**Министерство науки и высшего образования Российской Федерации**

**ФГБОУ ВО «Тверской государственный университет»**

**Математический факультет**

**Кафедра компьютерной безопасности и математических методов управления**

**Специальность 10.05.01 Компьютерная безопасность**

**Курсовая работа по дисциплине**

**Языки программирования**

**«Янус-дендримеры и самоорганизация триблоксополимеров в янус-мицеллы »**

Выполнил:

студент М-24 группы

математического факультета

специальность 10.05.01

Компьютерная безопасность

Павлова Злата Сергеевна

Проверил:

доцент кафедры КБиММУ

Шаповалова И.А.

Тверь, 2020

Содержание

Введение 3

1. Модель и метод4
   1. Подсчет объемов ячеек, площади контактов ячеек, вероятности перехода из ячейки в ячейку4
   2. Нелинейная оптимизация. Квазиньютоновский метод Дэвидона — Флетчера — Пауэлла7
   3. Моделирование ансамбля привитых цепей на решетке численным методом самосогласованного поля Схойтенса-Флира 8
   4. Численная оптимизация структурных свойств янус-подобных молекулярных щеток 10
2. Код программы12
3. Результаты 32

Список литературы 33

Введение

Янус-частицы - двухкомпонентные наночастицы, которые характеризуются постоянством состава, возможностью самосборки в сложные структуры, а также возможностью сочетать разные физико-химические свойства в одной частице. Для частиц-янусов существует возможность дальнейшей химической модификации их отдельных частей, что резко расширяет область их практического использования, в том числе, в биологии и медицине – для доставки лекарств, медицинской визуализации пораженных органов, фотодинамической и магнитотермической терапии онкологических заболеваний и пр.

В данной курсовой работе передо мной стояла задача численно исследовать янус-системы, внутри которых из-за специфических взаимодействий может происходить деление. Ветви макромолекулярной щетки, которые состоят из разного химического строения, могут стремиться к отталкиванию, что в конечном итоге приведет к образованию двуликой(янус) молекулы.

Таким образом, цель курсовой работы состоит в том, чтобы написать программу, которая при помощи определенных методов позволит визуализировать поведение двух отталкивающихся молекул в полимерной щетке.

Код выполнен на языке С++ с применением ООП. Преимущества такого метода программирования состоит в том, что он предоставляет возможность создавать объекты со своими собственными свойствами и поведением. Объектно-ориентированное программирование также имеет несколько других полезных концепций: наследование, инкапсуляция, абстракция и полиморфизм. Эти функции и методы придают коду универсальность и позволяют легко интегрировать объекты в новый проект.

1. Модель и метод
   1. Подсчет объемов ячеек, площади контактов ячеек,

вероятности перехода из ячейки в ячейку

Пользователь выбирает один из вариантов двухградиентной системы и получает на выходе вероятности переходов из слоя в слой и объемы ячеек.

Множители … - вероятности перехода из верхнего, текущего и нижнего слоя в текущий.

Полярная двухградиентная геометрия:

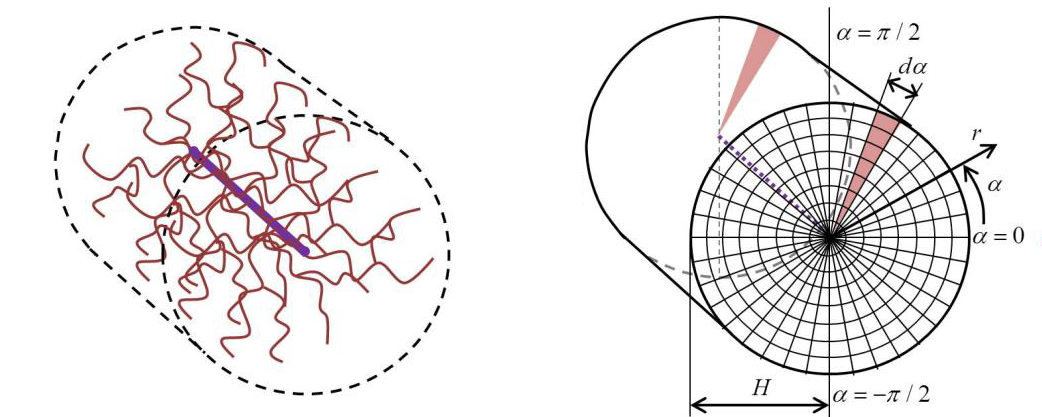
,

,

,

,

где - высота цилиндра, -центральный угол сектора

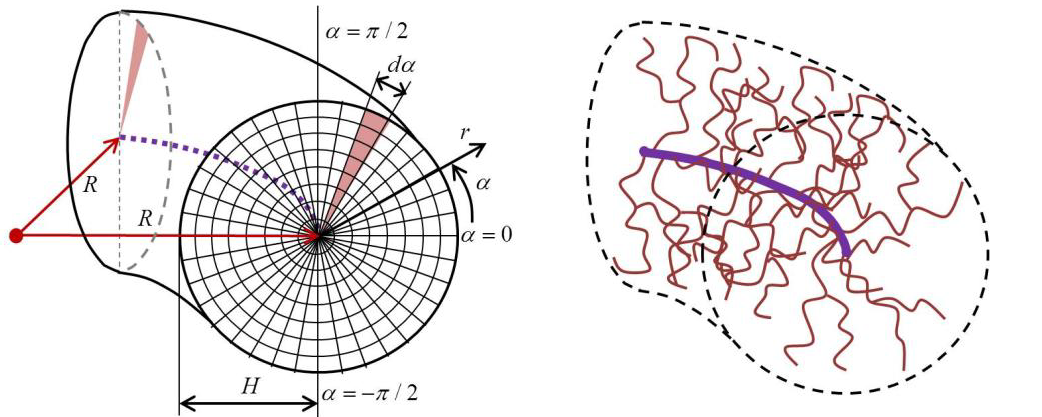


*рис.1- Цилиндр*

Таким образом, вероятности перехода:

Тороидальная двухградиентная геометрия:

,



*рис. 2 – Тор*

Таким образом, вероятности перехода:

* 1. Нелинейная оптимизация. Квазиньютоновский метод Дэвидона — Флетчера — Пауэлла.

Прежде чем приступить к анализу полимерной щетки, необходимо оптимизировать физические процессы: аппроксимировать набор экспериментальных точек, найти оптимальные параметры для управления системой. Эти действия помогут рассчитать градиент минимизируемого функционала, который потребуется для моделирования привитых цепей численным методом самосогласованного поля Схойтенса-Флира.

Алгоритм метода:

Выбирается точка и матрица . Обычно берут единичную матрицу

Далее начинается циклический процесс и подсчет итераций (k = 1,2…):

1. Вычисляется градиент целевой функции в точке
2. Одномерный поиск:

,

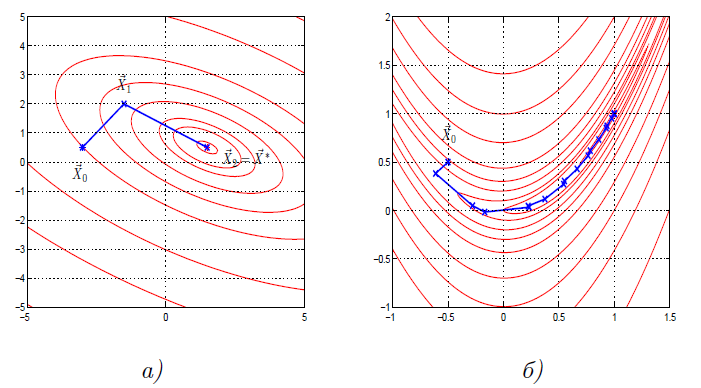
.

1. Обновление матрицы

,

Для выпуклой неквадратичной функции при восстанавливается матрица, обратная матрице Гессе:

,



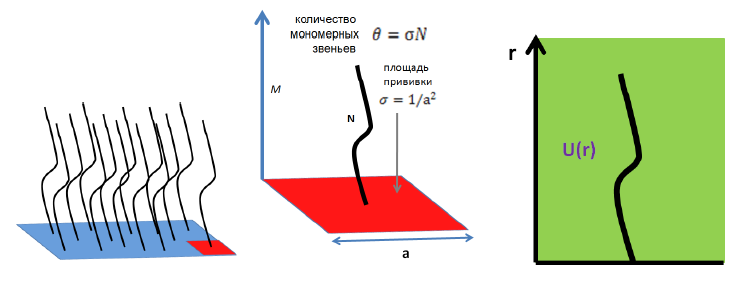
*рис. 3 - Траектория поиска минимума квадратичной функции (a) и функции Розенброка (б) методом ДФП.*

* 1. Моделирование ансамбля привитых цепей на

решетке численным методом самосогласованного

поля Схойтенса-Флира

Основная идея теории самосогласованного поля заключается в изучении поведения сложных систем через исследование простых моделей. Следовательно, в данной задаче используется одноградиентный метод, то есть вместо трехмерного пространства рассматривается одномерное направление от поверхности прививки цепей.



*Рис. 4 а) б) в)*

*Рис.4 - (а) Ансамбль привитых цепей, б) Отдельно взятая цепь, в) Цепь*

*под действием внешнего поля*

Условные обозначения:

num\_atoms - длина цепи, число шариков

𝜎 (theta) - плотность прививки цепей (1.0/площадь под одну цепь)

- число слоев по х

- число слоев по у

= 3M = 3()() – размерность задачи

𝜂 (nu) - скорость (шаг сходимости алгоритма (0 < 𝜂 ≤ 1)

𝛿 (tolerance) - точность метода сходимости

𝑚𝑎𝑥\_𝑠𝑡𝑒𝑝 - максимальное число шагов в алгоритме

– количество итераций

𝜃 (sigma)= 𝜎\*num\_atoms - число звеньев на площадь прививки

xmin - нижняя граница прививки по 𝑥

xmax - верхняя граница прививки по 𝑥

ymin - нижняя граница прививки по 𝑦

ymax - верхняя граница прививки по 𝑦

В каждой ячейке могут располагаться мономерные звенья двух сортов: 𝑝 - полимер, solw - растворитель. В конечном счете необходимо получить распределение плотности полимера, чтобы понять, как он будет располагаться в заданной системе.

Алгоритм метода:

Используем метод оптимизации DFP для поиска состояния системы с минимальной энергией F и максимальным числом max\_step:

1. Рассчитываем распределение Больцмана стат. весов 𝐺𝑝 (полимера),

то есть вероятностей нахождения мономеров в слое:

и стат. весов растворителя:

1. Пропагаторы для расчета вероятностей нахождения шариков цепи относительно поверхности прививки.

Прямой пропагатор - начинаем блуждание с привитого мономера:

Обратный пропагатор - начинаем блуждание с концевого мономера:

Учитываем граничные условия: периодические значения, наверху - зеркальные и внизу – стенка.

1. Рассчитываем стат. сумму цепи:
2. Рассчитываем профили плотности полимера и растворителя:

,

1. Рассчитываем градиент минимизируемого функционала

,

1. Применяем алгоритм DFP для расчета новых потенциалов

,

где А - квазиньютоновская матрица, полученная методом DFP

1. Проверяем условия остановки для достижения заданной точности

err < 𝛿, где отклонение от решения оценивается через длину вектора градиента g на текущем шаге:

.

* 1. Численная оптимизация структурных свойств

янус-подобных молекулярных щеток

Рассмотрим частный случай с двумя молекулами полимера. Соответственно, теперь задача сводится к поиску профилей плотности двух молекул, а также необходимо при помощи параметра Флори (>0) визуализировать распределение молекул по всей системе координат, так как он отвечает за степень отталкивания двух молекул и за взаимодействие между полимером и растворителе (χ). Тогда при нулевом параметре происходит равномерная сегрегация, а при его увеличение можно наблюдать, что каждый вид молекулы будет предпочитать определенный сектор системы.

Для начала необходимо проанализировать молекулы: определить число поколений (G), длину линейного блока (), плотность прививки (𝜎) и параметр Флори (χ). Тогда длина цепи рассчитывается по формуле:

.

Таким образом, методом самосогласованного поля Схойтенса-Флира находим распределение Больцмана, прямой и обратный пропагатор, статическую сумму цепи и профили плотности полимера для каждой молекулы и растворителя.

Тогда суммарная плотность:

Рассчитываем плотность окружения для молекул и растворителя, учитывая вероятности перехода из слоя в слой:

где m =

Рассчитываем множители Лагранжа:

Таким образом, принимая во внимание множители Лагранжа, рассчитываем градиент:

для первых M координат:

для вторых М координат:

для третьих М координат:

где среднее поле Лагранжа:

1. Код программы

Реализуем классы:

* для двухградиентной геометрии (class Geometry) с дочерними классами видов геометрий (class Polar, class Torus),
* для молекул (class Molecule),
* для нахождения градиента (class BaseOptimTools) с дочерними классами градиентного спуска (class Gradient) и методом DFP (class DFP)
* для считывания значений и реализации методов оптимизации (class System).

geometry. h

const double pi = 3.14159265359;

class Geometry {

public:

mutable int Mx, My;//число слоев по х, y

double R0, delta\_alpha;//кривизна, угол

void GetValue(int Mx\_, int My\_, double R0\_);//метод для считывания параметров

Geometry() {};//конструктор класса

vector < vector < double > >lambda\_bb;

vector < vector < double > >lambda\_bn;

vector < vector < double > >lambda\_bf;

vector < vector < double > >lambda\_fb;

vector < vector < double > >lambda\_fn;

vector < vector < double > >lambda\_ff;

vector < vector < double > >lambda\_nb;

vector < vector < double > >lambda\_nn;

vector < vector < double > >lambda\_nf;

vector < vector < double > >volume;

vector < vector < double > >square\_side;

vector < vector < double > >square\_up;

vector < vector < double > >square\_front;

vector < vector < double > >square\_right;

vector < vector < double > >square\_left;

virtual void UpdateVolume() const {}; //метод нахождения объема

virtual void UpdateSquareFront() const {};//метод нахождения площади фронтальной стороны

virtual void UpdateSquareUp() const {};//метод нахождения площади верхней стороны

virtual void UpdateSquareLeft() const {};//метод нахождения площади левой стороны

virtual void UpdateSquareRight() const {};//метод нахождения площади правой стороны

virtual void UpdateSquareSide() const {};//метод нахождения площади боковой стороны

virtual void Transposition() const {};//метод нахождения вероятностей перехода

void AllocateMemory();//инициализация данных

~Geometry() {};//деконструктор класса

};

class Polar : public Geometry {

public:

int delta\_h = 1;

void UpdateVolume();

void UpdateSquareFront();

void UpdateSquareUp();

void UpdateSquareSide();

void Transposition();

};

class Torus : public Geometry {

public:

void UpdateVolume();

void UpdateSquareFront();

void UpdateSquareUp();

void UpdateSquareLeft();

void UpdateSquareRight();

void Transposition();};

geometry.cpp

#include "geometry.h"

void Geometry::AllocateMemory() {

lambda\_bb.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_bn.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_bf.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_fb.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_fn.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_ff.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_nb.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_nn.assign(Mx + 2, vector<double>(My + 2, 0));

lambda\_nf.assign(Mx + 2, vector<double>(My + 2, 0));

volume.assign(Mx + 2, vector<double>(My + 2, 0));;

square\_side.assign(Mx + 2, vector<double>(My + 2, 0));

square\_up.assign(Mx + 2, vector<double>(My + 2, 0));

square\_front.assign(Mx + 2, vector<double>(My + 2, 0));

square\_right.assign(Mx + 2, vector<double>(My + 2, 0));;

square\_left.assign(Mx + 2, vector<double>(My + 2, 0));;

};

void Geometry::GetValue(int Mx\_, int My\_, double R0\_) {

Mx = Mx\_;

My = My\_;

R0 = R0\_;

double delta\_alpha = pi \* 2 / My ;

}

void Polar::Transposition() {

for (int i = 1; i < Mx + 1; ++i) {

for (int j = 1; j < My + 1; ++j) {

lambda\_bb[i][j] = 0.0;

lambda\_bn[i][j] = square\_front[i - 1][j] / volume[i][j] \* 1.0 / 6.0;

lambda\_bf[i][j] = 0.0;

lambda\_fb[i][j] = 0.0;

lambda\_fn[i][j] = square\_front[i][j] / volume[i][j] \* 1.0 / 6.0;

lambda\_ff[i][j] = 0.0;

double prob = 1.0 - lambda\_bn[i][j] - lambda\_fn[i][j] - 1.0 / 6.0 \* square\_front[i][j] / volume[i][j] - 1.0 / 6.0 \* square\_front[i - 1][j] / volume[i][j];

double sum = 2.0 \* (square\_side[i][j] + volume[i][j]);

lambda\_nb[i][j] = prob \* square\_side[i][j] / sum;

lambda\_nn[i][j] = prob \* 2.0 \* volume[i][j] / sum;

lambda\_nf[i][j] = prob \* square\_side[i][j] / sum;

}

}

}

void Polar::UpdateSquareFront() {

for (int i = 1; i < Mx + 1; ++i) {

for (int j = 1; j < My + 1; ++j) {

square\_front[i][j] = 0.5\*My\*(i \*i - (i - 1) \* (i - 1));

}

}

}

void Polar::UpdateSquareSide() {

for (int i = 1; i < Mx + 1; ++i) {

for (int j = 1; j < My + 1; ++j) {

square\_side[i][j] = delta\_h \* (i \*i - (i - 1) \* (i - 1));

}

}

}

void Polar::UpdateSquareUp() {

for (int i = 1; i < Mx + 1; ++i) {

for (int j = 1; j < My + 1; ++j) {

square\_up[i][j] = delta\_h \* i \* My;

}

}

}

void Polar::UpdateVolume() {

for (int i = 1; i < Mx + 1; ++i) {

for (int j = 1; j < My + 1; ++j) {

volume[i][j] = 0.5\*My\*(i \*i - (i - 1) \* (i - 1))\*delta\_h;

}

}

}

void Torus::UpdateSquareFront() {

for (int i = 1; i < Mx + 1; i++)

for (int j = 1; j < My + 1; j++) {

square\_front[i][j] = i \* delta\_alpha \* 2.0 \* pi\*(R0 + i \* cos((2 \* j - 1)\*delta\_alpha) / 2.0);

}

}

void Torus::UpdateVolume() {

vector <double> rho(Mx + 1, 0);

for (int i = 1; i < Mx + 1; ++i) {

rho[i] = 0.67\*(2 \* i - 1);

}

for (int i = 1; i < Mx + 1; i++) {

for (int j = 1; j < My + 1; j++) {

volume[i][j] = square\_up[i][j] \* pi \* 2.0\*(R0 + rho[i] \* cos(((2 \* j - 1) / 2.0)\*delta\_alpha));}}};

void Torus::UpdateSquareUp() {

for (int i = 1; i < Mx + 1; i++)

for (int j = 1; j < My + 1; j++) {

square\_up[i][j] = delta\_alpha / 2.0\*(i \*i - (i - 1) \* (i - 1));

}};

void Torus::UpdateSquareRight() {

for (int i = 1; i < Mx + 1; i++)

for (int j = 1; j < My + 1; j++) {

square\_right[i][j] = (i - (i - 1)) \* 2.0 \* pi\*(R0 + ((2 \* i - 1) / 2.0 \* cos(j\*delta\_alpha)));

}

};

void Torus::UpdateSquareLeft() {

for (int i = 1; i < Mx + 1; i++)

for (int j = 1; j < My + 1; j++) {

square\_left[i][j] = (i - (i - 1)) \* 2.0 \* pi\*(R0 + ((2 \* i - 1) / 2.0\*cos((j - 1)\*delta\_alpha)));

}

};

void Torus::Transposition() {

for (int i = 1; i < Mx + 1; ++i) {

for (int j = 1; j < My + 1; ++j) {

lambda\_bb[i][j] = 0;

lambda\_bn[i][j] = square\_front[i - 1][j] / volume[i][j] \* 1.0 / 6.0;

lambda\_bf[i][j] = 0.0;

lambda\_fb[i][j] = 0.0;

lambda\_fn[i][j] = square\_front[i][j] / volume[i][j] \* 1.0 / 6.0;

lambda\_ff[i][j] = 0.0;

double prob = 1.0 - 1.0 / 6.0 \* square\_front[i - 1][j] / volume[i][j] - 1.0 / 6.0 \* square\_front[i - 1][j] / volume[i][j];

double sum = square\_left[i][j] + square\_right[i][j] + 2 \* volume[i][j];

lambda\_nb[i][j] = prob \* square\_left[i][j] / sum;

lambda\_nn[i][j] = prob \* 2.0 \* volume[i][j] / sum;

lambda\_nf[i][j] = prob \* square\_right[i][j] / sum;

}

}

};

molecule.h

#include "geometry.h"

using vector3d = vector<vector<vector<double>>>;

class Molecule {

friend class System;

public:

double theta, sigma, chi;

int layers\_x, layers\_y, xmin, xmax, ymin, ymax, num\_generation, num\_atoms, ns;

void SetParameters();//метод считывания параметров молекулы

void AllocateMemory(int layers\_x\_, int layers\_y\_, int M);//метод инициализации данных

double q;

vector3d Gback;

vector3d Gforw;

vector <double> u;

vector <vector <double>> G;

vector <double> fi\_side;

vector<vector<double>> fi\_p;

vector<vector<double>> multipliers;

void FindG();//метод для нахождения распределения Больцмана

void FindGforw(Geometry geo);//метод нахождения прямого пропагатора

void FindGback(Geometry geo);//метод нахождения обратного пропагатора

void FindQ(Geometry geo);//метод нахождения статической суммы

void FindFiP();//метод нахождения профилей плотности молекулы

void FindFiSide(Geometry geo);//метод нахождения профиля плотности окружения

};

molecule.cpp

#include "molecule.h"

void Molecule::SetParameters() {

num\_atoms = 1 + ns \* (pow(2, num\_generation + 1) - 1);

theta = sigma \* num\_atoms;

};

void Molecule::AllocateMemory(int layers\_x\_, int layers\_y\_, int M) {

layers\_x = layers\_x\_;

layers\_y = layers\_y\_;

Gback.assign(layers\_x + 2, vector <vector<double>>(layers\_y + 2, vector<double>(num\_atoms + 2, 0)));

Gforw.assign(layers\_x + 2, vector <vector<double>>(layers\_y + 2, vector<double>(num\_atoms + 2, 0)));;

u.resize(M+2,0);

G.assign(layers\_x + 2, vector<double>(layers\_y + 2, 0));

fi\_side.resize(M+2,0);

fi\_p.assign(layers\_x + 2, vector<double>(layers\_y + 2, 0));

multipliers.assign(layers\_x + 2, vector <double>(layers\_y + 2, 0));

};

void Molecule::FindG() {

for (int i = 0; i < layers\_x + 1; ++i) {

for (int j = 0; j < layers\_y + 1; ++j) {

G[i][j] = exp(-u[i\*(layers\_y + 2) + j]);

}

}

};

void Molecule::FindGforw(Geometry geo) {

for (int j = 1; j < num\_atoms; ++j) {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int k = 1; k < layers\_y + 1; ++k) {

Gforw[i][k][j] = G[i][k] \* (geo.lambda\_bb[i][k] \* Gforw[i - 1][k - 1][j - 1] + geo.lambda\_bn[i][k] \* Gforw[i - 1][k][j - 1] + geo.lambda\_bf[i][k] \* Gforw[i - 1][k + 1][j - 1] +

geo.lambda\_nb[i][k] \* Gforw[i][k - 1][j - 1] + geo.lambda\_nn[i][k] \* Gforw[i][k][j - 1] + geo.lambda\_nf[i][k] \* Gforw[i][k + 1][j - 1] +

geo.lambda\_fb[i][k] \* Gforw[i + 1][k - 1][j - 1] + geo.lambda\_fn[i][k] \* Gforw[i + 1][k + 1][j - 1] + geo.lambda\_ff[i][k] \* Gforw[i + 1][k + 1][j - 1]);

//периодические граничные условия

Gforw[layers\_x + 1][k][j] = Gforw[layers\_x][k][j];

Gforw[0][k][j] = 0;

}

//зеркальные граничные условия + стенка

Gforw[i][layers\_y + 1][j] = Gforw[i][1][j];

Gforw[i][0][j] = Gforw[i][layers\_y][j];

}

}

};

void Molecule::FindGback(Geometry geo) {

for (int j = num\_atoms - 1; j > 0; --j) {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int k = 1; k < layers\_y + 1; ++k) {

Gback[i][k][j - 1] = G[i][k] \* (geo.lambda\_bb[i][k] \* Gback[i - 1][k - 1][j] + geo.lambda\_bn[i][k] \* Gback[i - 1][k][j] + geo.lambda\_bf[i][k] \* Gback[i - 1][k + 1][j] +

geo.lambda\_nb[i][k] \* Gback[i][k - 1][j] + geo.lambda\_nn[i][k] \* Gback[i][k][j] + geo.lambda\_nf[i][k] \* Gback[i][k + 1][j] +

geo.lambda\_fb[i][k] \* Gback[i + 1][k - 1][j] + geo.lambda\_fn[i][k] \* Gback[i + 1][k][j] + geo.lambda\_ff[i][k] \* Gback[i + 1][k + 1][j]);

//периодические граничные условия

Gback[layers\_x + 1][k][j - 1] = Gback[layers\_x][k][j - 1];

Gback[0][k][j - 1] = 0;

}

//зеркальные граничные условия + стенка

Gback[i][layers\_y + 1][j - 1] = Gback[i][1][j - 1];

Gback[i][0][j - 1] = Gback[i][layers\_y][j - 1];

}

}

};

void Molecule::FindQ(Geometry geo) {

double sum = 0;

q = 0;

for (int i = 1; i < layers\_x + 1; ++i) {

for (int k = 1; k < layers\_y + 1; ++k) {

sum = 0;

for (int j = 0; j < num\_atoms; ++j)

sum += (Gforw[i][k][j] \* Gback[i][k][j]) / G[i][k];

q += sum \* geo.volume[i][k];

}

}

};

void Molecule::FindFiP() {

double sum;

for (int i = 1; i < layers\_x + 1; ++i) {

for (int k = 1; k < layers\_y + 1; ++k) {

sum = 0;

for (int j = 0; j < num\_atoms; ++j)

sum += (Gforw[i][k][j] \* Gback[i][k][j]) / G[i][k];

fi\_p[i][k] = (theta / q)\*sum;

}

}

//периодические граничные условия

for (int i = 1; i < layers\_x + 1; ++i) {

fi\_p[i][layers\_y + 1] = fi\_p[i][1];

fi\_p[i][0] = fi\_p[i][layers\_y];

}

//зеркальные граничные условия + стенка

for (int k = 1; k < layers\_y + 1; ++k) {

fi\_p[layers\_x + 1][k] = fi\_p[layers\_x][k];

fi\_p[0][k] = 0;

}

};

void Molecule::FindFiSide(Geometry geo) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k) {

fi\_side[l \* (layers\_y + 2) + k] = 0.0;

}

}

for (int i = -1; i <= 1; ++i) {

for (int j = -1; j <= 1; ++j) {

if (i == j == -1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_bb[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == -1 && j == 0) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_bn[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == -1 && j == 1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_bf[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == 0 && j == -1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_nb[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == 0 && j == 0) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_nn[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == 0 && j == 1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_nf[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == 1 && j == -1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_fb[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == 1 && j == 0) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_fn[l][k] \* fi\_p[l + i][k + j];

}

}

else if (i == 1 && j == 1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_ff[l][k] \* fi\_p[l + i][k + j];

}

}

}

}

}

method.h

class BaseOptimTools

{

public:

int Mx, My, M;

int num\_iter;

double tolerance, length\_of\_grad, nu;

string name = "OptimTools";

vector <double> direction;

virtual double SetGradFirst(vector <double> grad) { double q = 0.0; return q; };

virtual double SetGradRegular(vector <double> grad) { double q = 0.0; return q; };

virtual void UpdateX(vector <double> &x, vector <double> grad) {};

void SetParameters(int Mx, int My, vector <double> grad, vector <double> u);//метод инициализации данных

virtual ~BaseOptimTools() {};

};

class Gradient : public BaseOptimTools

{

public:

double SetGradFirst(vector <double> grad) override;

double SetGradRegular(vector <double> grad) override;

void UpdateX(vector <double> &x, vector <double> grad) override;

Gradient(int \_num\_iter, double \_tolerance, double \_nu);

};

class DFP : public BaseOptimTools

{

public:

double SetGradFirst(vector <double> grad) override;

double SetGradRegular(vector <double> grad) override;

void UpdateX(vector <double> &x, vector <double> grad) override;

DFP(int \_num\_iter, double \_tolerance, double \_nu);

vector <double> alpha;

vector <double> beta;

vector < vector < double > > A;

private:

vector < vector < double > > Formula();//метод нахождения обратной матрицы Гессе

void SingularMatrix();//метод инициализации единичной матрицы

vector<double> FindDirection(vector<double>grad);//метод нахождения направления вектора

vector < vector < double > > Multiply\_matrixes(vector < vector < double > >, vector < vector < double > >);//метод перемножения двух матриц

vector < vector < double > > Multiply\_vectors(vector<double>, vector<double>);// метод перемножения вектора и транспонированного вектора

double Find\_number(vector<double>, vector<double>);// метод перемножения двух векторов

vector < double > Multiply\_matrix\_by\_vector(vector < vector < double > >, vector<double>);//метод перемножения матрицы и вектора

vector < vector < double > > Division\_matrix\_on\_number(vector < vector < double >>, double);//метод деления матрицы на число

};

method.cpp

#include "method.h"

void BaseOptimTools::SetParameters(int Mx, int My, vector <double> grad, vector <double> u) {

u.resize(3 \* (Mx + 2)\*(My + 2),0);

grad.resize(3 \* (Mx + 2)\*(My + 2),0);

M = grad.size();

direction.resize(M + 2, 0);

};

Gradient::Gradient(int \_num\_iter, double \_tolerance, double \_nu) {

name = "Gradient";

num\_iter = \_num\_iter;

tolerance = \_tolerance;

nu = \_nu;

}

double Gradient::SetGradFirst(vector <double> grad) {

double lenght=0.0;

for (int i = 0; i < M; i++)

lenght += grad[i]\*grad[i];

return sqrt(lenght);

};

double Gradient::SetGradRegular(vector <double> grad) {

double lenght = 0.0;

for (int i = 0; i < M; i++)

lenght += grad[i] \* grad[i];

return sqrt(lenght);};

void Gradient::UpdateX(vector <double> &u, vector <double> grad) {

for (int i = 0; i < M; i++)

u[i] = u[i] - nu \* grad[i];

};

DFP::DFP(int \_num\_iter, double \_tolerance, double \_nu) {

name = "DFP";

num\_iter = \_num\_iter;

tolerance = \_tolerance;

nu = \_nu;

}

double DFP::SetGradFirst(vector <double> grad) {

alpha.resize(M+2,0);

beta.resize(M+2,0);

A.assign(M, vector<double>(M));

SingularMatrix();

double lenght = 0.0;

for (int i = 0; i < M; i++)

lenght += grad[i] \* grad[i];

return sqrt(lenght);

};

double DFP::SetGradRegular(vector <double> grad) {

for (int i = 0; i < M; i++)

beta[i] = grad[i] - beta[i];

A = Formula();

double lenght = 0.0;

for (int i = 0; i < M; i++)

lenght += grad[i] \* grad[i];

return sqrt(lenght);

};

void DFP::UpdateX(vector <double> &u, vector <double> grad) {

direction = FindDirection(grad);

for (int i = 0; i < M; i++) {

alpha[i] = u[i];

}

for (int i = 0; i < M; i++) {

u[i] = u[i] + nu \* direction[i];

}

cout << endl;

for (int i = 0; i < M; i++)

alpha[i] = u[i] - alpha[i];

for (int i = 0; i < M; i++)

beta[i] = grad[i];

};

void DFP::SingularMatrix() {

for (int i = 0; i < M; ++i) {

for (int j = 0; j < M; ++j) {

if (i == j)

A[i][j] = 1;

else A[i][j] = 0;

}

}

};

vector<double> DFP::FindDirection(vector<double>grad) {

direction = Multiply\_matrix\_by\_vector(A, grad);

for (int i = 0; i < M; i++) {

direction[i] = -1 \* direction[i];

}

return direction;

};

vector < vector < double > > DFP::Formula() {

double number1 = 0, number2 = 0;

vector < vector < double > > matrix1(M, vector<double>(M, 0));

vector < vector < double > > matrix2(M, vector<double>(M, 0));

vector < vector < double > > B(M, vector<double>(M, 0));

vector < vector < double > > C(M, vector<double>(M, 0));

vector <double> a(M, 0);

vector <double> b(M, 0);

matrix1 = Multiply\_vectors(alpha, alpha);

number1 = Find\_number(alpha, beta);

B = Division\_matrix\_on\_number(matrix1, number1);

a = Multiply\_matrix\_by\_vector(A, beta);

matrix2 = Multiply\_vectors(a, beta);

matrix2 = Multiply\_matrixes(A, matrix1);

b = Multiply\_matrix\_by\_vector(A, beta);

number2 = Find\_number(b, beta);

C = Division\_matrix\_on\_number(matrix2, number2);

for (int i = 0; i < M; i++) {

for (int j = 0; j < M; j++)

A[i][j] = A[i][j] + B[i][j] - C[i][j];

}

return A;

};

vector < vector < double > > DFP::Multiply\_matrixes(vector < vector < double > > A, vector < vector < double > > U) {

vector < vector < double > > c(M, vector<double>(M, 0));

for (int i = 0; i < M; i++) {

for (int j = 0; j < M; j++) {

c[i][j] = 0;

for (int t = 0; t < M; t++)

c[i][j] += A[i][t] \* U[t][j];

}

}

return c;

};

vector < vector < double > > DFP::Multiply\_vectors(vector<double> a, vector<double> b) {

vector < vector < double > > matrix(M, vector<double>(M, 0));

for (int i = 0; i < M; i++) {

for (int j = 0; j < M; j++)

matrix[i][j] = a[i] \* b[j];

}

return matrix;

};

double DFP::Find\_number(vector<double> a, vector<double> b) {

double number = 0;

for (int i = 0; i < M; i++)

number += a[i] \* b[i];

return number;

};

vector < double > DFP::Multiply\_matrix\_by\_vector(vector<vector<double>>A,

vector<double>grad) {

vector <double> a(M, 0);

for (int i = 0; i < M; i++) {

a[i] = 0;

for (int j = 0; j < M; j++) {

a[i] += A[i][j] \* grad[j];

}

}

return a;

};

vector < vector < double > > DFP::Division\_matrix\_on\_number(vector<vector<double>>A, double b) {

vector < vector < double > > matrix1(M, vector<double>(M, 0));

for (int i = 0; i < M; i++) {

for (int j = 0; j < M; j++)

matrix1[i][j] = A[i][j] / b;

}

return matrix1;

};

system.h

#include "geometry.h"

#include "molecule.h"

#include "method.h"

using vector3d = vector<vector<vector<double>>>;

template<typename T> struct Box : std::unique\_ptr<T> {

using std::unique\_ptr<T>::unique\_ptr;

operator T &() const { return \*\*this; }};

class System

{

public:

string source;

vector <double> u;

vector <double> grad;

void ReadParameters();//метод для считывания параметров системы

int layers\_x;

int layers\_y;

int M;

double Rcurv, chi\_seg;

string geometry\_name;

System(string \_source);//конструктор класса

Geometry geo;//объявление объекта типа геометрии

void SetGeometry();//метод для считывания параметров геометрии

vector<Molecule> mol;//массив молекул

void ReadMolecules();//метод для считывания параметров молекул

vector<Box<BaseOptimTools>> methods; //шаблон для реализации методов

void ReadMethods();//метод для считывания

void Function();// метод самосогласованного поля

void Cycling();// метод поиска градиента

void Output();//метод вывода в файл

private:

void AllocateMemory();//метод инициализация данных

void FindFiSolv();//метод расчета профиля плотности растворителя

void FindFiTotal();//метод расчета общего профиля плотности

void FindFiSide();//метод расчета профиля плотности окружности для растворителя

void FindLagrangeMultipliers(int t);//метод нахождения множителей Лагранжа для молекул

void FindLagrangeMultipliers();//метод нахождения множителей Лагранжа для растворителя

void FindGrad();//метод расчета градиента

vector <double> fi\_side;

vector < vector < double > > fi\_solv;

vector < vector < double > > fi\_total;

vector <vector<double>> multipliers;

vector <vector<double>> middle\_multipliers;

};

system.cpp

#include "geometry.h"

#include "molecule.h"

#include "method.h"

#include "system.h"

System::System(string \_source) {

source = \_source;

}

void System::ReadParameters() {

ifstream ifs(source, ifstream::in);

string word;

while (ifs.good()) {

ifs >> word;

if (word.find("geometry") != string::npos) ifs >> geometry\_name;

if (word.find("layers\_x") != string::npos) ifs >> layers\_x;

if (word.find("layers\_y") != string::npos) ifs >> layers\_y;

if (word.find("curvature") != string::npos) ifs >> Rcurv;

if (word.find("chi\_seg") != string::npos) ifs >> chi\_seg;

}

ifs.close();}

void System::SetGeometry() {

if (geometry\_name == "polar") {

Polar polar;

polar.GetValue(layers\_x, layers\_y, Rcurv);

polar.AllocateMemory();

polar.UpdateVolume();

polar.UpdateSquareFront();

polar.UpdateSquareUp();

polar.UpdateSquareSide();

polar.Transposition();

geo = polar;

}

else if (geometry\_name == "torus") {

Torus torus;

torus.GetValue(layers\_x, layers\_y, Rcurv);

torus.AllocateMemory();

torus.UpdateVolume();

torus.UpdateSquareFront();

torus.UpdateSquareUp();

torus.UpdateSquareLeft();

torus.UpdateSquareRight();

torus.Transposition();

geo = torus;

}

else {

printf("ERROR: unknown geometry\n");

return;

}

}

void System::ReadMolecules() {

ifstream ifs(source, ifstream::in);

string word;

while (ifs.good()) {

ifs >> word;

if (word.find("molecule") != string::npos) {

Molecule new\_mol;

while (word != "[" && ifs.good()) {

ifs >> word;

if (word.find("Ns") != string::npos) ifs >> new\_mol.ns;

if (word.find("gen") != string::npos) ifs >> new\_mol.num\_generation;

if (word.find("sigma") != string::npos) ifs >> new\_mol.sigma;

if (word.find("chi") != string::npos) ifs >> new\_mol.chi;

if (word.find("xmin") != string::npos) ifs >> new\_mol.xmin;

if (word.find("xmax") != string::npos) ifs >> new\_mol.xmax;

if (word.find("ymin") != string::npos) ifs >> new\_mol.ymin;

if (word.find("ymax") != string::npos) ifs >> new\_mol.ymax;

}

new\_mol.SetParameters();

mol.push\_back(new\_mol);

}

}

ifs.close();

};

void System::ReadMethods() {

ifstream ifs(source, ifstream::in);

string word;

while (ifs.good()) {

ifs >> word;

if (word.find("method") != string::npos) {

double tolerance\_, nu\_, \_num\_iter;

string name\_method;

while (word != "[" && ifs.good()) {

ifs >> word;

if (word.find("type") != string::npos) ifs >> name\_method;

if (word.find("tolerance") != string::npos) ifs >> tolerance\_;

if (word.find("num\_iter") != string::npos) ifs >> \_num\_iter;

if (word.find("step") != string::npos) ifs >> nu\_;

}

if (name\_method == "gradient") {

methods.push\_back(make\_unique<Gradient>(\_num\_iter, tolerance\_, nu\_));

}

else if (name\_method == "DFP") {

methods.push\_back(make\_unique<DFP>(\_num\_iter, tolerance\_, nu\_));

}

else {

printf("ERROR: unknown method");

}

}

}

ifs.close();

}

void System::Function() {

int MM = (layers\_x + 2)\*(layers\_y + 2);

for (int t = 0; t < mol.size(); ++t) {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j) {

mol[t].u[i\*(layers\_y + 2) + j] = u[i\*(layers\_y + 2) + j + (t + 1) \* MM];

}

}

mol[t].FindG();

for (int i = mol[t].xmin; i < mol[t].xmax + 1; ++i) {

for (int k = mol[t].ymin; k < mol[t].ymax + 1; ++k) {

mol[t].Gforw[i][k][0] = mol[t].G[i][k];

}

}

mol[t].FindGforw(geo);

for (int i = 1; i < layers\_x + 1; ++i) {

for (int k = 1; k < layers\_y + 1; ++k) {

mol[t].Gback[i][k][mol[t].num\_atoms - 1] = mol[t].G[i][k];

}

}

mol[t].FindGback(geo);

mol[t].FindQ(geo);

mol[t].FindFiP();

mol[t].FindFiSide(geo);

}

FindFiSolv();

FindFiTotal();

FindFiSide();

for (int t = 0; t < mol.size(); ++t) {

FindLagrangeMultipliers(t);

}

FindLagrangeMultipliers();

FindGrad();

};

void System::FindFiSolv() {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j)

fi\_solv[i][j]= exp(-u[i\*(layers\_y + 2) + j]);

}

}

void System::FindFiTotal() {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j)

fi\_total[i][j] = mol[0].fi\_p[i][j] + mol[1].fi\_p[i][j] + fi\_solv[i][j];

}

}

void System::FindFiSide() {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k) {

fi\_side[l \* (layers\_y + 2) + k] = 0.0;

}

}

for (int i = -1; i <= 1; ++i) {

for (int j = -1; j <= 1; ++j) {

if (i == j == -1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_bb[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == -1 && j == 0) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_bn[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == -1 && j == 1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_bf[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == 0 && j == -1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_nb[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == 0 && j == 0) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_nn[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == 0 && j == 1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_nf[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == 1 && j == -1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_fb[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == 1 && j == 0) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_fn[l][k] \* fi\_solv[l + i][k + j];

}

}

else if (i == 1 && j == 1) {

for (int l = 1; l < layers\_x + 1; ++l) {

for (int k = 1; k < layers\_y + 1; ++k)

fi\_side[l \* (layers\_y + 2) + k] += geo.lambda\_ff[l][k] \* fi\_solv[l + i][k + j];

}

}

}

}

}

void System::FindLagrangeMultipliers(int t) {

int t2 = -1;

for (int t1 = 0; t1 < mol.size(); ++t1) {

if (t1 != t) {

t2 = t1;

}

}

if (t2 == -1) {

cout << "error" << endl;

return;

}

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j) {

mol[t].multipliers[i][j] = mol[t].u[i\*(layers\_y + 2) + j] - (mol[t2].fi\_side[i\*(layers\_y + 2) + j] \* chi\_seg + (fi\_side[i\*(layers\_y + 2) + j] - 1)\*mol[t].chi) / fi\_total[i][j];

}

}

};

void System::FindLagrangeMultipliers() {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j)

multipliers[i][j] = u[i\*(layers\_y + 2) + j] - (mol[0].fi\_side[i\*(layers\_y + 2) + j] \* mol[0].chi + mol[1].fi\_side[i\*(layers\_y + 2) + j] \* mol[1].chi) / fi\_total[i][j];

}

};

void System::FindGrad() {

int MM = (layers\_x + 2)\*(layers\_y + 2);

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j)

middle\_multipliers[i][j] = (mol[0].multipliers[i][j] + mol[1].multipliers[i][j] + multipliers[i][j])/3.0;

}

for (int t = 0; t < mol.size(); ++t) {

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j)

grad[i\*(layers\_y + 2) + j + (t + 1)\*MM] = 1 - 1.0 / fi\_total[i][j] + middle\_multipliers[i][j] - mol[t].multipliers[i][j];

}

}

for (int i = 1; i < layers\_x + 1; ++i) {

for (int j = 1; j < layers\_y + 1; ++j)

grad[i\*(layers\_y + 2) + j] = 1 - 1.0 / fi\_total[i][j] + middle\_multipliers[i][j] - multipliers[i][j];

}

for (int i = 1; i < M; ++i) {

grad[i] = -grad[i];

}

};

void System::AllocateMemory() {

u.resize(M + 2);

grad.resize(M + 2);

fi\_side.resize(M + 2);

fi\_solv.assign(layers\_x + 2, vector<double>(layers\_y + 2, 0));

fi\_total.assign(layers\_x + 2, vector<double>(layers\_y + 2, 0));

multipliers.assign(layers\_x + 2, vector<double>(layers\_y + 2, 0));

middle\_multipliers.assign(layers\_x + 2, vector<double>(layers\_y + 2, 0));

};

void System::Cycling() {

M = 3 \* (layers\_x + 2)\*(layers\_y + 2);

for (int i = 0; i < mol.size(); ++i) {

mol[i].AllocateMemory(layers\_x, layers\_y, M);

}

AllocateMemory();

for (BaseOptimTools &scheme : methods) {

cout << scheme.name << endl;

scheme.SetParameters(layers\_x, layers\_y, grad, u);

double length\_of\_grad = 0.0;

int step = 0.0;

Function();

length\_of\_grad = scheme.SetGradFirst(grad);

do {

scheme.UpdateX(u, grad);

Function();

length\_of\_grad = scheme.SetGradRegular(grad);

step++;

cout << step << " " << length\_of\_grad << endl;

} while ((length\_of\_grad > scheme.tolerance) && (step < scheme.num\_iter));

}

};

void System::Output() {

FILE \* txt1 = fopen("fi\_p\_1.txt", "w");

fprintf(txt1, "%5d ", (int)layers\_y);

for (int j = 1; j < layers\_y + 1; j++) {

fprintf(txt1, " %11d", j);

}

fprintf(txt1, "\n");

for (int i = 1; i < layers\_x + 1; i++) {

fprintf(txt1, "%5d ", i);

for (int j = 1; j < layers\_y + 1; j++) {

fprintf(txt1, " %8.5e", mol[0].fi\_p[i][j]);

}

fprintf(txt1, "\n");

}

fclose(txt1);

FILE \* txt2 = fopen("fi\_p\_2.txt", "w");

fprintf(txt2, "%5d ", (int)layers\_y);

for (int j = 1; j < layers\_y + 1; j++) {

fprintf(txt2, " %11d", j);

}

fprintf(txt2, "\n");

for (int i = 1; i < layers\_x + 1; i++) {

fprintf(txt2, "%5d ", i);

for (int j = 1; j < layers\_y + 1; j++) {

fprintf(txt2, " %8.5e", mol[1].fi\_p[i][j]);

}

fprintf(txt2, "\n");

}

fclose(txt2);

FILE \* txt3 = fopen("fi\_p\_all.txt", "w");

fprintf(txt3, "%5d ", (int)layers\_y);

for (int j = 1; j < layers\_y + 1; j++) {

fprintf(txt3, " %11d", j);

}

fprintf(txt3, "\n");

for (int i = 1; i < layers\_x + 1; i++) {

fprintf(txt3, "%5d ", i);

for (int j = 1; j < layers\_y + 1; j++) {

fprintf(txt3, " %8.5e", mol[1].fi\_p[i][j]+ mol[0].fi\_p[i][j] );

}

fprintf(txt3, "\n");

}

fclose(txt3);

};

main.cpp

#include "molecule.h"

#include "method.h"

#include "geometry.h"

#include "system.h"

int main() {

string my\_source = "input.txt";

System system(my\_source);

system.ReadParameters();

system.SetGeometry();

system.ReadMolecules();

system.ReadMethods();

system.Cycling();

system.Output();

return 0;

}

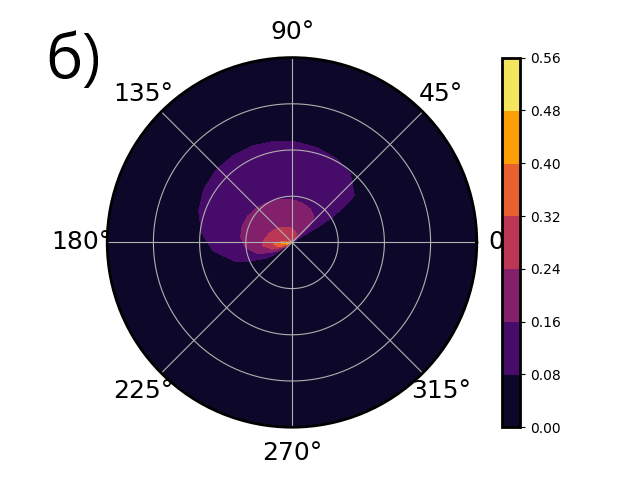
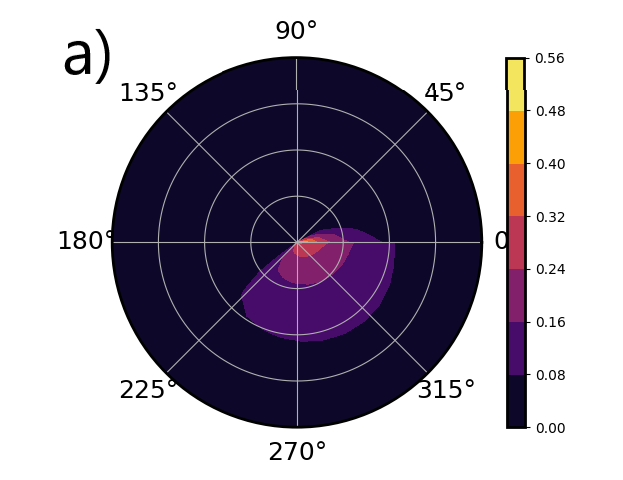
1. Результаты

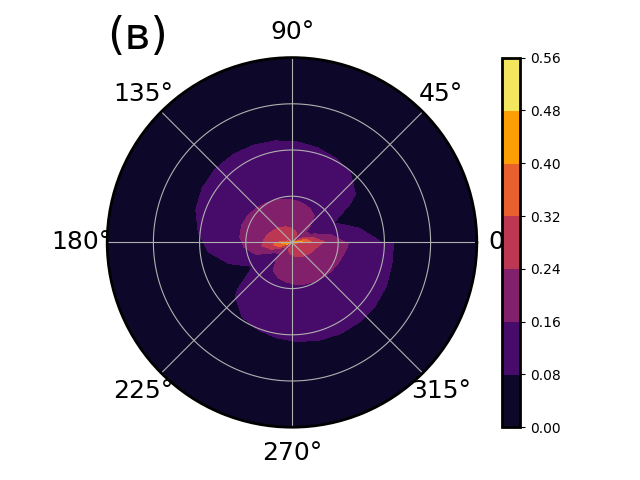
При входных данных:

input.txt

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| [ common ]  geometry polar  curvature 100  layers\_x 40  layers\_y 30  chi\_seg 2.0 | [ molecule1]  Ns 100  gen 0  sigma 15  chi 0.5  xmin 1  xmax 1  ymin 1  ymax 15 | [ molecule2]  Ns 100  gen 0  sigma 15  chi 0.5  xmin 1  xmax 1  ymin 16  ymax 30 | [ method ]  type gradient  num\_iter 200  tolerance 1e-7  step 0.2 | [ method ]  type gradient  num\_iter 5000  tolerance 1e-7  step 0.1 |

распределение молекул в полимерной щетке выглядит следующим образом:





*Рис.5 – а) профиль плотности первой молекулы, б) профиль плотности второй молекулы, в) профиль плотности двух молекул*

# Список литературы

1. Павловская Т. А., Щупак Ю. А. C++. Объектно-ориентированное программирование: Практикум. — СПб.: Питер, 2006. — 265 с: ил.
2. Гладких Б. А. Методы оптимизации и исследование операций для бакалавров информатики. Ч. II. Нелинейное и динамическое программирование: учебное пособие. — Томск: Изд-во НТЛ, 2011. — 264 с.
3. Михайлов И. В. Теория изгибной жесткости полимерных щеток, образованных привитыми дендронами: дис. канд. физ-мат. наук. ИВС РАН, СПб, 2018.
4. Иванов И.В. Амфифильные полиимидные щетки с гомо- и сополимерными полиметакрилатными боковыми цепями: дис. канд. хим. наук. ИВС РАН, СПб, 2017.
5. Шавыкин О. В. Методы оптимизации. Численный метод самосогласованного поля Схойтенса-Флира [Электронный ресурс] : презентация – Тверь, 2019. – 45 слайдов