на чистом Python

Для полного ускорения кода на Python необходимо сначала использовать все доступные способы ускорения на чистом Python, без использования дополнительных библиотек. К таковым относится:

- оптимизация вычислений
- уменьшение количества копирования объектов
- переход на константные объекты

Такие изменения позволяют оптимизировать код и значительно ускорить его исполнение, а затем применять дополнительные библиотеки, которые позволяют ещё сильнее ускорить код

6.2 Ускорение с помощью Numba

В данной части будут рассмотрен способ ускорения вычислений с использованием Numba. Для ускорения каждая функция была переписана в соответствии со строгими правилами использования Numba в Python. Все списки были заменены на кортежи, убраны классы сферы и луча, что понизило читаемость кода, но при этом позволило ему запускаться без ошибок. Классы в Numba имеют только экспериментальную поддержку, и основным их ограничением является поддержка только типа float32 для массивов numpy, в связи с этим возникали забавные ошибки из-за недостаточной точности вычисления.

Преимущества:

В качестве преимуществ Numba можно выделить высокую производительность вычислений, при использовании кода на python.

Недостатки:

Numba имеет множество ограничений, вот некоторые из которых пришлось столкнуться при адаптации кода программы:

- ограничения на использование классов,
- ограничения на использования стандартных Python функций (например в print нельзя использовать именные аргументы),

- ограничение на использование списков,

- ограничение на использование типизации аргументов функции (например если функция np.dot(a, b) использовалась аргументами типа float64, её нельзя будет использовать с аргументами типа float32). Также в Numba имеются параметры у @jit, как parallel=True, которые на сложных функциях наоборот замедлят вычисления.

7. Результаты работы

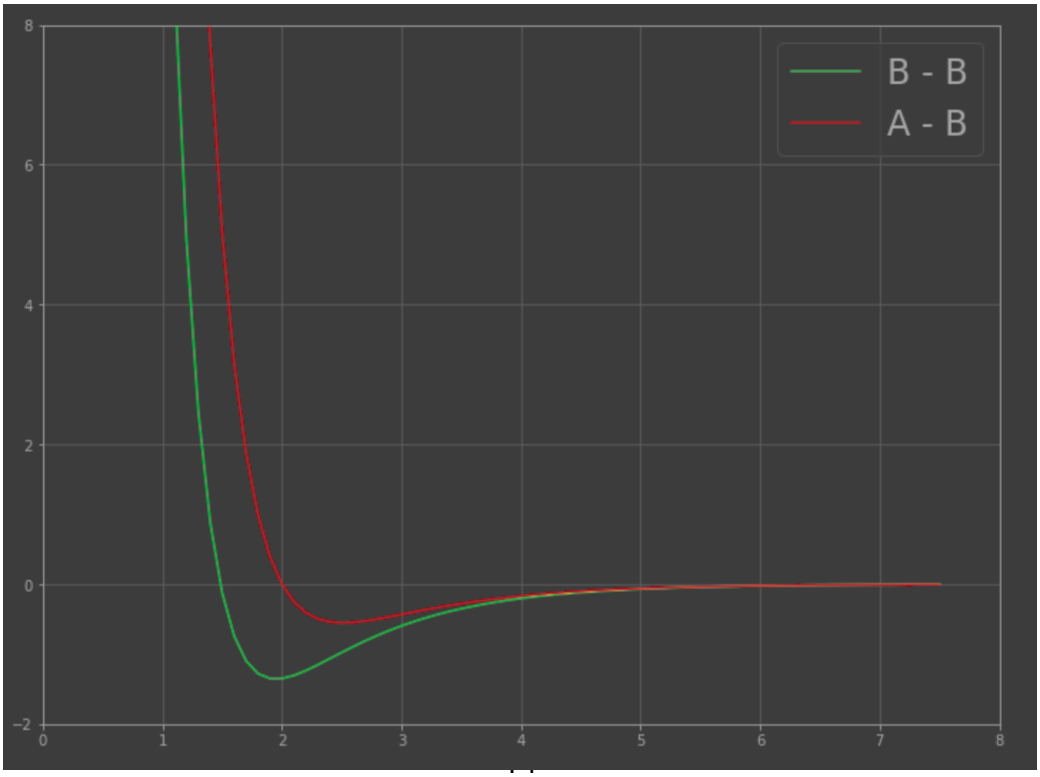
Расчеты проводились для системы A/B(001): Cr/Ag(001).
Параметры потенциалов для взаимодействия B - B, A - B оптимизировались с помощью алгоритма минимизации нулевого порядка Нелдера-Мида.
Для взаимодействия Ag - Ag получены следующие значения:

Переменная	Табличное значение	Полученное значение
E_{coh}	4.085	4.0857
a	-2.960	-2.9603
B	1.08	1.0836
C_{11}	1.32	1.3173
C_{12}	0.97	0.9668
C_{44}	0.51	0.5067

Для взаимодействия Cr - Ag получены следующие значения

Переменная	Табличное значение	Полученное значение
E_{sol}	0.881	0.88098

График зависимости полной энергии от расстояния:



8. Вывод

Выполнение данной курсовой работы помогло разобраться на практике с темой которой был посвящен курс “Параллельной обработки данных в естественнонаучных исследованиях”.

В роли основной задачи в рамках которой было изучены принципы параллельной обработки данных выступала задача определения потенциалов межатомного взаимодействия для системы вида A - B (001) Cr - Ag (001).

Были реализованы все формулы для подсчета характеристик, при условии достижения оптимальной скорости. А также были реализованы два алгоритма: тернарного поиска и алгоритма Нелдера-Мида. В результате были получены положительные результаты для систем A - B и B - B.

9. Листинг

atom.py:

```
import numpy as np
class Atom:
    def __init__(self, _type="Ag", x=1, y=1, z=1):
        self.type = _type
        self.x = x
        self.y = y
        self.z = z

    def __add__(self, other):
        if isinstance(other, Atom):
            return Atom(self.type, self.x + other.x, self.y + other.y, self.z + other.z)

    def __sub__(self, other):
        if isinstance(other, Atom):
            return Atom(self.type, self.x - other.x, self.y - other.y, self.z - other.z)

    def __eq__(self, other):
        if isinstance(other, Atom):
            return self.x == other.x and self.y == other.y and self.z == other.z
        return False

    def __str__(self):
        return f"X: {self.x}\tY: {self.y}\tZ: {self.z}”

    def transform(self, tansformation):
        return Atom(
            self.type,
            self.x * tansformation[0] + self.y * tansformation[1] + self.z *
tansformation[2],
            self.x * tansformation[3] + self.y * tansformation[4] + self.z *
tansformation[5],
            self.x * tansformation[6] + self.y * tansformation[7] + self.z *
tansformation[8]
        )

    def distance(self):
        return np.sqrt(self.x * self.x + self.y * self.y + self.z * self.z)
```

crystal_structure.py

```
import numpy as np
from POD.atom import Atom

class CrystalStructure:
    def __init__(self, dim):
        self.dim = dim
        self.fcc = []
    def face_centred_cubic(self, _type):
        for i in range(self.dim):
            for j in range(self.dim):
                for k in range(self.dim):
                    self.fcc.append(Atom(_type, i, j, k))
                    self.fcc.append(Atom(_type, i, j + 0.5, k + 0.5))
                    self.fcc.append(Atom(_type, i + 0.5, j + 0.5, k))
                    self.fcc.append(Atom(_type, i + 0.5, j, k + 0.5))

    def calculate_energy(self, trans, a1, a0, ksi, p, q, r0, o=None):
        if o is None:
            o = []
        full_energy = 0.
        t = Atom().transform(trans)
        cutoff = 1.7 * max(t.x, max(t.y, t.z))
        for X1 in self.fcc:
            energy_r = 0.
            energy_b = 0.
            for X2 in self.fcc:
                for dx in range(-1, 2):
                    for dy in range(-1, 2):
                        for dz in range(-1, 2):
                            if X1 != X2 or dx != 0 or dy != 0 or dz != 0:
                                rij = ((X2 + Atom('', dx * self.dim, dy * self.dim, dz *
self.dim)).transform(trans) -
                                       X1.transform(trans)).distance()
                                if rij < cutoff:
                                    if X1.type == "Ag" and X2.type == "Ag":
                                        energy_r += ((a1 / r0) * (rij - r0) + a0) * np.exp(-1 * p * (rij /
r0 - 1))
                                        energy_b += ksi * ksi * np.exp(-2 * q * (rij / r0 - 1))
                                    elif X1.type == "Cr" and X2.type == "Ag" or X1.type == "Ag"
and X2.type == "Cr":
                                        energy_r += ((o[0] / o[5]) * (rij - o[5]) + o[1]) *\
                                                np.exp(-1 * o[3] * (rij / o[5] - 1))
                                        energy_b += o[2] * o[2] * np.exp(-2 * o[4] * (rij / o[5] - 1))
```

```
            energy_b = -1 * np.sqrt(energy_b)
            full_energy += energy_r + energy_b
        return full_energy
    def get_size(self):
        return len(self.fcc)
    def change_atom(self, t):
        self.fcc[13].type = t
```

utils.py

```
import numpy as np
from POD.crystal_structure import CrystalStructure
CONST = 0.8018993929636421
ALPHA = 10e-4
a0_true = 4.085
e_coh_true = -2.960
b_true = 1.08
c11_true = 1.32
c12_true = 0.97
c44_true = 0.51
b_pos = np.array([1 + ALPHA, 0, 0, 0, 1 + ALPHA, 0, 0, 0, 1 + ALPHA])
b_neg = np.array([1 - ALPHA, 0, 0, 0, 1 - ALPHA, 0, 0, 0, 1 - ALPHA])
c11_pos = np.array([1 + ALPHA, 0, 0, 0, 1 + ALPHA, 0, 0, 0, 1])
c11_neg = np.array([1 - ALPHA, 0, 0, 0, 1 - ALPHA, 0, 0, 0, 1])
c12_pos = np.array([1 + ALPHA, 0, 0, 0, 1 - ALPHA, 0, 0, 0, 1])
c12_neg = np.array([1 - ALPHA, 0, 0, 0, 1 + ALPHA, 0, 0, 0, 1])
c44_pos = np.array([1, ALPHA, 0, ALPHA, 1, 0, 0, 0, 1 / (1 - ALPHA * ALPHA)])
c44_neg = np.array([1, -ALPHA, 0, -ALPHA, 1, 0, 0, 0, 1 / (1 - ALPHA * ALPHA)])
identity_matrix = np.array([1., 0, 0, 0, 1., 0, 0, 0, 1.])

def get_derivative(dim, positive, negative, energy, a1, a0, ksi, p, q, r0):
    cs = CrystalStructure(dim)
    cs.face_centred_cubic("Ag")
    coh_energy = (
                    cs.calculate_energy(positive, a1, a0, ksi, p, q, r0)
                    + cs.calculate_energy(negative, a1, a0, ksi, p, q, r0)
                ) / cs.get_size()
    return (coh_energy - 2. * energy) / (ALPHA * ALPHA)

def calculate_characteristics(m0, dim, energy, a1, a0, ksi, p, q, r0):
    v0 = m0 * m0 * m0 / 4
    sq_derivative_energy_b = get_derivative(dim, m0 * b_pos, m0 * b_neg, energy,
a1, a0, ksi, p, q, r0)
    sq_derivative_energy_c11 = get_derivative(dim, m0 * c11_pos, m0 * c11_neg,
energy, a1, a0, ksi, p, q, r0)
```

```
    sq_derivative_energy_c12 = get_derivative(dim, m0 * c12_pos, m0 * c12_neg,
energy, a1, a0, ksi, p, q, r0)
    sq_derivative_energy_c44 = get_derivative(dim, m0 * c44_pos, m0 * c44_neg,
energy, a1, a0, ksi, p, q, r0)
    b = 2. * sq_derivative_energy_b * CONST / (9. * v0)
    c11 = (sq_derivative_energy_c11 * CONST + sq_derivative_energy_c12 *
CONST) / (2. * v0)
    c12 = (sq_derivative_energy_c11 * CONST - sq_derivative_energy_c12 *
CONST) / (2. * v0)
    c44 = (sq_derivative_energy_c44 * CONST) / (2. * v0)
    return b, c11, c12, c44
def calculate_energy_sol(e_coh_a, e_coh_b, e_b, e_ab):
    return e_ab - e_b - e_coh_a + e_coh_b
def error_bb(w, dim, a0, e_coh):
    b, c11, c12, c44 = calculate_characteristics(a0, dim, e_coh, w[0], w[1], w[2], w[3],
w[4], w[5])
    err = (a0 - a0_true) * (a0 - a0_true) / (a0_true * a0_true) + \
        (e_coh - e_coh_true) * (e_coh - e_coh_true) / (e_coh_true * e_coh_true) + \
        (b - b_true) * (b - b_true) / (b_true * b_true) + \
        (c11 - c11_true) * (c11 - c11_true) / (c11_true * c11_true) + \
        (c12 - c12_true) * (c12 - c12_true) / (c12_true * c12_true) + \
        (c44 - c44_true) * (c44 - c44_true) / (c44_true * c44_true)
    err = np.sqrt(err / 6)
    return err

def error_ab(w, dim, m0, A1, A0, KSI, P, Q, R0, e_coh_b, e_b):
    e_coh_a = 4.10
    e_true = 0.881
    A_B = CrystalStructure(dim)
    A_B.face_centred_cubic("Ag")
    A_B.change_atom("Cr")
    e_ab = A_B.calculate_energy(m0 * identity_matrix, A1, A0, KSI, P, Q, R0, w)
    e_sol = calculate_energy_sol(e_coh_a, e_coh_b, e_b, e_ab)
    err = (e_sol - e_true) * (e_sol - e_true) / (e_true * e_true)
    return err

def nrg(rij, o):
    energy_r = ((o[0] / o[5]) * (rij - o[5]) + o[1]) * np.exp(-1 * o[3] * (rij / o[5] - 1))
    energy_b = o[2] * o[2] * np.exp(-2 * o[4] * (rij / o[5] - 1))
    energy_b = -1 * np.sqrt(energy_b)
    return energy_r + energy_b
```