Московский Государственный Университет

им. М. В. Ломоносова

Механико-математический факультет

Кафедра вычислительной математики

Шестов Алексей Михайлович

Группа 532

Дипломная работа

Построение и анализ дескрипторов молекулярной поверхности.

Научный руководитель

д. ф.-м. н., профессор

Кумсков М. И.

Москва, 2011

В работе описан новый подход к построению 3d дескрипторов молекулярных графов, учитывающих геометрию молекулярной поверхности и ее локальные физико-химические свойства (эти дескрипторы описаны в [45]) для решения задач «структура-свойство» и «структура-активность. Разработан алгоритм построения, оценена асимптотическая сложность и необходимая память алгоритма, рассмотрены дальнейшие возможные улучшения. Проект реализован в системе MatlabR2008a и на C++. Проведены вычислительные эксперименты на выборках молекул гликозидов, пиримидинов и стекла.

Оглавление.

0. Введение.

0.1. Задача «структура-свойство»

Задача «структура-свойство» («структура-активность») или QSPR(QSAR) , что является сокращением от английского Quantitative Structure Activity(Property) Relationships, - это задача распознавания образов [44]. Задача «Структура-свойство» заключается в поиске взаимозависимостей между структурами химических соединений и их свойствами с помощью построения математических моделей. Структура соединений описана с помощью молекулярных графов, потом с помощью них вычисляются дескрипторы (численно выраженные свойства химической структуры). Далее на основе этих дескрипторов с использованием различных алгоритмов строятся модели, которые впоследствии используются для предсказания свойств или активности новых молекул.

Решение этой задачи позволяет существенно уменьшить затраты на синтез новых соединений, т. е. создаются только те молекулы, у которых согласно прогнозу имеется нужное свойство. Основателем данного направления во многих литературных источниках считается американский ученый Корвин Ганч, хотя в ее создание существенный вклад внесли и такие ученые, как Ч.Овертон, Г.Мейер, Г.Фюнер, С.Фри, Дж.Вильсон, Т.Фуджита и др.

В методологии QSAR выделяют две подзадачи: прямую и обратную. Прямая заключается в определении по структуре свойств молекулы. Обратной QSAR задачей является создание структур химических соединений, которые будут обладать заданными свойствами.

В данной работе рассмотрено решение части прямой задачи «Структура-свойство» - построение дескрипторов.

0.2. Цель работы и результаты.

Цель работы – разработка, описание, реализация, тестирование и анализ алгоритма построения новых 3d дескрипторов молекулярных графов, учитывающих геометрию молекулярной поверхности и ее локальные физико-химические свойства. В дальнейшем будем называть построенные дескрипторы «дескрипторами молекулярной поверхности», или MS-дескрипторами (MS = Molecular Surface).

Анализ заключается в исследовании влияния изменения параметров алгоритма на качество прогноза. Для выборки молекул вычисляются дескрипторы с разными

параметрами. На основе вычисленных дескрипторов для молекулы формируется строка свойств. В фиксированном классе функций (в данной работе – среди *kNN* с кластеризацией [46]) строится наилучшая по какому-то критерию (в данной работе – минимизирующая q^2) функциональная зависимость между свойствами молекул выборки и их строками свойств. Вычисляется функция качества прогноза.

Так как взаимодействия в биологических реакциях происходят на уровне молекулярных поверхностей [10], то ожидается, что модель, построенная на дескрипторах молекулярной поверхности, покажет лучшие результаты, чем модель с использованием топологических 2d дескрипторов [47] и «сеточных» 3d дескрипторов (CoMFA [48], CoMSIA [49]). Кроме того, в отличие от упомянутых дескрипторов, дескрипторы молекулярной поверхности легко интерпретируемы с химической точки зрения.

В результате построены 3 алфавита дескрипторов

1. A - алфавит особых точек.
2. A^2 - алфавит пар особых точек, с 1 изменяемым параметром.
3. A^3- алфавит троек особых точек, с 3 изменяемыми параметрами.

Полученный прогноз на их основе имеет приблизительно такое же качество, как и на основе топологических дескрипторов [47]. Для прогноза использовался метод [46].

Научная новизна работы состоит в следующем:

1. ***Предложен и реализован новый метод построения содержательных с химической точки зрения дескрипторов в задаче «структура-свойство», использующий информацию о трехмерном строении молекулы и ее локальных физико-химических свойствах.***
2. ***Предложен и реализован новый метод сегментации трехмерной триангулированной поверхности.***
3. ***Проведена оценка вычислительной сложности и необходимой памяти алгоритма и проверенна его эффективность.***
4. ***Подтверждена практическая значимость подхода в серии вычислительных экспериментов по прогнозированию биологической активности органических соединений.***

Практическая значимость работы состоит в том, что разработанный алгоритм решения QSAR-задачи могут быть использован для решения прикладных задач предсказания физико-химической или биологической активности веществ по их структуре. Это может позволить отказаться от дорогостоящих и длительных исследований свойств большого числа молекул. Анализировать можно только те соединения, которые разработанная модель считает активными.

Материалы диплома докладывались и обсуждались на конференции PRIA 2010. По материалам диплома опубликованы 2 научные работы ([46], [50]).

Работа поддержана грантом РФФИ № 10–07–00694.

В первой главе работы рассмотрим общую постановку задачи «Структура-свойство» для молекулярных графов и дадим основные определения и описания молекулярных графов. *Во второй главе* дано общее описание типа дескрипторов, к которым относятся MS-дескрипторы (структурных дескрипторов и 3D дескрипторов). *В третьей главе* описан алгоритм построения дескрипторов и описаны дальнейшие направления развития. *В четвертой главе* проведен анализ алгоритма. *В пятой главе* приведем полученные результаты, их сравнения. *В шестой главе*, заключении, подведен итог работы. В приложении приведен текст программы.

Общая постановка задачи

«структура-свойство» для молекулярных графов.

1.1. Постановка задачи распознавания образов

Прежде чем рассматривать саму задачу «структура-свойство», которую предстоит решать, коснемся совокупности математических методов под названием ***распознавание образов.*** Задача ***Quantitative Structure Activity Relationships*** является всего лишь одной из частных проблем данной области.

Исходной информацией являются описания объектов, ситуаций, предметов, явлений или процессов S в виде векторов значений признаков $S = (x_1(S), x_2(S), ..., x_n(S))$, где признаки x_i, $i = 1, ..., n$, характеризуют различные стороны-свойства S. У объектов S существует "основное свойство" $y(S)$, которое для части объектов $S_1, S_2, ..., S_m$ предполагается известным, а для части объектов нет. Задача распознавания (прогноза, идентификации, "классификации с учителем") состоит в определении значения свойства $y(S)$ по информации $S_1, S_2, ..., S_m, y(S_1), y(S_2), ..., y(S_m)$ (*обучающей или эталонной выборке*).

Признаки могут быть числовыми (задающими степень выраженности какого-либо свойства), бинарными ("есть" или "нет" свойство), номинальными (обозначающими наличие различных свойств без числовой оценки - пол, цвет, и т.д.).

1.2. Основные понятия и определения задачи «структура-свойство», постановка задачи.

Меченый молекулярный граф $G = \{E, V\}$ – помеченный граф, вершины которого интерпретируются как атомы молекулы, а ребра – как валентные связи между парами атомов. Метки вершин и ребер (числа или символы) кодируют атомы и связи различной химической природы. В качестве меток вершин могут быть использованы любые характеристики соответствующих атомов (например, трехмерные координаты, символ химического элемента, заряд ядра, липофильность, атомный вес, атомный радиус и др.), а в качестве меток ребер – любые характеристики соответствующих связей (кратность, длины, порядки связей, полученные из квантово-химических расчетов, и т.д.).

Задача «структура-свойство»(QSAR) Пусть задана *обучающая (или эталонная) выборка* $L = \{G_i\}$, $i = 1 \ldots m$ - база данных из m химических соединений, которые обладают следующими свойствами:

1. *i*-ое соединение представлено меченым молекулярным графом G_i, имеющим укладку в трехмерном пространстве (т.е., для каждой вершины в качестве меток заданы ее трехмерные координаты);
2. Заданы значения свойства вещества $p(G_i)$, $p(G_i) = k_i$. Либо $k_i \in K$, K – множество классов активности (например, «активные», «слабоактивные», «неактивные» вещества), либо $k_i \in \boldsymbol{R}$ (для G_i задано численное значение исследуемого свойства).

Пусть $\mathcal{G}$ – множество всех меченых молекулярных графов с метками того же типа, что и в обучающей выборке *L*.

Необходимо построить классифицирующую функцию $F(G)$, $G \in \mathcal{G}$, где F должна наилучшим по определенному критерию образом относить это соединение к одному из классов активности ($F: \mathcal{G} \to K$) (предсказывать численное значение исследуемого свойства($F: \mathcal{G} \to \boldsymbol{R}$)), либо отказываться от прогноза.

В качестве критерия нахождения классифицирующей функции обычно используется максимизация функционала качества.

Функционал качества $\varphi(F)$ позволяет определить какая из классифицирующих функций наилучшая, т.е. . $F = \arg(\max_{f:\mathcal{G}\to K} \varphi(f))$ ($F = \arg(\max_{f:\mathcal{G}\to \boldsymbol{R}} \varphi(f))$). Например, в качестве функционала качества можно использовать процент верно классифицированных функцией F молекул из обучающей выборки:

$$\varphi(F) = 1 - \frac{\sum_{i=1}^{N} \varepsilon_i}{N}, \text{ где } \varepsilon_i = \begin{cases} 0, \textit{ если } F(G_i) = C_i \\ 1, \textit{ в противном случае} \end{cases},$$

или, в случае, когда функция должна предсказывать численное значение свойства,

$$\varphi(F) = 1 - \frac{\sum_{i=1}^{N} (F(G_i) - A_i)^2}{\sum_{i=1}^{N} A_i^2}$$

Дескриптором d будем называть какое-либо свойство, численное значение которого может быть вычислено для произвольного молекулярного графа G.

Алфавитом дескрипторов $A=\{d_i\}$, $i=1...n$, будем называть множество всех дескрипторов, используемых для анализа обучающей выборки, обозначенных различными символьными метками.

Дескрипторы из используемого алфавита должны обладать следующими свойствами:

1. Инвариантность относительно перенумерования вершин молекулярного графа.
2. Инвариантность относительно абсолютных координат вершин молекулярного графа.
3. Приемлемая сложность алгоритма вычисления дескриптора.
4. Для любой молекулы численное значение дескриптора находится в приемлемом числовом интервале, не достигая слишком маленьких (10^{-8}) или слишком больших значений (10^{12})(например, если в качестве дескриптора взять произведение каких-либо атомных свойств молекулы, то значение дескриптора будет быстро возрастать с ростом размера молекулы).

MS-дескрипторы удовлетворяют всем этим свойствам.

Также желательно, чтобы дескриптор обладал следующими свойствами:

1. Интерпретация с точки зрения структуры молекулы.
2. Высокая корреляция хотя бы с одним химически свойством.
3. Отсутствие тривиальной корреляции с другими дескрипторами.
4. Постепенное изменение значения при постепенном изменении молекулярной структуры.
5. В определении дескриптора не используются экспериментально полученные свойства.
6. Нет ограничения использования на слишком маленьком классе молекул.
7. Различение изомеров.
8. Дескриптор не определяется тривиальным образом через другие дескрипторы.
9. Обратная связь со структурой молекулы (т.е. по значению дескриптора можно восстановить структуру молекулы).

MS-дескрипторы удовлетворяют свойствам 1, 2, 3, 5, 6, 8.

Вектором признаков молекулярного графа *G* будем называть вектор $\bar{x}(G)=(x_1,..,x_n) \in R^n$,

где x_j - значение *j*-ого дескриптора, вычисленное для *G*, алфавит дескрипторов *A* состоит из *n* элементов.

Матрица «молекула-признак» (матрица признаков, MD-матрица) *MD* для рассматриваемой обучающей выборки $\{G_i\}$, $i = 1 \ldots m$, - матрица размера *m* x *n*, в *i*-ой строке которой стоит вектор признаков $\bar{x}_i = (x_{i1},..,x_{in})$ *i*-ого соединения.

1.3. Этапы решения задачи «структура-свойство»

Задача «структура-свойство» разбивается на два этапа: выбор описания молекулярных графов и построение функциональной зависимости.

На первом, исходя из формата молекулярных графов (типа меток вершин и ребер) обучающей выборки *L*, выбирается алфавит дескрипторов *A*. На основе этого алфавита строится отображение из множества молекулярных графов в признаковое пространство R^n, $\bar{x}: G \rightarrow R^n$, и формируется матрица «молекула-признак» для обучающей выборки.

На втором, в результате анализа матрицы «структура-свойство» на признаковом пространстве, строится модель функциональной зависимости - классифицирующая функция $F: \boldsymbol{R^m \rightarrow R}$, принадлежащая некоторому классу функций $\boldsymbol{\Phi}$ (например, линейные), с наилучшей прогностической способностью, т.е. $F = \arg(\max_{f \in \Phi} \varphi(f))$.

2. Обзор современных методов построения дескрипторов

2.1. Основные типы дескрипторов

Как правило, при решении задачи «структура-свойство» прогнозирующая функция *F* ищется одним из стандартных методов машинного обучения (линейные и нелинейные регрессии, нейронные сети и т. д.). Поэтому ключевой в задаче «структура-свойство» является задача выбора представления информации о структуре молекулы (т.е., набора признаков-дескрипторов).

Подходы к описанию молекул можно разделить на несколько больших групп:

1. Описание топологическими дескрипторами, основанными на свойствах молекулярного графа (теоретико-графовыми индексами);
2. Описание дескрипторами, соответствующими базовым геометрическим свойствам молекул (таким, как площадь поверхности, объем, диаметр и т.д.);
3. Описание структурными дескрипторами, характеризующими наличие, количество, и взаимное расположение в молекуле определенных структурных фрагментов;
4. Описание дескрипторами, характеризующими локальные физико-химические свойства молекулы в трехмерном пространстве, заполненном некоторой регулярной сеткой.

Полный обзор методов построения алфавитов дескрипторов можно найти в [55], а программу для их вычисления можно найти в [56]

2.2. Структурные дескрипторы.

Использование данного типа дескрипторов основано на выделении в молекулах структурных фрагментов. Структурные дескрипторы были впервые использованы в [5].

Структурным фрагментом молекулярного графа называется группа вершин с заданными условиями на их метки или метки их связей.

После выделения фрагментов каждому фрагменту сопоставляется структурный дескриптор, значение которого соответствует либо наличию или отсутствию данного фрагмента в молекулярном графе, либо количеству повторений фрагмента. В первом случае получаем дескриптор, принимающий логические значения, во втором - целые неотрицательные.[7,8]

Структурные фрагменты и соответствующие им дескрипторы подразделяют на два типа:

1. 2D-дескрипторы

В данном случае не учитываются значения валентных углов и евклидовых расстояний между атомами, не учитывается трехмерная структура фрагмента, важны только связи между атомами. Структурные 2D-фрагменты обычно имеют вид цепочек связанных атомов с определенными метками вершин и ребер, образующих данную цепочку.

2. 3D-дескрипторы

Дескрипторы этого типа учитывают трехмерную структуру фрагмента и обычно представляют собой множество вершин с заданными условиями на расстояния между ними и на их метки.

2.3. 3D QSAR

Построение зависимостей «структура-свойство» с использованием пространственного представления молекул обучающей выборки получило название 3D-QSAR. Основные этапы 3D-QSAR моделирования можно представить следующим образом:

1. Строятся пространственные представления молекул обучающей выборки, и проводится их «выравнивание» (alignment) согласно заданным правилам выбора ориентаций.
2. Для всех молекул вычисляется набор пространственно зависимых признаков.
3. Строится функция, выражающая зависимость вычисленных признаков и исследуемого значения химико-биологической активности.
4. Определяется устойчивость и предсказательную способность найденной функциональной зависимости;
5. При необходимости модель модифицируется, повторив пункты 1-4.

Наиболее распространенным методом 3D-QSAR является метод сравнительного анализа молекулярных полей CoMFA (Comparative Molecular Field Analysis), впервые предложенный Р. Крамером с соавторами в [**Error! Reference source not found.**]. При классическом CoMFA моделировании после выполнения процедуры пространственного выравнивания каждая молекула помещается в трехмерную регулярную решетку с заданным достаточно маленьким шагом. В узлах решетки вычисляются значения стерического (ван-дер-ваальсова) и электронного взаимодействия между данной молекулой и «пробным» атомом, имеющим свойства атома углерода в состоянии с зарядом «+1». Получаются две трехмерные матрицы, каждая из которых является дискретным представлением стерического и электронного поля исследуемой молекулы. Значения элементов этих матриц являются признаками молекул обучающего множества:

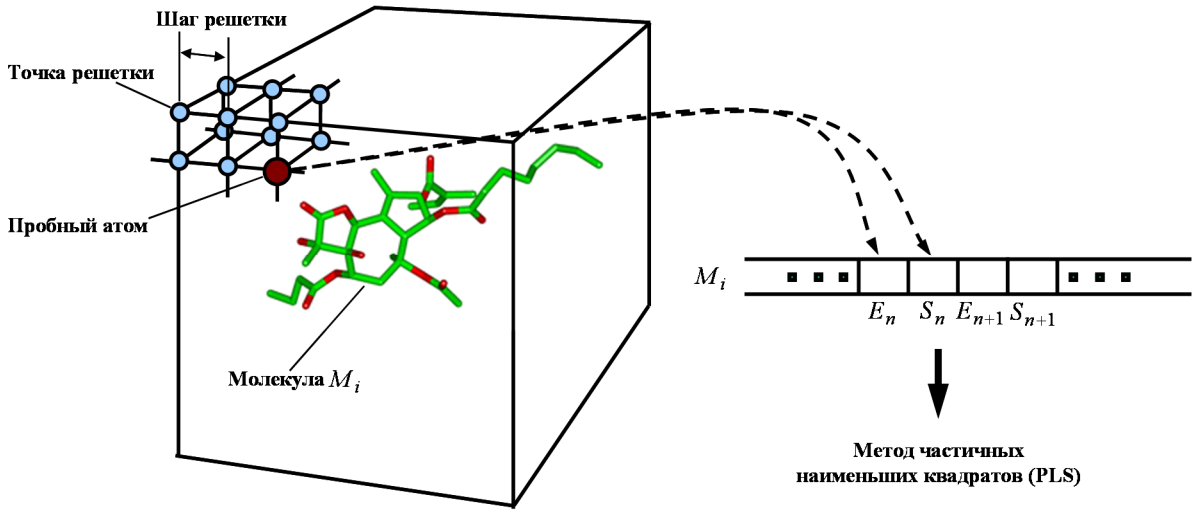


Рис. 1 Построение матрицы «молекула-признак» в алгоритме CoMFA.

В результате формируется очень широкая ($M >> N$) таблица «молекула-признак», которая затем анализируется одним из регрессионных методов.

Одним из важных достоинств метода является возможность (после возврата к исходным признакам) определения пространственных участков вокруг молекулы, изменения стерического и/или электронного поля в которых приводит к существенным изменениям в биологической активности, т.е. содержательной интерпретации полученной модели

Развитием CoMFA является метод сравнительного анализа индексов молекулярного сходства CoMSIA (Comparative Molecular Similarity Index Analysis) [49]. В этом методе в узлах решетки вычисляется широкий набор численных свойств (а именно, значения электростатического, стерического взаимодействия, липофильности, а также свойства донора-акцептора водородной связи). Этот набор свойств затем с помощью заданной функции специального вида трансформируется в единый индекс молекулярного сходства, который и помещается в матрицу «молекула-признак».

При правильном выборе метода выравнивания молекул в регулярной решетке методы CoMFA и CoMSIA могут показывать хорошие результаты классификации [**Error! Reference source not found.**], что сделало такие методы стандартом для прикладных исследований.

Наиболее сложным моментом в использовании CoMFA и подобных ему методов является необходимость «пространственной нормализации» молекул обучающей выборки, т.е. выбор их расположения (после взаимного пространственного выравнивания)

относительно системы координат регулярной решетки. Так, было обнаружено, что даже изменение положения системы координат (например, ее простой поворот) может привести к падению прогностической способности модели в два раза [**Error! Reference source not found.**].

Кроме этого, при увеличении сложности и гибкости молекул обучающего множества, методы 3D-QSAR, как правило, становятся неприменимы. При усложнении молекул резко возрастает число возможных способов их связывания с рецептором, что сильно затрудняет (а в ряде случаев делает невозможным) выбор однозначного выравнивания молекул в регулярной решетке.

3. Алгоритм построения дескрипторов

3.1. Постановка задачи вычисления MS-дескрипторов.

Дадим необходимы определения:

1. Ван-дер-ваальсовая поверхность

Пусть *G* – молекулярный граф, у которого для каждого атома (вершины) $V_i, i = 1,...,n$ заданы координаты в трехмерном пространстве и ван-дер-ваальсов радиус атома r_i – радиус сферической области, «занимаемой» атомом. Тогда **ван-дер-ваальсовой поверхностью** *V(G)* будем называть границу объединения открытых шаров ван-дер-ваальсова радиуса с центрами в соответствующих атомах, т.е. $V(G) = \partial \bigcup_{i=1}^{n} B_o(V_i, r_i)$ (рисунок 2).

2. Молекулярная поверхность

Молекулярной поверхностью $M_r(G)$ с пробным радиусом *r* [**Error! Reference source not found.**] для молекулярного графа *G* называют границу множества точек пространства, «доступных» для пробного шара радиуса *r* при прокатывании его по поверхности *W(G)*. Строго говоря,

$$M_r(G) = \partial\{x \in R^3 : \exists y \notin \bigcup_{i=1}^{n} B(P_i, r_i), \rho(x, y) \le r\}$$

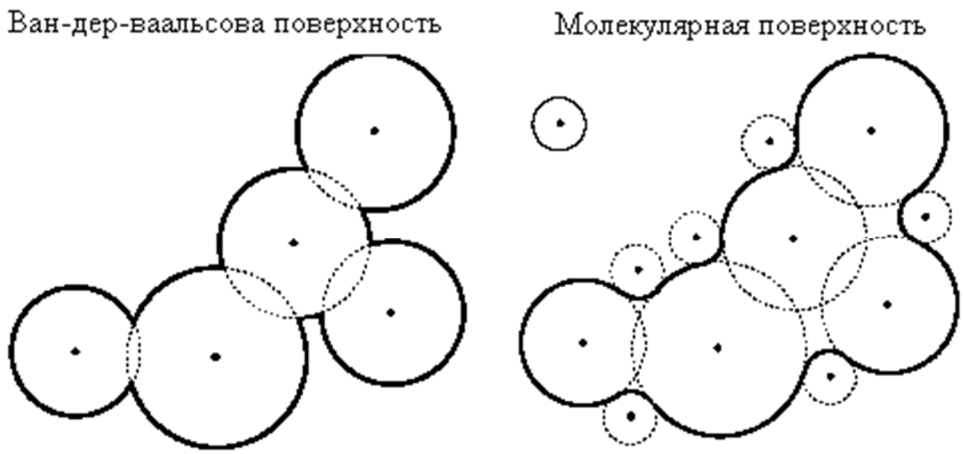


Рис. 2 Ван-дер-ваальсова и молекулярная поверхности.

3. Особые точки

Описанием молекулы особыми точками будем называть неупорядоченное множество $M_i = \{P_i^j\}_{j=1}^{n_i}$ особых точек, причем для каждой точки P_i^j заданы ее координаты в трехмерном пространстве $(x_i^j . y_i^j, z_i^j)$ и вектор свойств $(p_i^{j,1},...,p_i^{j,L}) \in R^L$.

Основные этапы задачи:

1. На молекулярной поверхности найти точки ξ_i, соответствующие ее геометрическим особенностям («впадины», «выпуклости» и седловые точки) (в дальнейшем будем называть их «особыми точками»).
2. В особых точках рассчитать локальные физико-химические свойства =( ) Это могут быть заряд, липофильность, гидрофобность, значение какого-либо молекулярного силового поля [52] и т.д.
3. На основе типа геометрической особенности и значении свойства классифицировать особые точки: $L(\xi_i)=l_i$.
4. На основе типов особых точек и расстояния между ними классифицировать пары (тройки) особых точек: $L(\xi_i, \xi_j)=l_{ij}$ ($L(\xi_i, \xi_j, \xi_k)=l_{ijk}$).

Численное значения *i*-ого дескриптора из алфавита дескрипторов для *j*-ой молекулы из выборки – это количество пар (троек) особых точек типа *i*, найденных в молекуле *j*.

Для решения этапов 2-4 использовались готовые методы, описанные в [45] (исследование этих этапов является дальнейшим направлением развития). Новизна MS-дескрипторов заключается в способе решения первого этапа задачи. В [45] уже был описан алгоритм выделения особых точек на молекулярной поверхности, но при его применении особых точек получалось слишком много и многие из них были близко расположены, что является нежелательным (см. рисунок). Поэтому придуман новый алгоритм.

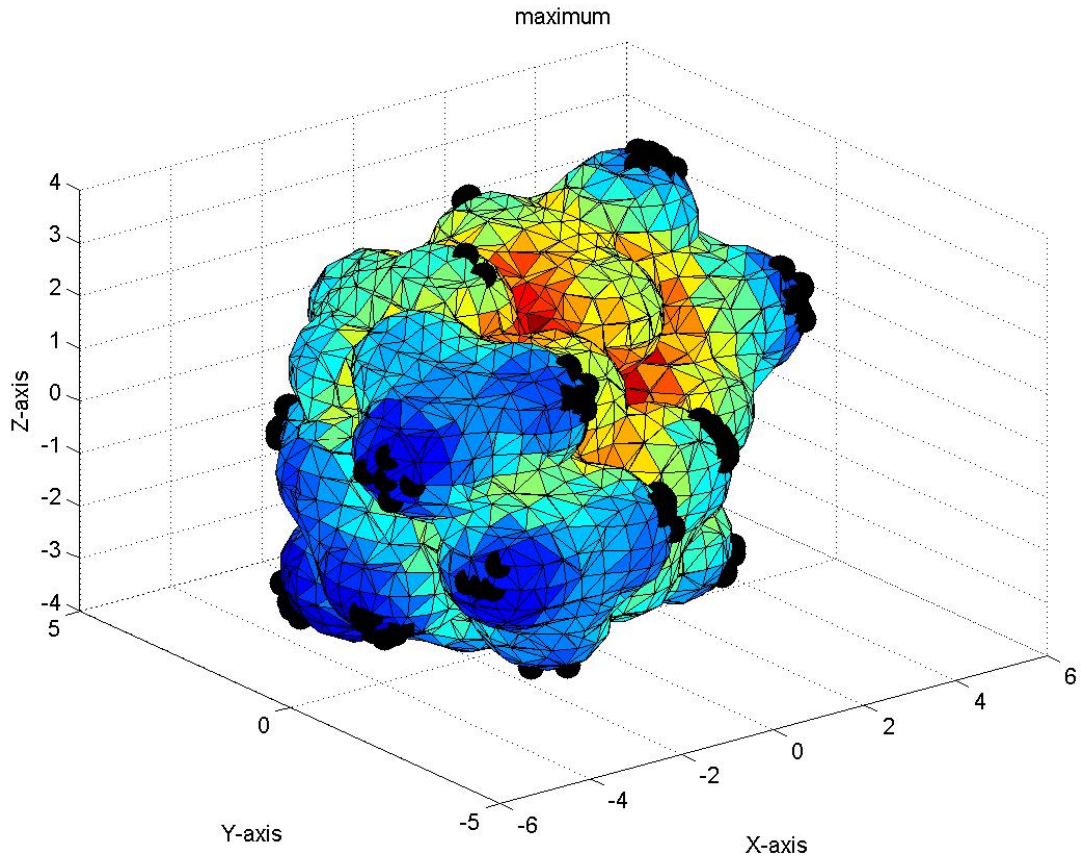


Рис. 3 Молекулярная поверхность с выделенными особыми точками в [45].

1.2. Актуальность подхода.

Известной моделью биологической активности является пространственный треугольник, вершины которого имеют заданные локальные физико-химические свойства. Взаимодействие лиганда и рецептора происходит согласно гипотезе «ключ-замок» между их молекулярными поверхностями, т.е. их молекулярные поверхности должны «подходить» друг к другу по форме, а также иметь необходимые физические и химические характеристики в точках соприкосновения [51]. Форма молекулярной поверхности играет ключевую роль при взаимодействии рецептора и лиганда [18].

В связи с этим было решено искать вершины треугольника активности именно на молекулярной поверхности. Таким образом, построенные дескрипторы отражают сразу 2 важных химических понятия – пространственный треугольник активности и молекулярную поверхность, и поэтому могут быть легко интерпретированы с химической точки зрения.

1.3. Идея алгоритма поиска особых точек.

Основной недостаток алгоритма, описанного в [45] – большое количество близко расположенных особых точек. На каждую «выпуклость» или «впадину» приходится несколько особых точек. Был придуман следующий способ избежать этого – разбить молекулярную поверхность на такие сегменты, что каждый представляет собой характерный участок поверхности – выпуклость (K (гауссова кривизна) >0, H (средняя кривизна) >0), впадину (K (гауссова кривизна) >0, H(средняя кривизна)<0) и седло (K (гауссова кривизна) <0), и в качестве особой точки взять геометрический центр каждого сегмента. Тогда особые точки будут обладать следующими свойствами:

1. Каждой особой точке будет соответствовать участок определенной формы (выпуклость, впадина или седло).
2. Каждому характерному участку поверхности будет соответствовать особая точка.
3. Особые точки не будут расположены слишком близко.

Таким образом, описание молекулярной поверхности будет полным, неизбыточным и содержательным.

Соответственно наиболее содержательная часть алгоритма – сегментация молекулярной поверхности.

Программы для вычисления молекулярной поверхности (MSMS [9], Discovery Studio Visualizer [10], Metamol [11]) представляют молекулярную поверхность в триангулированном виде. Соответственно, можно поставить задачу – сегментация триангулированной поверхности.

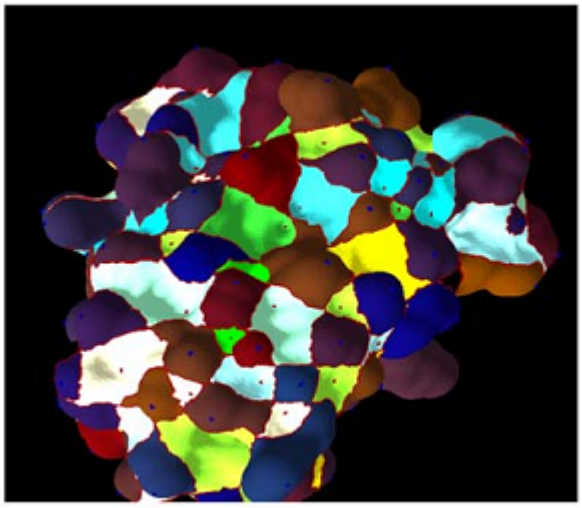


Рис. 4 Сегментированная молекулярная поверхность[25]

3.4. Обзор методов сегментации триангулированной поверхности

Рассмотрим общую задачу сегментации триангулированной повехности и дадим обзор основных методов ее решения.

Постановка задачи:

Дана кусочно-линейная двумерная поверхность G и гладкая двумерная поверхность M в $\boldsymbol{R}^3$. $G = \cup T_k, T_k = \{p_i,\ p_j,\ p_l\},\ p_i \in M\ \forall i$. Необходимо построить разбиение $\{G_i\}, \cup_i G_i = G, \cap_i G_i = \emptyset$.

Задача сегментации возникает во многих приложениях, связанных с трехмерной графикой - наложение текстуры [12], изменение формы [13], сравнение формы [14], инженерные приложения [15] и т.д. Общий обзор существующих методов можно найти в [16,17], а методов для сегментации именно молекулярной поверхности – в [18].

Триангулированную поверхность можно также рассматривать как граф.

Суть большинства методов заключается в следующем – выбирается функция $f: G \rightarrow \boldsymbol{R}$, поверхность разбивается на сегменты в соответствии со значениями этой функции.

Соответственно, решение задачи сегментации можно разбить на 2 шага – выбор и вычисление функции f и разбиение поверхности G. Рассмотрим, какие существуют варианты реализации каждого из этих двух шагов алгоритма.

3.4.1. Вычисление скалярной функции на поверхности

3.4.1.1.Кривизна

Как функции, характеризующие форму поверхности, могут быть выбраны

- гауссова кривизна, в дальнейшем будем обозначать как *K*. $K=k_1 \cdot k_2$
- средняя кривизна в дальнейшем будем обозначать как *H*. $H=(k_1+k_2)/2$
- их различные комбинации [20,21]

$$STI(x) = \begin{cases} \dfrac{\kappa_1 - \kappa_2}{\kappa_1} & \text{if } \kappa_1 > 0, \kappa_2 > 0; \\ & \text{or, if } \kappa_1 > 0, \kappa_2 \leq 0, \text{ and } |\kappa_1| > |\kappa_2| \\ \dfrac{\kappa_1 + 3\kappa_2}{\kappa_2} & \text{if } \kappa_1 > 0, \kappa_2 \leq 0, \text{ and } |\kappa_1| \leq |\kappa_2|; \\ & \text{or, if } \kappa_1 \leq 0 \text{ and } \kappa_2 < 0 \\ -1 & \text{if } \kappa_1 = 0 \text{ and } \kappa_2 = 0. \end{cases}$$

, где k_1 и k_2 – главные кривизны поверхности *M*[19]

Аналитическая форма поверхности *M*, вообще говоря, неизвестна, поэтому значение кривизны восстанавливается по поверхности *G*. Существуют различные методы восстановления значения кривизны, их обзор дан в [22,23].

Недостаток кривизны – локальный характер описания поверхности. В работе [32] описывается дескриптор формы, который рассчитывается с помощью одного из методов нахождения кривизны – приближения параболоидом, но для окрестности большего размера, чем при вычислении кривизны. Таким образом, он учитывает форму большей окрестности точки, чем кривизна.

Авторами работ были получены следующие результаты

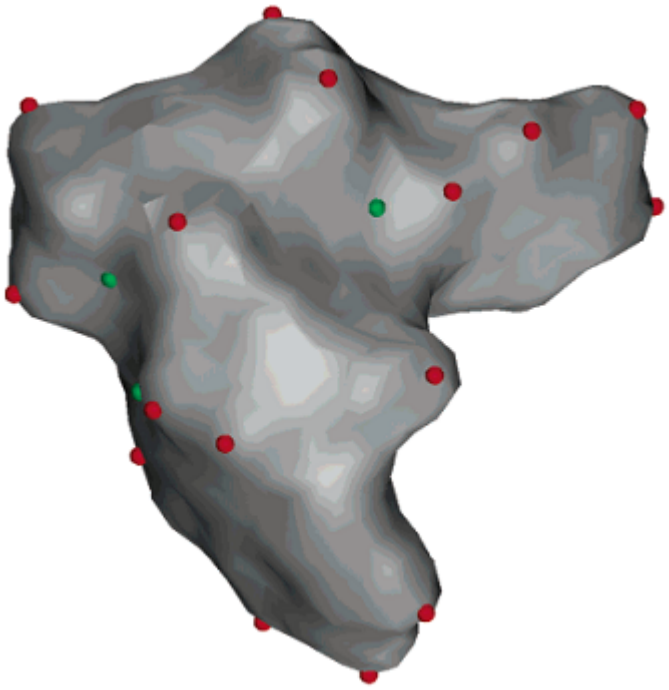


Рис. 5 Экстремумы кривизны на поверхности, кривизна вычисляется с помощью приближения поверхности параболоидами [24]

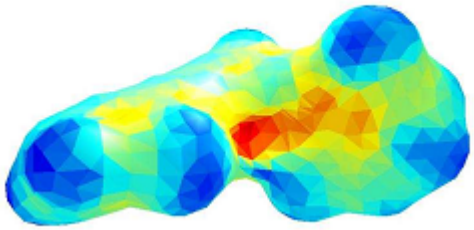


Рис. 6 Поверхность окрашена в соответствии с величиной средней кривизны [25]

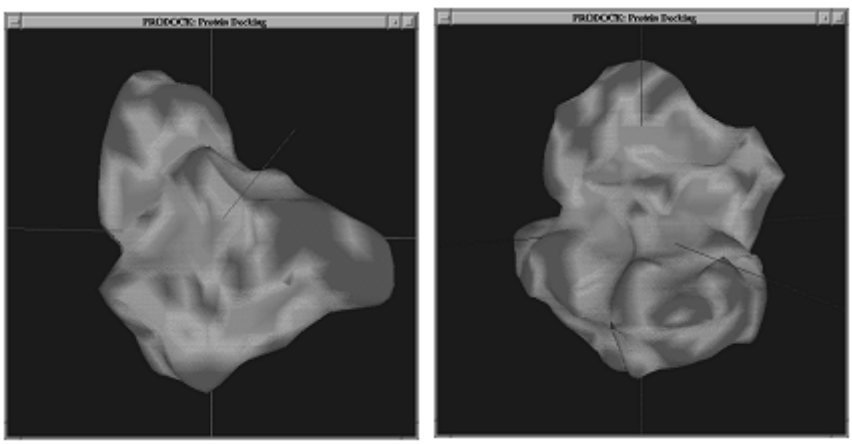
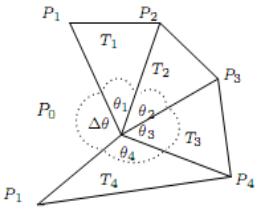


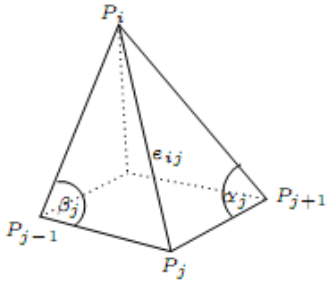
Рис. 7 Поверхность окрашена в соответствии со значениями главной и гауссовой кривизн[19]

В работе [19] главная и гауссова кривизна вычислены в каждой вершине P_i по следующим формулам

$K=3*(2pi-\sum \theta_j)/\sum A_j$, где A_j- площадь j-ого треугольник, содержащего P_i в качестве вершины



$$-H\vec{n} = \frac{1}{4A} \sum_{j \in N(i)} (\cot \alpha_j + \cot \beta_j)(P_j - P_i),$$



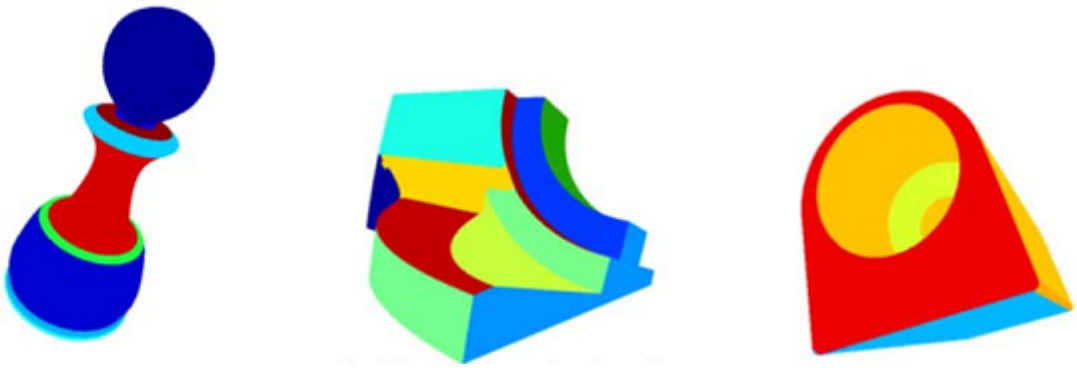


Рис. 8 [15] Сегментация поверхности, основанная на значении главных кривизн, вычисленных с помощью метода, описанного в[26]

Метод, описанный в работе [15]: для каждой вершины триангуляции вычисляются главные кривизны, потом все значения главных кривизн делятся на кластеры алгоритмом *k*-средних, каждой вершине присваивается номер кластера, поверхность сегментируется в зависимости от полученных номеров.

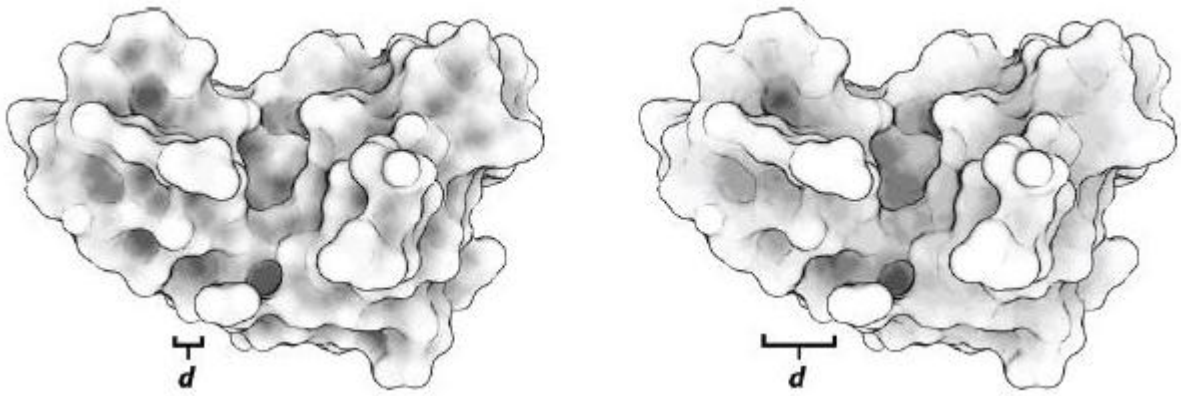


Рис. 9 Значение дескриптора формы[32] для маленькой(слева) и большой(справа) окрестности точки.

3.4.1.2.Функция Коннолли [27,28].

Функция Коннолли C определяется следующим образом – пусть поверхность G ограничивает область O в $\boldsymbol{R^3}$. Рассмотрим сферу S с центром в точке p $\in G$ радиуса r такого, что G разделяет сферу на 2 части, S_i и S_o, каждая из которых топологически эквивалентна диску. $S_i \in O$, $S_o \notin O$. Тогда C(r,p)=SurfaceArea(S_i)/r^2.

Для двумерного случая функция Коннолли – это угол, отсекаемый кривой на окружности[18].

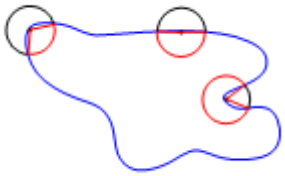


Рис. 10 Двумерный вариант функции Коннолли

Достоинства данной функции – в зависимости от радиуса сферы мы будем получать описания разной степени локальности. При r→0 экстремумы функции Коннолли стремятся к экстремумам средней кривизны[25].

Недостатки – при вычислении не учитываются детали внутри сферы, поэтому при использовании сфер определенного радиуса для случаев, показанных на рисунке, функция Коннолли будет принимать одинаковые значения[18]. Также к недостаткам можно отнести сложность вычисления функции.

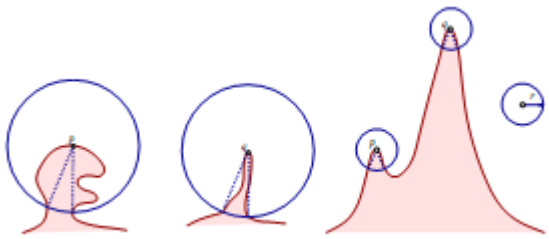


Рис. 11 Функция Коннолли не учитывает детали внутри сферы

В [25] были получены следующие результаты:

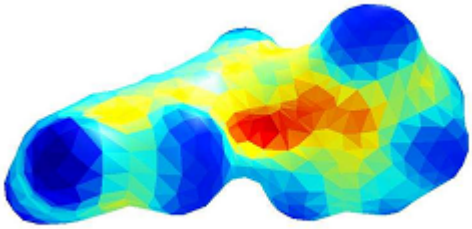
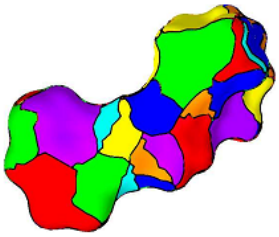
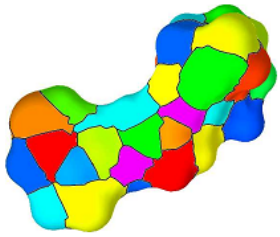
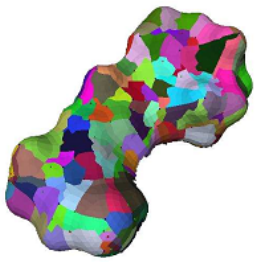


Рис. 12 Поверхность окрашена в соответствии со значениями функции Коннолли[25]







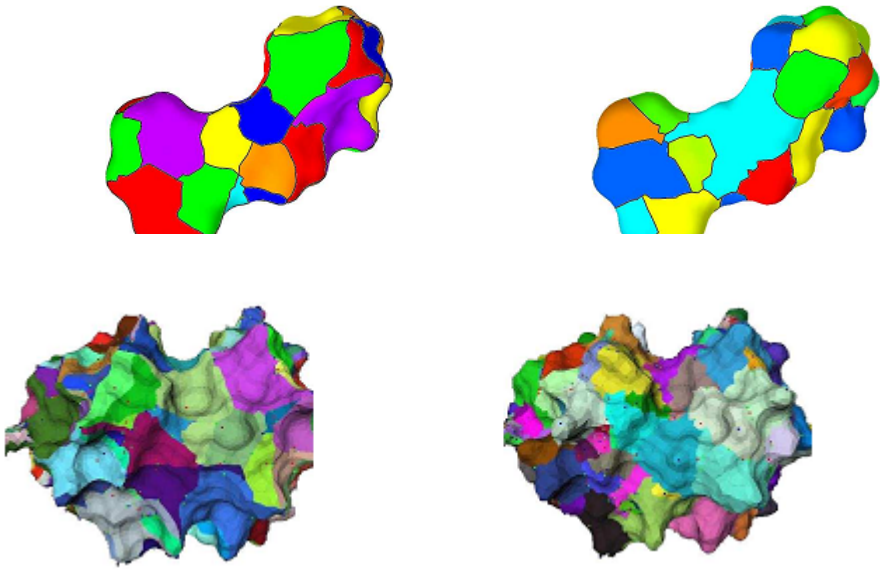


Рис. 13 Поверхности сегментированы в соответствии со значениями функции Коннолли[25]

3.4.1.3.Функция атомной плотности [29].

Это дальнейшее развитие идеи функции Коннолли. Пусть поверхность G ограничивает область O в $\boldsymbol{R^3}$. Рассмотрим набор шаров S_i с центрами в точке $p \in G$ радиусов r_i. $V(p, r_i)$-объем $O \cap S_i$. С помощью метода наименьших квадратов аппроксимируем множество точек $(r_i, V(p, r_i))$ функцией $y=\sigma x^3$. Полученный коэффициент σ – это и есть функция атомной плотности в точке p.

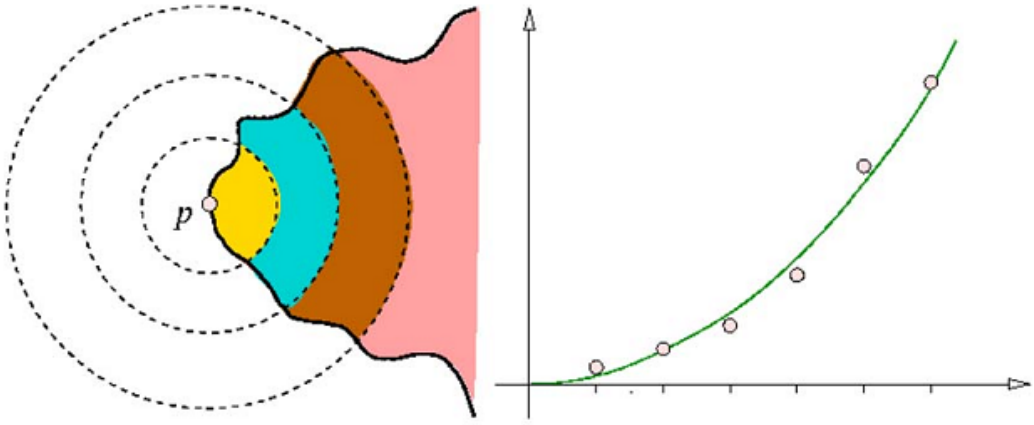


Рис. 14 Иллюстрация двумерного варианта функции атомной плотности[31].

Улучшение по сравнению с функцией Коннолли состоит в том, что учитываются больше особенностей поверхности благодаря нескольким сферам.

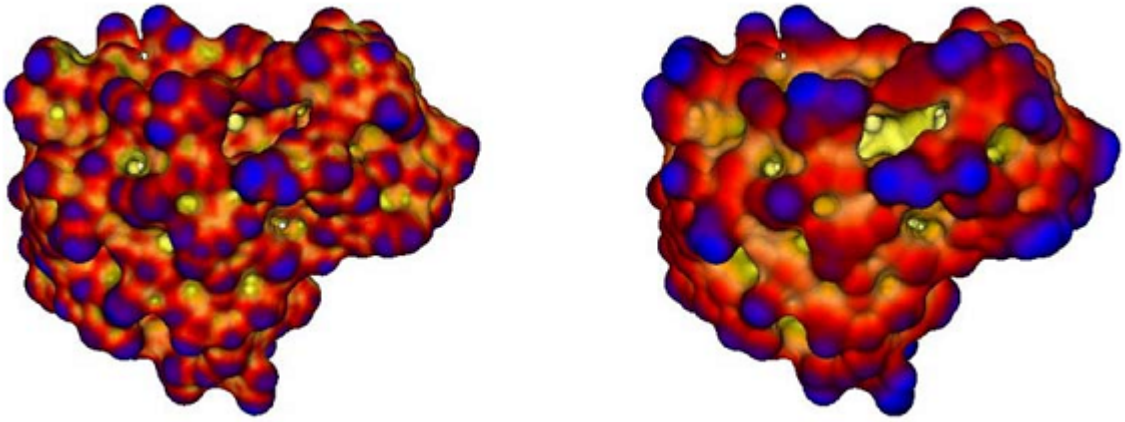
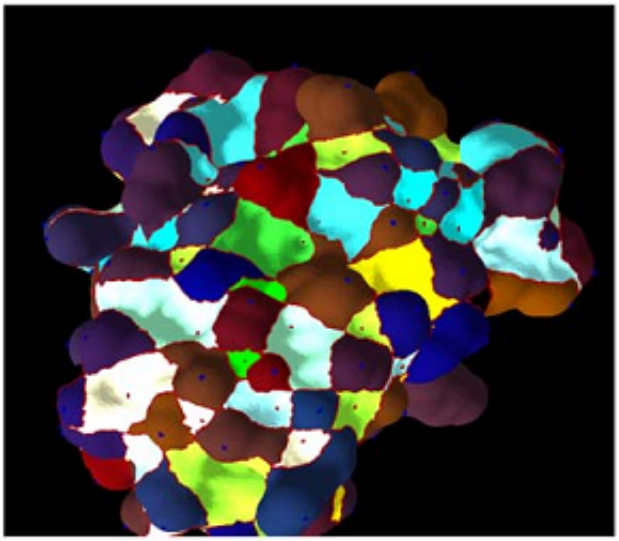


Рис. 15 Сравнение функции Коннолли (слева) с функцией атомной плотности (справа) [31]

Программа для вычисления функции атомной плотности доступна по адресу [30].

В работе [31] были получены следующие результаты (см. рисунок)



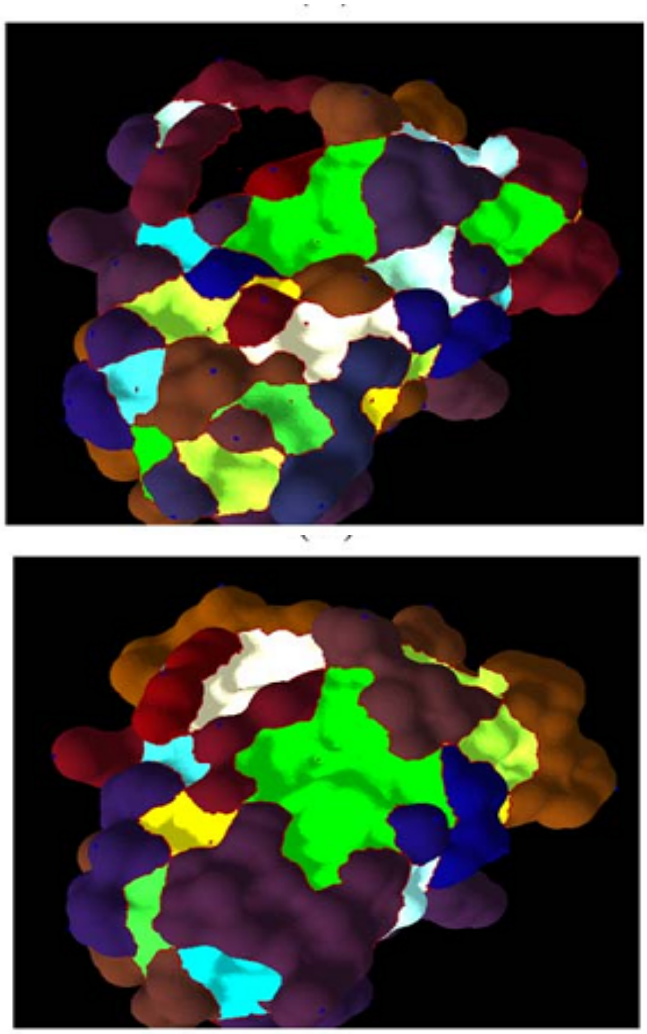


Рис. 16 Сегментация молекулярной поверхности разной степени подробности с использованием функции атомной плотности [31].

3.4.1.4. Использование выпуклой оболочки.

Также есть функции, исследование сегментации на основе которых, не было проведено, но возможно, их использование может дать хорошие результаты в каких-то случаях – расстояние от точки до выпуклой оболочки поверхности [33,18] и минимальное расстояние без пересечений до выпуклой оболочки поверхности [18]

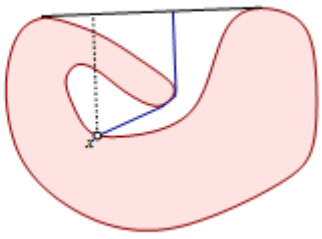


Рис. 17 Минимальное расстояние без пересечений до выпуклой оболочки поверхности[18]

3.4.2. Различные алгоритмы сегментации

Алгоритмы можно разделить на 2 группы – наращивание сегментов и выделение границ

3.4.3.1. Выделение сегментов

Обзор работ, в которых используется этот вариант сегментации дан в[8].

Приведем основные алгоритмы:

- *Region Growing Algorithm*

```
Initialize a priority queue Q of elements
Loop until all elements are clustered
    Choose a seed element and insert to Q
    Create a cluster C from seed
    Loop until Q is empty
        Get the next element s from Q
        If (s can be clustered into C)
            Cluster s into C
            Insert s neighbours to Q
Merge small clusters into neighbouring ones
```

- *Multiple Source Region Grow Algorithm*

```
Initialize a priority queue Q of pairs
Choose a set of seed elements {si }
Create a cluster Ci from each seed si
Insert the pairs < si , Ci> to Q
Loop until until Q is empty
        Get the next pair < sk , Ck> from Q
        If( (sk is not clustered already) and (sk can be clustered into Ck))
            Cluster sk into Ck
            For all un-clustered neighbours si of sk insert < si , Ck> to Q
Merge small clusters into neighbouring ones
```

- *Hierarchical Clustering Algorithm*

```
Initialize a priority queue Q of pairs
Insert all valid element pairs to Q
Loop until Q is empty
    Get the next pair (u, v) from Q
    If ((u, v) can be merged)
        Merge (u, v) into w
    Insert all valid pairs of w to Q
```

- *Iterative Clustering Algorithm*

Initialize k representatives of k clusters
Loop until representatives do not change
For each element s
Find the best representative i for s
Assign s to the ith cluster
For each cluster i
Compute a new representative

Результаты:

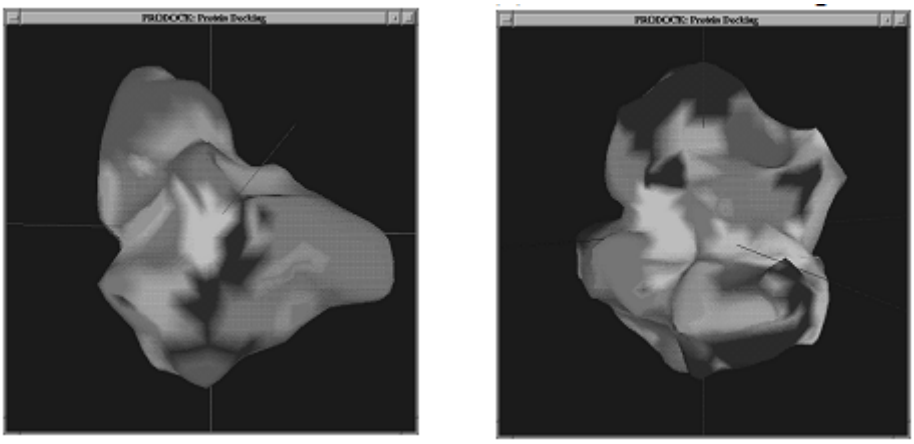


Рис. 18 Multiple Source Region Grow Algorithm [11]

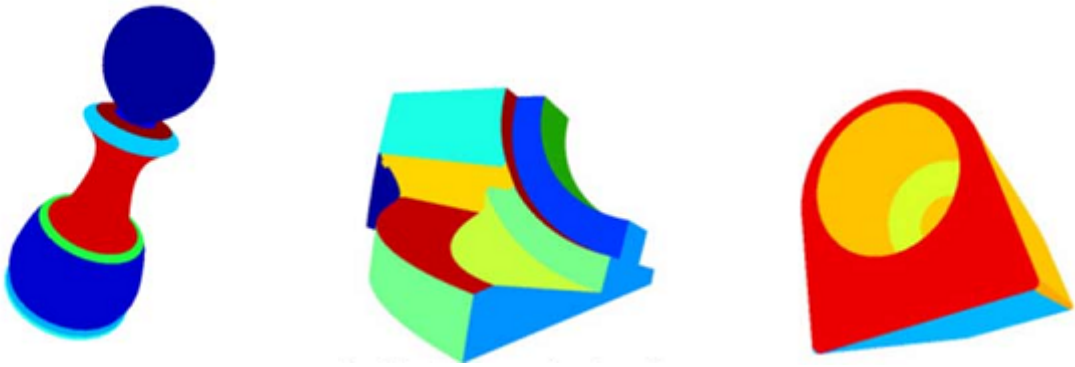


Рис. 19 Multiple Source Region Grow Algorithm [7]

3.4.2.1.Выделение границ.

Этот метод часто используется, когда поверхность представляет собой какой-то объект, и разделить его на значимые части (например, разделить фигуру человека на части тела). В

этом случае разделение производится по граням – границы проводятся по вершинам с высокой кривизной. Однако в случае молекулярных поверхностей такой метод не подходит, т.к. большая часть молекулярной поверхности является гладкой.

В работах [17,23] границы были выделены с помощью построения комплекса Морса-Смейла. В [17] была использована теория для кусочно-линейных поверхностей, введенная в [26]. В [23] используются результаты работ [27,28], расширяющих теорию для случая кусочно-линейных поверхностей. С помощью данного метода можно проводить сегментацию разной степени подробности.

Результаты:

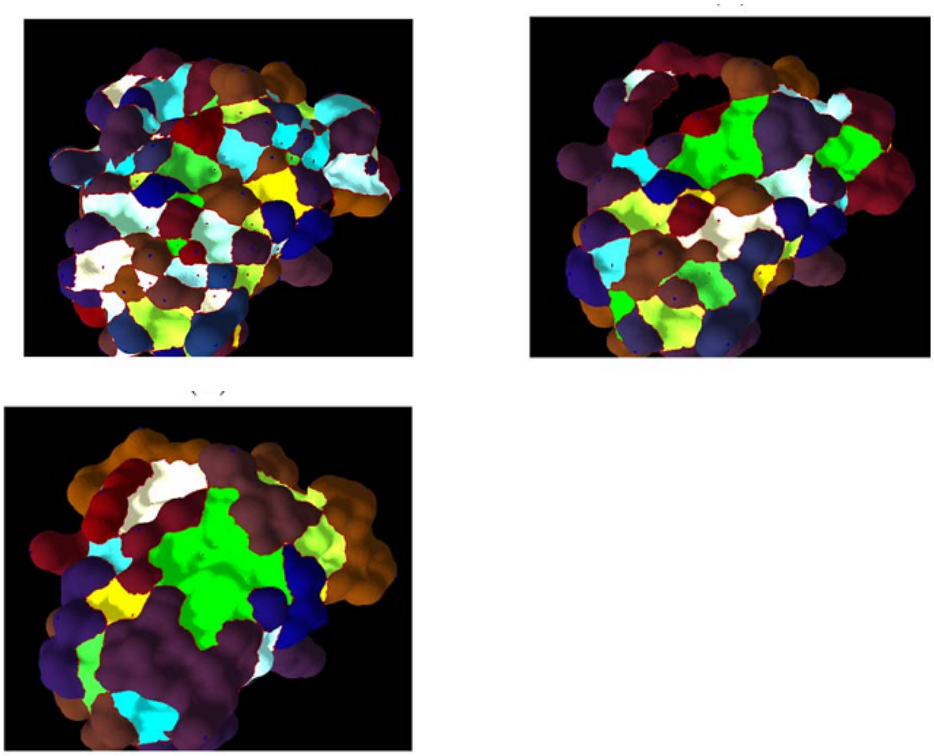
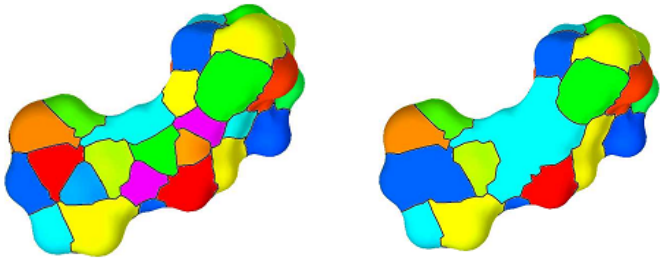


Рис. 20 Сегментация разной степени подробности в [23].



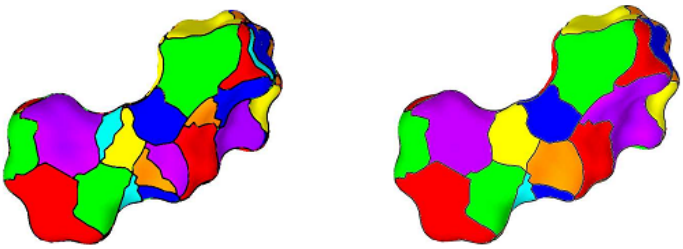


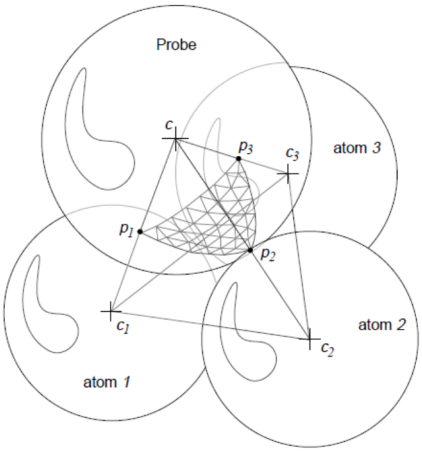
Рис. 21 Сегментация разной степени подробности в [17]

3.5. Общая идея алгоритма сегментации молекулярной поверхности.

Способ построения молекулярной поверхности подсказывает идею ее сегментации.

Опишем построение молекулярной поверхности.

Молекулярная поверхность – это пересечение сфер, «сглаженное» прокатыванием пробного шара определенного радиуса по ним [40]. «Сглаживаются» либо пересечения двух сфер, либо пересечения трех сфер. Фигура, получающаяся, когда пробный шар касается двух сфер – это часть тора. Фигура, получающаяся, когда пробный шар касается трех сфер – внутренняя сторона сферы. Получается, что молекулярная поверхность – это поверхность, «составленная» из фрагментов сфер и торов разных радиусов [9].



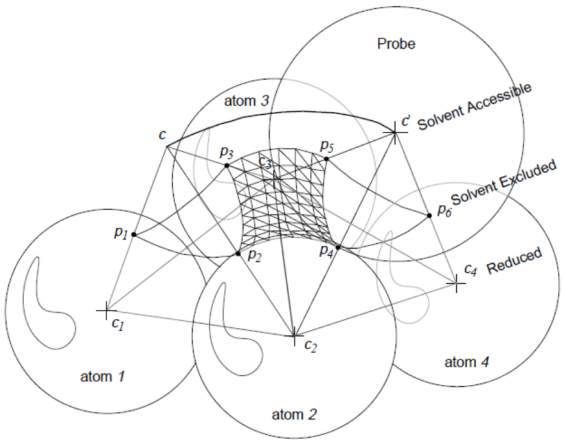


Рис. 22 Кусок тора, получающийся при прокатывании пробной сферы между 2х атомов

Рис. 23. Внутренняя часть сферы, получающаяся при касании пробной сферы 3х атомов

Наружные части разных сфер ($K>0$, $H>0$) разделены кусками внутренней части сферы ($K>0$, $H<0$), либо кусками тора ($K<0$). Значит, части молекулярной поверхности положительных средней и гауссовой кривизн, соответствующие разным сферам, попарно несвязанны, и каждую такую часть можно выделить Region Groing Algorithm (см. 3.4.2.1)

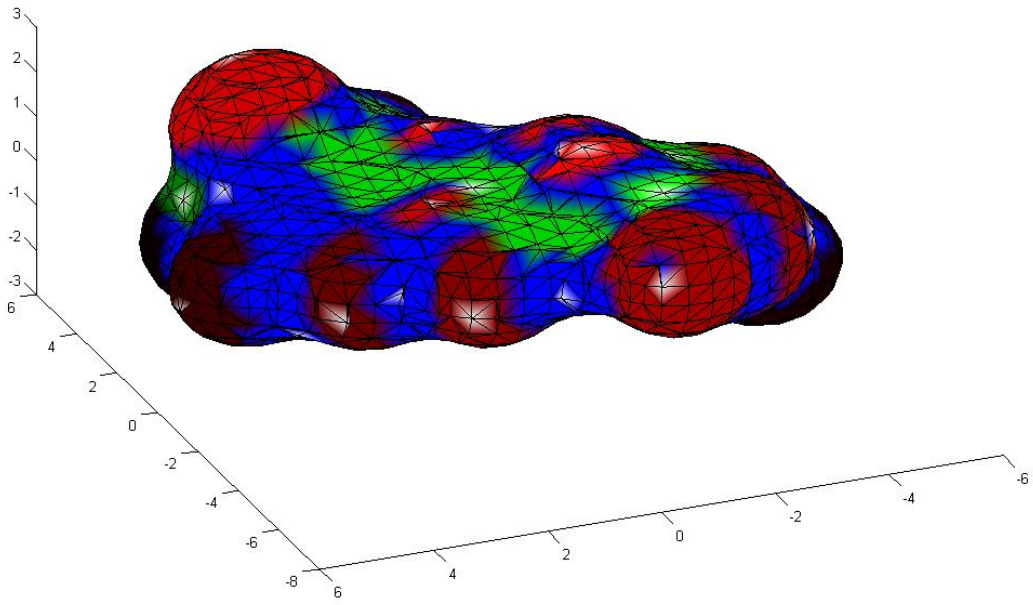


Рис. 24 Молекулярная поверхность с посчитанной кривизной

На рисунке красным цветом выделены участки $K>0$, $H>0$, зеленым – $K>0$, $H<0$, синим – $K<0$.

Идея алгоритма – сначала выделить участки отрицательной средней кривизны, потом положительной гауссовой, оставшийся участок сегментировать алгоритмом к-средних[37].

3.6. Реализация алгоритма.

3.6.1. Построение молекулярной поверхности.

Из всех возможных программ для построения молекулярной поверхности была выбрана Accelrys Discovery Studio[10], потому что остальные либо не сохраняли поверхность в нужном формате(Molekel[41], LSMS[42]), либо имеют слишком крупный размер треугольников при триангуляции(MSMS[9]). Также эта программа проводит трехмерную

оптимизацию геометрии молекулы, что является большим плюсом. Поверхность сохраняется в файл с расширением wrl (Plain Text VRML File), откуда извлекаем список координат вершин триангуляции, список треугольников (треугольник описывается номерами вершин, из которых он состоит) и список координат нормалей к каждой вершине (благодаря этому мы избавлены от одной из частых проблем, возникающих при сегментации – расчет приближенного значения нормали в каждой точке).

3.6.2. Входные данные.

Считаем, что входные данные алгоритма:

1. Множество вершин триангуляции $V=\{v_i\}$, $i=1...N_v$, заданных своими *3D* координатами (x_i, z_i, y_i).
2. Множество треугольников $T=\{t_i\}$, $i=1... N_t$, заданных своими вершинами (v_{i1}, v_{i2}, v_{i3}), $v_{ik} \in V$, $k=1...3$.
3. Множество векторов нормалей к исходной поверхности M $N=\{n_i\}$, $i=1... N_v$, заданных своими *3D* координатами (x_i, z_i, y_i).

Назовем *окрестностью вершины триангуляции* v_i $O(v_i)=\{v_j: \exists e \in E, e=(v_j, v_i)\}$, *окрестностью нормалей вершины триангуляции* v_i $O_n(v_i)=\{n_j: \exists e \in E, e=(v_j, v_i)\}$.
Для дальнейшей работы вычисляются

1. $O_{all}=\{O(v_i)\}$, $i=1... N_v$.
2. $O_{n\,all}=\{O_n(v_i)\}$, $i=1... N_v$.

В алгоритме выделяются 3 типа вершин – выпуклые ($H>0$, $K>0$), вогнутые ($H<0$, $K>0$) и седловые ($K<0$). Каждой вершине v_i присваивается метка L_i по ее типу, $L_i=\{1, 2, 3\}$.

3.6.3. Расчет кривизны в вершинах триангуляции

Для сегментации необходимо посчитать метку каждой вершины, а для этого нужно посчитать кривизну в каждой вершине триангуляции. Т.к. исходная поверхность M не известна, то приходится восстанавливать кривизну по поверхности G. Среди всех способов были выбраны 2 - приближением квадратичной поверхностью [23] и кубической [38] . Эти методы были выбраны в связи с хорошим качеством приближения [22] и простотой реализации. Средняя кривизна считается вторым способом, гауссова – первым.

Приведем описание алгоритма расчета кривизны.

Надо рассчитать кривизну в каждой вершине триангуляции v_i.

Рассмотрим триангулированную поверхность G как граф $G=\{V, E\}$

Шаги алгоритма приближения кривизны с помощью параболлоида:

1. Введем новую ортонормированную систему координат $\widetilde{OXYZ}$, т.ч $\tilde{O} = v_i$, $\widetilde{e_z} = \bar{n}_i$. $\widetilde{e_x} \in A \perp \widetilde{OZ}$, $\|\widetilde{e_x}\| = 1$, $\widetilde{e_y} = \frac{\widetilde{e_z} \times \widetilde{e_x}}{\|\widetilde{e_z} \times \widetilde{e_x}\|}$.
2. Расчитаем координаты $v_j \in O(v_i)$ в новой системе координат.
3. С помощью метода наименьших квадратов ищем поверхность U в системе координат $\widetilde{OXYZ}$ в форме $z=ax^2+bxy+cy^2$, аппроксимирующую множество вершин $O(v_i)$.
4. Для полученного параболоида $z=ax^2+bxy+cy^2$ искомая средняя кривизна $H=a+c$, гауссова $K=4ac-b^2$.
5. Повторяем этот процесс для каждой вершины.

Шаги алгоритма приближения кривизны с помощью кубической поверхности:

1. Введем новую ортонормированную систему координат $\widetilde{OXYZ}$, т.ч $\tilde{O} = v_i$, $\widetilde{e_z} = \bar{n}_i$. $\widetilde{e_x} \in A \perp \widetilde{OZ}$, $\|\widetilde{e_x}\| = 1$, $\widetilde{e_y} = \frac{\widetilde{e_z} \times \widetilde{e_x}}{\|\widetilde{e_z} \times \widetilde{e_x}\|}$.
2. Расчитаем координаты $v_j \in O(v_i)$ в новой системе координат.
3. Расчитаем координаты $n_j \in O_n(v_i)$ в новой системе координат.
4. С помощью метода наименьших квадратов ищем поверхность U в системе координат $\widetilde{OXYZ}$ в форме $z=ax^2+bxy+cy^2+dx^3+ex^2y+fxy^2+gy^3$, аппроксимирующую множество вершин $O(v_i)$ и множество нормалей $O_n(v_i)$. Т.е. нормали к U в вершинах $v_j \in O(v_i)$ аппроксимируют нормали $n_j \in O_n(v_i)$.
5. Для полученной поверхности $z=ax^2+bxy+cy^2+dx^3+ex^2y+fxy^2+gy^3$ искомая средняя кривизна $H=a+c$, гауссова $K=4ac-b^2$.
6. Повторяем этот процесс для каждой вершины.

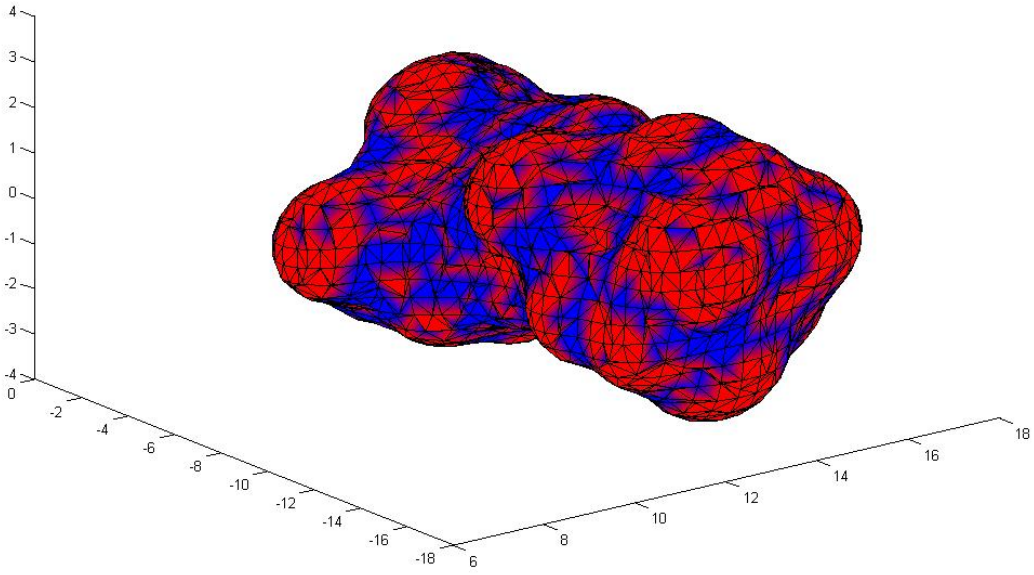


Рис. 25 Гауссова кривизна посчитана с помощью приближения кубической поверхностью

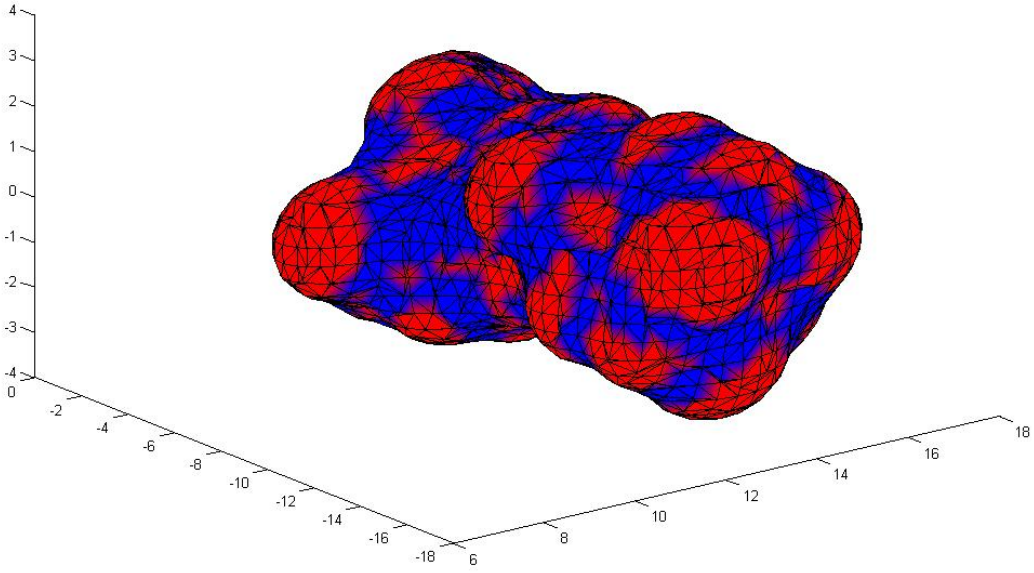


Рис. 26 Гауссова кривизна посчитана с помощью приближения квадратичной поверхностью

Красным обозначены вершины с $K>0$, синим – с $K<0$. На первом рисунке K посчитана с помощью приближения кубической поверхностью, на втором – квадратичной. Видно, что для выделения выпуклых сегментов предпочтительней считать кривизну через приближение квадратичной поверхностью, т.к. в этом случае сегменты, соответствующие разным атомам, оказываются несвязанны между собой. Для выделения вогнутых сегментов лучше подходит приближение кубической поверхностью, т.к. если считать через квадратичную поверхность, то их оказывается слишком мало.

Но при любом способе расчета кривизны возникают ошибки, связанные с неравномерным расположением вершин триангуляции и разным размеров треугольников.

3.6.4. Исправление ошибок приближения кривизны

Одна из таких ошибок - это ситуация, когда для вершины типа i все соседние к ней вершины имеют тип j.

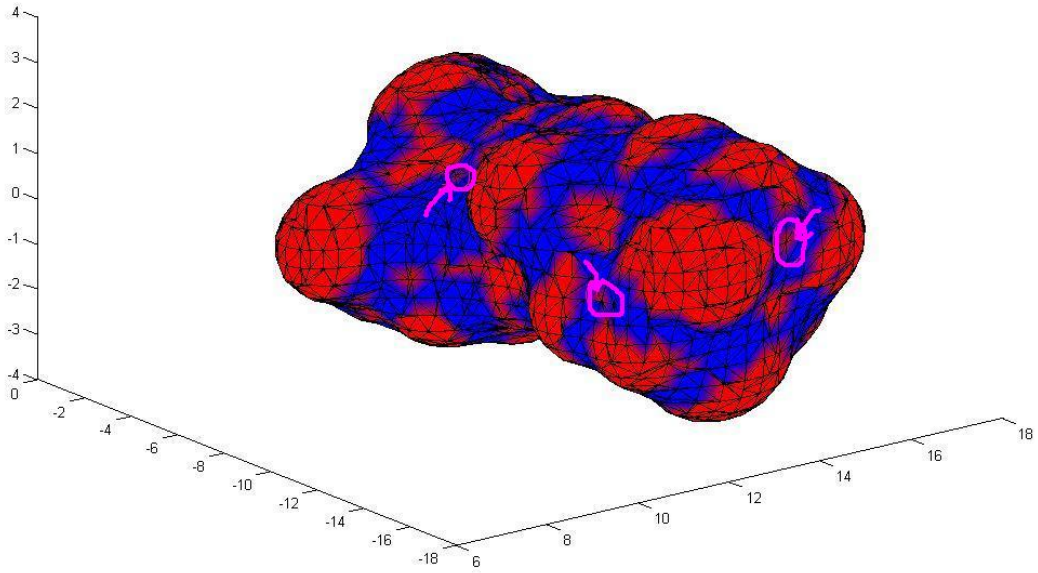


Рис. 27 Ошибки в нахождении кривизны

Такие ошибки устраняются - этой вершине тоже присваивается тип j.

3.6.5. Выделение выпуклостей и впадин

Далее надо сегментировать поверхность так, чтобы в каждом сегменте были вершины одинакового типа. Сначала выделяются связные выпуклые и вогнутые сегменты, потом оставшиеся вершины разделяются на сегменты алгоритмом к-средних.

Как выделять связные сегменты вершин? Есть 2 способа – выделять сегменты, элементами которых являются вершины, или сегменты, элементами которых являются треугольники. Общий алгоритм выделения сегментов одинаков в обоих случаях.

Пусть надо сегментировать поверхность P, состоящую из N элементов (треугольников или вершин), а количество сегментов - k.

Используется следующий алгоритм

Region Growing Algorithm:

```
k=0;
Для (i=1;i<=N;i++)
      Если (P[i] не принадлежит никакому сегменту)
            k++;
            Создать новый сегмент S[k];
            Добавить (P[i] ,S[k]);
```

Функция «*Добавить* (P[i] ,S[k]);» использует следующий алгоритм, где n – количество соседних к P[i] элементов P:

Добавить (P[i] ,S[k]):

```
Добавить P[i] в S[k]
Для (j=1;j<n;j++)
     Если (сосед[P[i]][j] не принадлежит никакому сегменту)
            Добавить (сосед[P[i]][j],S[k]);
```

Сегментировать вершины или треугольники?

При сегментации вершин соседние вершины определяются очевидным образом как связанные ребром.

Чтобы сегментировать треугольники, а не вершины, надо каждому треугольнику присвоить метку и определить, какие треугольники является соседними к данному.

Треугольнику присваивается метка *i*, если все три его вершины имеют метку *i*, иначе ему присваивается метка 3 (соответствующая седловым вершинам). Соседним треугольником для данного является треугольник, имеющий с ним общее ребро.

Если сегментируются вершины, то возникают следующие проблемы, связанные с ошибками при расчете кривизны:

1. У сегментов появляются «отростки».

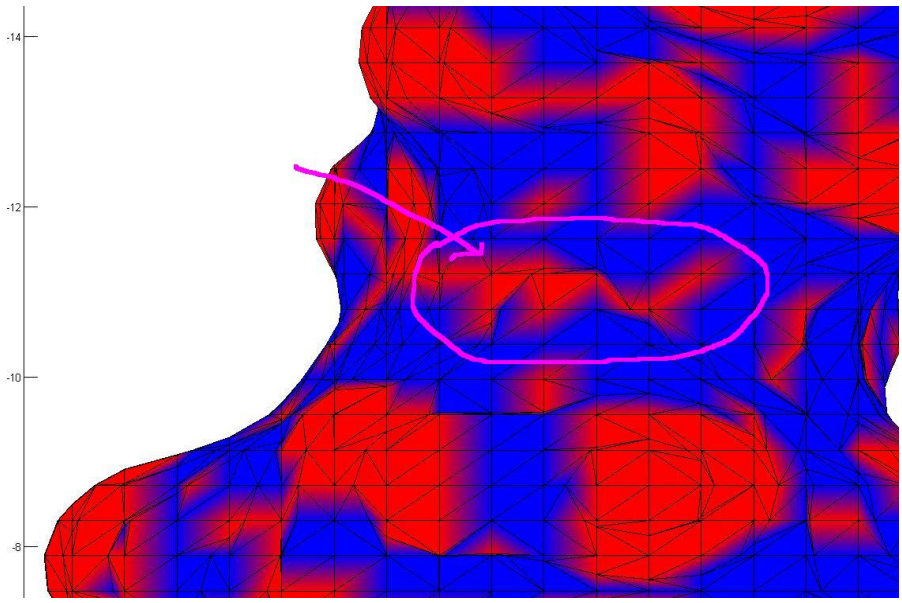


Рис. 28 Отростки

2. Появляются нежелательные связи между выпуклыми сегментами, соответствующими разным сферам - «перемычки» (т.е. при выделении 2 сегмента будут объединены в 1).

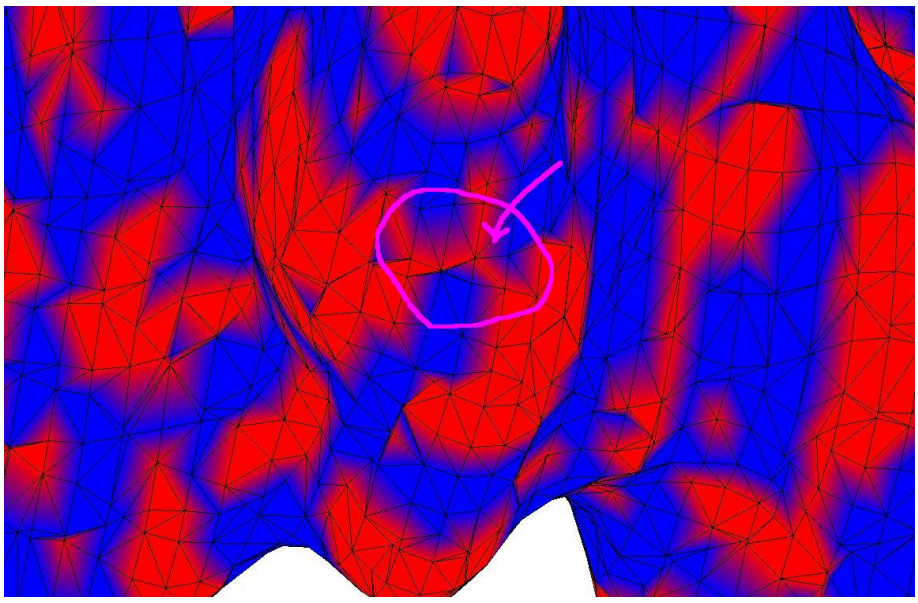


Рис. 29 Перемычки между сегментами

Если сегментировать треугольники, а не вершины, то эти проблемы не возникают. Большинство вершин одного типа в «перемычках» и «отростках» образуют не треугольники, а ломаную линию, поэтому треугольники, содержащие их, не попадают в сегмент.

Результатом данного этапа является множество $S=\{S_i\}$:

$$\begin{cases} S_i = j, \text{если } \exists G_j : v_i \in G_j,\ \text{j} \in 1 \dots N_{s1} \\ S_i = 0 \quad \text{иначе} \end{cases}$$, где N_{s1}- общее количество полученных сегментов.

Пусть n_j - количество вершин в сегменте G_j . Если $n_j < V_{min}$ (в реализации алгоритма V_{min} =5), то G_j удаляется, т.е. элементам массива S, для которых $S_i = j$, присваивается значение 0.

3.6.6. Сегментация вершин седлового типа

Оставшиеся вершины (те, которым в S соответствует 0) – это вершины седлового типа (типа 3). Они в большинстве случаев образуют малое число компонент связности(<5). Обозначим каждую компоненту связанности вершин седлового типа как V_l .Каждая компонента связности V_l сегментируется отдельно.

Для сегментации компоненты связности V_l понадобится следующее:

1. Множество вершин триангуляции $V_l=\{v_{i_l}\}$, $i_l=1 \dots N_l$, заданных своими *3D* координатами $(x_{i_l}, y_{i_l}, z_{i_l})$.
2. Множество окрестностей $O^l_{all} = \{O^l(v_{i_l})\}$, $v_j \in O^l(v_{i_l}) \Rightarrow v_j \in V_l$.
3. Матрица кратчайших расстояний между вершинами $D^l=\{d^l_{ij}\}$,

   $$d^l_{ij} = \begin{cases} dist(v_{i_l}, v_{j_l}), i_l \neq j_l \\ 0,\ i_l = j_l \end{cases}.$$

   где $dist(v_{i_l}, v_{j_l})$ (дистанция между вершинами) рассчитывается алгоритмом Дейкстры с реализацией очереди с приоритетом в виде бинарной кучи [39]. Т.к. все вершины v_{i_l} принадлежат одной компоненте связанности, то посчитать расстояние можно между любыми двумя вершинами.

<u>Выбор алгоритма для сегментации вершин седлового типа</u>

Опробованы 2 варианта сегментации – горная кластеризация [43] и методом *k*-средних [37]. Опишем кратко используемую модификацию каждого из них.

- Алгоритм *k*-средних

Пусть дано множество *d*-мерных векторов $\boldsymbol{X}=(\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_n)$. Кластеризация методом *k*-средних распределяет *n* векторов в *k* множеств ($k \leq n$) $\mathbf{S} = \{S_1, S_2, \ldots, S_k\}$,

$$\mathbf{S} = arg\ min_S[\sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \mu_i\|],$$

где $\boldsymbol{\mu}_i$ – среднее арифметическое координат векторов $x_j \in S_i$.

В нашем случае

1. $\boldsymbol{X}$ – это множество V_l.
2. $k = \frac{N_{s1}}{2}$, где N_{s1}- общее количество полученных сегментов на предыдущем этапе алгоритма.
3. $\exists v_{i_l} \in V_l : v_{i_l} = \mu_i$, т. е. в качестве μ_i выбирается не среднее арифметическое координат $v_j \in S_i$, а вершина v_{i_l}: $v_{i_l} = arg\ min_{v_j \in S_i} \|v_j - \mu_i\|$, наиболее близкая к среднему арифетическому. обозначим эту вершину как $v_{\mu i}$.
4. $\mathbf{S} = arg\ min_S[\sum_{i=1}^{k} \sum_{x_j \in S_i} dist(v_j, v_{\mu i})]$, где $dist(v_j, v_{\mu i})$ – это расстояние, вычисленное алгоритмом Дейкстры, элемент матрицы D^l.

Алгоритм состоит из следующих шагов:

1. Выбираются начальные центры сегментов $v_{\mu i}$. Обычно в реализации алгоритма *k*-средних они выбираются случайно, но в нашем случае это может привести к тому, что при разных запусках программы будут получаться разные сегменты и разные особые точки, что недопустимо. Поэтому $v_{\mu i}$ выбираются по номерам вершин, т.к. нумерация не меняется при разных запусках программы.
2. Каждая вершина $v_j \in V_l$ относится к сегменту S_i:
$$S_i = arg\ min_{v_j \in S_i} dist(v_j, v_{\mu i}).$$
3. Центры сегментов пересчитываются:
$$v_{\mu i} = arg\ min_{v_j \in S_i} \left\| v_j - \frac{1}{n_i} \sum_{x_m \in S_i} v_m \right\|,$$
где n_i- количество вершин в сегменте S_i.

4. Если хотя бы один из центров $v_{\mu i}$изменился во время шага 3 и количество итераций алгоритма меньше константы IT_{max}, то вернуться к шагу 2. Ограничение на количество итераций сделано потому что, вообще говоря, метод может не сойтись или сходится слишком долго при использовании данного алгоритма (т.к. центры на шаге 3 вычисляются «неточно»).

При реализации алгоритма *k*-средних было опробовано 2 варианта расчета центра сегмента $v_{\mu i}$.

1. Центром является вершина, наиболее близкая к среднему арифметическому координат вершин, входящих в сегмент:

$$v_{\mu i} = arg\ min_{v_j \in S_i} \left\| v_j - \frac{1}{n_i} \sum_{x_m \in S_i} v_m \right\|$$

2. Центром является вершина, сумма расстояний от которой до остальных вершин сегмента минимальна:

$$v_{\mu i} = arg\ min_{v_j \in S_i} [\sum_{v_m \in S_i} dist(v_m, v_j)].$$

Визуально при использовании первого варианта результаты получаются лучше, т.е. центр находится приблизительно в геометрическом центре сегмента.

- <u>Горная кластеризация</u>

Шаги алгоритма:

1. Необходимо сформировать множество потенциальных центров сегментов $Z = \{z_h\}$, $h=1...q$. В нашем случае $Z= V_l$.
2. Рассчитывается потенциал центров сегментов по следующей формуле:

$$P(z_h) = \sum_{k=1}^{n_l} \exp\left(-\alpha \cdot dist(z_h, v_k)\right), h = 1 \dots q,$$

где α - положительная константа, n_l- количество вершин в компоненте связанности V_l

$dist(z_h, v_k)$ - расстояние между z_h и v_k, вычисленное алгоритмом Дейкстры, элемент матрицы D^l.

3. В качестве центров сегментов выбирают координаты "горных" вершин. Для этого, центром первого сегмента назначают точку с наибольшим потенциалом $v_{\mu 1}$. Обычно, наивысшая вершина окружена несколькими достаточно высокими пиками. Поэтому назначение центром следующего сегмента точки с максимальным потенциалом среди оставшихся вершин привело бы к выделению большого числа близко расположенных центров сегментов. Чтобы выбрать следующий центр сегмента необходимо вначале исключить влияние только что найденного сегмента. Для этого значения потенциала для оставшихся возможных центров сегментов пересчитываются следующим образом: от текущих значений потенциала вычитают вклад центра только что найденного сегмента (поэтому кластеризацию по этому методу иногда называют субтрактивной). Перерасчет потенциала происходит по формуле:

$P_2(z_h) = P_1(z_h) - P_1(v_{\mu 1}) \cdot \exp(-\beta \cdot dist(z_h, v_{\mu 1}))$,

где $P_1(\cdot)$ - потенциал на 1-й итерации, $P_2(\cdot)$ - потенциал на 2-й итерации, $v_{\mu 1}$ - центр первого найденного сегмента:

$v_{\mu 1} = arg\ max_{z_h \in Z} P_1(z_h)$, β - положительная константа.

Центр второго кластера определяется по максимальному значению обновленного потенциала:

$v_{\mu 2} = arg\ max_{z_h \in Z} P_2(z_h)$.

Затем снова пересчитывается значение потенциалов:

$$P_3(z_h) = P_2(z_h) - P_2(v_{\mu 2}) \cdot \exp(-\beta \cdot dist(z_h, v_{\mu 2}))$$ и т.д.

Итерационная процедура пересчета потенциалов и выделения центров кластеров продолжается до тех пор, пока максимальное значение потенциала превышает некоторый порог.

Сравнение методов

При использовании *k*-средних визуально получается более «равномерная» сегментация, чем при использовании горного алгоритма (в горном алгоритме центрами сегментов становятся вершины, которые инцидентны наиболее коротким ребрам).

3.6.7. Выделение и классификация особых точек на сегментированной поверхности.

После этапа сегментации мы имеем множество $S=\{S_i\}$:

$\boldsymbol{S_i = j}$, если $\exists \boldsymbol{G_j}: \boldsymbol{v_i} \in \boldsymbol{G_j}$, $\boldsymbol{j} \in \boldsymbol{1 \ldots N_s}$, где $\boldsymbol{N_s}$ - общее количество сегментов.

т.е. все вершины принадлежат какому-либо сегменту.

Особые точки ξ_i находятся как вершины, наиболее близкие к среднему арифметическому координат вершин, входящих в сегмент:

$$\xi_i = arg\ min_{v_j \in S_i} \left\| v_j - \frac{1}{n_i} \sum_{x_m \in S_i} v_m \right\|$$

Каждой особой точке ξ_i присваивается метка $L(\xi_i)$, в зависимости от знака потенциала (+\-) в этой точке и типа поверхности (выпуклая, вогнутая, седло). Всего получается 6 типов меток.

Таким образом, построен алфавит дескрипторов A^1, основанный на особых точках, из 6 дескрипторов.

3.7. Алгоритм построения алфавита MS-дескрипторов на парах и тройках особых точек.

3.7.1. Построение алфавита дескрипторов A^2 (классификация пар особых точек).

Пусть дана выборка $\boldsymbol{L}=\{G_i\}$, $i=1,\ldots,m$. N_{vi}- количество вершин в триангуляции молекулярной поверхности i-ой молекулы G_i, N_{ti} – количество треугольников в триангуляции G_i, $N_{tmax} = max_i\ (N_{ti})$, L_{max} - количество дескрипторов в алфавите A^1.

Алфавит дескрипторов A^2 – это алфавит дескрипторов, основанных на парах ОТ (ξ_i, ξ_j). Используется следующий алгоритм:

1. Построить множество ОТ на всех молекулах выборки и алфавит дескрипторов A^1. Пусть Ns_i – количество особых точек на поверхности i-ой молекулы.
2. Формируется одномерный массив D_{all}, в который записываются расстояния между всеми парами особых точек всех молекул выборки:

$D_{all}[l] = ||\xi_i, \xi_j||$, $l=1 \ldots N_{all}$, $\xi i, \xi_j \in G_l$, где

$$N_{all}=\sum_{m=1}^{M_{all}} \frac{Ns_m(Ns_m-1)}{2},$$

N_{all} – общее количество пар особых точек по всей выборке, M_{all}- количество молекул в выборке G, Ns_m - количество особых точек для молекулы G_m

3. Массив D_{all} сортируется алгоритмом быстрой сортировки.
4. Пусть надо классифицировать расстояние на k типов. Составляется множество интервалов $U = \{U_i\}$, $i=1,\ldots,k$, где

   $U_i = (D_{all}[\frac{(i-1)\cdot n}{k}]; D_{all}[\frac{i\cdot n}{k}])$, т.е. отрезок $[0; d_{max}]$ делится на k интервалов.
5. Если $||\xi_i, \xi_j|| \in U_l$, то $L_d(||\xi_i, \xi_j||) = l$.
6. Теперь, пусть дана пара особых точек (ξ_i, ξ_j), $L(\xi_i)=l_i$, $L(\xi_j)=l_j$, $l_i \leq l_j$, $L(\xi_h) \in [1; L_{max}]$ $\forall h$. Тогда $L(\xi_i, \xi_j)$ рассчитывается следующим образом:

$$L(\xi_i, \xi_j) = l_i + L_{max} \cdot l_j + L_d(||\xi_i, \xi_j||).$$

Таким образом, построен алфавит дескрипторов A^2, основанный на парах особых точек, из $\boldsymbol{L_{max}^2 \cdot k}$ дескрипторов.

3.7.2. Построение алфавита дескрипторов A^3 (классификация троек особых точек).

Для классификации троек особых точек (ξ_i, ξ_j, ξ_l) использовался следующий алгоритм:

1. Для выборки G выбирается алфавит дескрипторов A^2 в виде пар особых точек(см. предыдущий пункт) и вычисляется MD-матрица, $MD_{ij} = |\{(\xi_k, \xi_k)$: $\xi_k, \xi_l \in G_i$, $L(\xi_k, \xi_l) =$ j$\}|$.
2. Строится функциональная зависимость $F_i: Ç \rightarrow K$ для каждого дескриптора $a_i \in A^2$.
3. Строятся множества $\tilde{A}=\{a_{i_k}: \varphi(F_{i_k}) \geq \varphi(F_l)\ \forall a_l \notin \tilde{A}\}$, $P=\{(\xi_i,\xi_j)$: $L((\xi_i,\xi_j) \in \tilde{A}\}$. В данной реализации $|\tilde{A}|=10$.
4. Строится множество троек особых точек $\boldsymbol{\Sigma}$следующим образом:
   $\boldsymbol{\Sigma} = \{(\xi_i, \xi_j, \xi_l)$: $(\xi_i, \xi_j) \in P\}$, тройки перечисляются без повторений.
5. Строится множество векторов расстояния $D^3 = \{(||\xi_i, \xi_j||, ||\xi_i, \xi_l||, ||\xi_j, \xi_l||)\}$, без ограничения общности можно считать, $||\xi_i, \xi_j|| \leq ||\xi_i, \xi_l|| \leq ||\xi_j, \xi_l||$.
6. D^3 разделяется алгоритмом k-средних на k кластеров, $D^3=\{D_i^3\}, i = 1, \ldots, k$.
7. Векторам расстояния присваиваются метки по следующему правилу:
   если $d_r = (||\xi_i, \xi_j||, ||\xi_i, \xi_l||, ||\xi_j, \xi_l||) \in D_h^3$, то $L_d(d_r) = h$.

8. Тройкам особых точек (ξ_i, ξ_j, ξ_l) присваиваются метки по следующему правилу:

пусть $||\xi_i, \xi_j|| \leq ||\xi_i, \xi_l|| \leq ||\xi_j, \xi_l||$, $L(\xi_i)=l_i$, $L(\xi_j)=l_j$, $L(\xi_l)=l_l$, $L(\xi_h) \in [1; L_{max}]$ $\forall h$.

Тогда $L(\xi_i, \xi_j, \xi_l) = l_i + L_{max} \cdot l_j + L_{max}^2 \cdot l_l + L_d(||\xi_i, \xi_j||, ||\xi_i, \xi_l||, ||\xi_j, \xi_l||)$.

Таким образом, построен алфавит дескрипторов A^3, основанный на тройках особых точек, из $\boldsymbol{L_{max} \cdot |\widetilde{A}| \cdot k}$ дескрипторов.

4. Анализ алгоритма.

4.1. Анализ асимптотической сложности

4.1.1. Сложность нахождения особых точек на одной молекуле.

Пусть N_v- количество вершин в триангуляции молекулярной поверхности G, N_t – количество треугольников.

Тогда N_e (количество ребер) $=1.5 \cdot N_t$ (в каждом треугольнике содержится 3 ребра и каждое ребро содержится в двух треугольниках), а $N_v < 3 N_t$, т.е $N_v = O(N_t)$.

Проанализируем сложность выполнения основных шагов алгоритма в зависимости от N_t:

1. Составление списков $\{O(v_i)\}$

Выполняется единственным проходом по T. Следовательно, время работы **$O(N_t)$.**

2. Подсчет кривизны.

Для одной вершины v_i выполняется за O(1), значит, для всех вершин время работы $O(N_v) =$ **$O(N_t)$.**

3. Выделение выпуклостей и впадин

В течение работы мы проходим по всем вершинам $v_i \in V$ и для каждой вершины v_i проверяем все вершины $v_j \in O(v_i)$. Т.к. по наблюдениям $|O(v_i)|<10$ $\forall i$, то общее время работы– $O(N_v) =$ **$O(N_t)$.**

4. Расчет расстояний между вершинами седлового типа

Алгоритм Дейкстры с использованием очереди с приоритетом в виде бинарной кучи работает $\Theta(N_e N_v \ln N_v)$ [39] = $\mathbf{\Theta(N_t^2 \ln N_t)}$.

5. Сегментация вершин седлового типа алгоритмом *k*-средних

Каждая итерация работает за $O(N_v) = O(N_t)$. В данном алгоритме количество итераций ограничено константой $IT_{max} = 1000$, а $N_t > 1000$ в подавляющем большинстве случаев, т.е. $IT_{max} = O(N_t)$. Общее время работы - $O(IT_{max} \cdot N_t) = \mathbf{O(N_t^2)}$.

6. Классификация ОТ

Выполняется за один проход по всем вершинам каждого сегмента, т.е. время работы - $O(N_v) = \mathbf{O(N_t)}$.

Таким образом, сложность обработки одной молекулы определяется сложностью расчета расстояний между вершинами седлового типа алгоритмом Дейкстры (шаг 4) - $\boldsymbol{\Theta(N_t^2 ln\ N_t)}$.

4.1.2. Сложность построения алфавита дескрипторов A^2(классификации пар особых точек).

Пусть дана выборка $\boldsymbol{L}=\{G_i\}$, $i=1,\ldots,m$. N_{vi}- количество вершин в триангуляции молекулярной поверхности *i*-ой молекулы G_i, N_{ti} – количество треугольников в триангуляции G_i, $N_{tmax} = max_i(N_{ti})$, Ns_i – количество особых точек на поверхности *i*-ой молекулы. Рассмотрим сложность основных шагов алгоритма:

1. Сложность нахождения особых точек на всех молекулах

Сложность нахождения особых точек на одной молекуле – $\Theta(N_{ti}^2 ln\ N_{ti})$, тогда сложность для всей выборки – $\sum_{i=1}^{m} \Theta(N_{ti}^2 ln\ N_{ti}) = \boldsymbol{m\Theta(N_{tmax}^2 ln\ N_{tmax})}$.

2. Сложность формирования массива расстояний D_{all}

D_{all} формируется одним проходом по всем парам ОТ, значит который записываются расстояния между всеми парами особых точек всех молекул выборки: $|D_{all}|$

$=O(\sum_{i=1}^{m} \frac{Ns_i(Ns_i-1)}{2})$. Т.к. $N_{ti} > Ns_i$ (Обычно $N_{ti} \sim 1000$, $Ns_i \sim 100$), то $Ns_i = O(N_{ti})$ и время работы шага $\sum_{i=1}^{m} O({N_{ti}}^2) = \boldsymbol{mO({N_{tmax}}^2)}$.

3. Сложность сортировки массива расстояний D_{all}

Время работы алгоритма быстрой сортировки – $O(mO({N_{tmax}}^2) ln(mO({N_{tmax}}^2))) = \boldsymbol{mO({N_{tmax}}^2 ln\, N_{tmax})}$.

4. Сложность классификации пар ОТ.

Классификация выполняется одним проходом по всем парам ОТ, значит время работы шага - $\boldsymbol{mO({N_{tmax}}^2)}$.

Таким образом, сложность обработки выборки молекул определяется сложностью нахождения ОТ на всех молекулах выборки (шаг 1) - $\boldsymbol{m\Theta({N_{tmax}}^2 ln\, N_{tmax})}$.

4.1.3. Сложность построения алфавита дескрипторов A^3(классификации троек особых точек).

Пусть дана выборка $\boldsymbol{L}=\{G_i\}$, $i=1,\dots,m$. N_{vi}- количество вершин в триангуляции молекулярной поверхности i-ой молекулы G_i, N_{ti} – количество треугольников в триангуляции G_i, $N_{tmax} = max_i(N_{ti})$, Ns_i – количество особых точек на поверхности i-ой молекулы, $Ns_{max} – max_i(Ns_i)$. Рассмотрим сложность основных шагов алгоритма:

1. Построение алфавита дескрипторов A^2 в виде пар особых точек.

(см. предыдущий пункт) $\boldsymbol{m\Theta({N_{tmax}}^2 ln\, N_{tmax})}$.

2. Построение функциональной зависимости $F_i: G \rightarrow K$ для каждого дескриптора $a_i \in A^2$.

Выполняется с помощью алгоритма, описанного в [46]. Обозначим время выполнения этого этапа, как $\boldsymbol{T_f}$.

3. Построение множеств $\tilde{A}=\{a_{i_k} : \varphi(F_{i_k}) \geq \varphi(F_l)\ \forall a_l \notin \tilde{A}\}$ и $P=\{(\xi_i, \xi_j): L((\xi_i, \xi_j) \in \tilde{A}\}$

Выполняется одним проходом по всем парам ОТ, значит время работы шага - $\boldsymbol{mO({Ns_{max}}^2)}$.

4. Построение множества троек особых точек $\boldsymbol{\Sigma}$:

Из построения алфавита дескрипторов A^2(см *3.7.2.*) *:*

$$|\{(\xi_i, \xi_j)\colon L((\xi_i, \xi_j) \in a_k\}| = |\{(\xi_i, \xi_j)\colon L((\xi_i, \xi_j) \in a_l\}| \; \forall l \; \forall \; k.$$

Поэтому $|P| = \frac{|\tilde{A}|}{|A^2|} \sum_{i=1}^{m} \frac{Ns_i(Ns_i-1)}{2} = m \cdot O(Ns_{max}{}^2)$**.**

Множество троек особых точек для каждой молекулы G_l строится одним проходом по множеству ОТ для каждой пары (ξ_i, ξ_j): $(\xi_i, \xi_j) \in P$ и $(\xi_i, \xi_j) \in G_l$. Поэтому время построения – $O(t_l)$, где t_l – количество построенных троек для молекулы G_l.

$$\sum_{k=1}^{m} t_k < |P| \cdot \sum_{i=1}^{m} Ns_i = \boldsymbol{m^2 \cdot O(Ns_{max}{}^2)}$$

Поэтому время построения множества $\boldsymbol{\Sigma} = \{(\xi_i, \xi_j, \xi_l)\colon (\xi_i, \xi_j) \in P\}$ - $\boldsymbol{m^2 \cdot O(Ns_{max}{}^2)}$**.**

5. Построение множества векторов расстояния $D^3 = \{(\|\xi_i, \xi_j\|, \|\xi_i, \xi_l\|, \|\xi_j, \xi_l\|)\}$

Нужно один раз пройтись по множеству всех троек ОТ - $\boldsymbol{m^2 \cdot O(Ns_{max}{}^2)}$

6. Разделение D^3алгоритмом *k*-средних.

Используется ограничение на количество итераций константой IT_{max}, $IT_{max} = O(N_{tmax})$. Поэтому время работы $O(|D^3| \cdot IT_{max}) = \boldsymbol{m^2 \cdot O(Ns_{max}{}^2 N_{tmax})}$**.**

7. Классификация троек ОТ

Нужно один раз пройтись по множеству всех троек ОТ - $\boldsymbol{m^2 \cdot O(Ns_{max}{}^2 N_{tmax})}$**.**

Таким образом, сложность обработки выборки молекул определяется сложностью построения алфавита пар ОТ A^2 (шаг 1) и разделения множества расстояний алгоритмом *k*-средних (шаг 6) - $\boldsymbol{m\Theta(N_{tmax}{}^2 ln\, N_{tmax}) + m^2 \cdot O(Ns_{max}{}^2 N_{tmax}) + T_f}$**.**

4.2. Оценка количества используемой памяти.

4.2.1. Необходимая память для нахождения особых точек на одной молекуле.

Пусть N_v- количество вершин в триангуляции молекулярной поверхности G, N_t – количество треугольников.

Тогда N_e (количество ребер) =1.5· N_t (в каждом треугольнике содержится 3 ребра и каждое ребро содержится в двух треугольниках), а $N_v < 3\ N_t$, т.е $N_v = O(N_t)$.

Массивы, представляющие множество

Проанализируем необходимую память для основных шагов алгоритма в зависимости от N_t:

0. Входные данные алгоритма:
   - Множество вершин триангуляции $V=\{v_i\}$, $i=1...N_v$, заданных своими *3D* координатами (x_i, z_i, y_i), - $\Theta(N_v) = \boldsymbol{\Theta(N_t)}$.
   - Множество треугольников $T=\{t_i\}$, $i=1... N_t$, заданных своими вершинами (v_{i1}, v_{i2}, v_{i3}), $v_{ik} \in V$, $k=1...3$, - $\boldsymbol{\Theta(N_t)}$.
   - Множество векторов нормалей к исходной поверхности M $N=\{n_i\}$, $i=1... N_v$, заданных своими *3D* координатами (x_i, z_i, y_i) - $\Theta(N_v) = \boldsymbol{\Theta(N_t)}$.

   Всего - **$O(N_t)$, используется в течение всего времени работы алгоритма.**
1. Составление списков $\{O(v_i)\}$

$|\{O(v_i)\}| = \Theta(\Delta(G) \cdot N_v)$, где $\Delta(G) = max_{v \in V_i} d(v)$, $d(v)$ - степень вершины v. Для графа триангуляции G $\Delta(G) < 10$ во всех рассмотренных случаях, поэтому можно считать, что $|\{\Theta(v_i)\}| = \Theta(N_v) = \boldsymbol{\Theta(N_t)}$, **используется в течение всего времени работы алгоритма.**

2. Подсчет кривизны.

Кривизна и типы вершин хранятся в массивах, размер массивов $\Theta(N_v) = \boldsymbol{\Theta(N_t)}$.

3. Выделение выпуклостей и впадин

Массив с номером сегмента для каждой вершины - $\boldsymbol{\Theta(N_t)}$.

4. Расчет расстояний между вершинами седлового типа

Матрица кратчайших расстояний - $\boldsymbol{\Theta(N_t^2)}$.

5. Сегментация вершин седлового типа алгоритмом *k*-средних

Массив с номером сегмента для каждой вершины - $\boldsymbol{\Theta(N_t)}$.

6. Классификация ОТ

Массив с типом вершины для каждой вершины - $\boldsymbol{\Theta}$ **(**N_t**),** *MD*-матрица размерности 6 x *m*

Таким образом, необходимое количество памяти для обработки одной молекулы определяется памятью для матрицы кратчайших расстояний между вершинами седлового типа (шаг 4) - $\boldsymbol{\Theta(N_t^2)}$**.**

4.2.2. Необходимая память для построения алфавита дескрипторов A^2(классификации пар особых точек).

Пусть дана выборка $\boldsymbol{L}=\{G_i\}$, $i=1,\dots,m$. N_{vi}- количество вершин в триангуляции молекулярной поверхности *i*-ой молекулы G_i, N_{ti} – количество треугольников в триангуляции G_i, $N_{tmax}= max_i (N_{ti})$, Ns_i – количество особых точек на поверхности *i*-ой молекулы. Рассмотрим необходимую память для основных шагов алгоритма:

1. Память для нахождения особых точек на всех молекулах

После обработки каждой молекулы выделенная память освобождается, поэтому необходимый объем памяти - $\boldsymbol{\Theta(N_{tmax}^2)}$**.**

2. Память для формирования массива расстояний D_{all}

$|D_{all}| = O(\sum_{i=1}^{m} \frac{Ns_i(Ns_i-1)}{2})$. Т.к. $N_{ti} > Ns_i$ (Обычно $N_{ti} \sim 1000$, $Ns_i \sim 100$), то $Ns_i = O(N_{ti})$ и необходимая память - $\boldsymbol{m\Theta(N_{tmax}^2)}$.

3. Память для классификации пар ОТ.

Пусть количество типов ОТ l_{max} . Тогда память для *MD*-матрицы $\boldsymbol{\Theta(l_{max}^2 \cdot m)}$**.**

Таким образом, количество памяти для обработки выборки молекул определяется количество памяти для формирования массива расстояний D_{all} (шаг 2) - $\boldsymbol{m\Theta(N_{tmax}^2)}$**.**

4.2.3. Необходимая память для построения алфавита дескрипторов A^3(классификации троек особых точек).

Пусть дана выборка $\boldsymbol{L}=\{G_i\}$, $i=1,\ldots,m$. N_{vi}- количество вершин в триангуляции молекулярной поверхности i-ой молекулы G_i, N_{ti} – количество треугольников в триангуляции G_i, $N_{tmax}= max_{\text{i}}\ (N_{ti})$, Ns_i – количество особых точек на поверхности i-ой молекулы, $Ns_{max} – max_{\text{i}}\ (Ns_i)$. Рассмотрим необходимую память для основных шагов алгоритма:

1. Построение алфавита дескрипторов A^2 в виде пар особых точек.

   (см. предыдущий пункт) $\boldsymbol{m\Theta(N_{tmax}{}^2)}$.

2. Построение функциональной зависимости $F_i: G \rightarrow K$ для каждого дескриптора $a_i \in A^2$.

   Выполняется с помощью алгоритма, описанного в [46]. Обозначим память выполнения этого этапа, как $\boldsymbol{M_f}$.

3. Построение множеств $\tilde{A}=\{a_{i_k}: \varphi(F_{i_k}) \geq \varphi(F_l)\ \forall a_l \notin \tilde{A}\}$ и $P=\{(\xi_i,\xi_j): L((\xi_i,\xi_j) \in \tilde{A}\}$

Из построения алфавита дескрипторов A^2(см *3.7.2.*) *:*

$$|\{(\xi_i,\xi_j): L((\xi_i,\xi_j) \in a_k\}| = |\{(\xi_i,\xi_j): L((\xi_i,\xi_j) \in a_l\}|\ \forall l\ \forall\ k.$$

Поэтому $|P| = \frac{|\tilde{A}|}{|A^2|}\sum_{i=1}^{m}\frac{Ns_i(Ns_i-1)}{2} = \boldsymbol{m \cdot \Theta(Ns_{max}{}^2)}$.

4. Построение множества троек особых точек $\boldsymbol{\Sigma}$:

t_l – количество построенных троек для молекулы G_l

$$\sum_{k=1}^{m} t_k < |P| \cdot \sum_{i=1}^{m} Ns_i = \boldsymbol{m^2 \cdot \Theta(Ns_{max}{}^2)}$$

Поэтому размер $\boldsymbol{\Sigma} = \{(\xi_i,\xi_j,\xi_l): L((\xi_i,\xi_j) \in \tilde{A}\}$ - $\boldsymbol{m^2 \cdot \Theta(Ns_{max}{}^2)}$.

5. Построение множества векторов расстояния $D^3 = \{(||\xi_i, \xi_j||, ||\xi_i, \xi_l||, ||\xi_j, \xi_l||)\}$
   $|D^3| = |\boldsymbol{\Sigma}| = \boldsymbol{m^2 \cdot \Theta(Ns_{max}{}^2)}$
6. Разделение D^3алгоритмом k-средних.

Дополнительная память не требуется.

7. Классификация троек ОТ

Пусть количество типов ОТ l_{max} . Тогда память для MD-матрицы $\boldsymbol{l_{max}{}^3 \cdot m}$

Таким образом, необходимая память для обработки выборки молекул –

$$\boldsymbol{m^2 \cdot \Theta(Ns_{max}{}^2) + M_f}$$

4.3. Возможные улучшения алгоритма.

- Расширить список свойств, по значению которых присваивается метка особой точке

В настоящий момент метка присваивается только на основе кривизны поверхности и знака потенциала, что, конечно, не отражает всех свойств молекулы. Возможно использование таких свойств, как значение молекулярного поля [52], поля APF [53], потенциала липофильности [53], гидрофобности.

- Учет глобальной формы поверхности

В настоящий момент в алгоритме учитывается только локальная форма поверхности (кривизна), возможно учет глобальной формы поверхности улучшит результаты. Возможно использование дескрипторов формы [32], расстояния до выпуклой оболочки [33][18], функции Коннолли [27][28], функции атомной плотности [29].

- Уменьшение времени работы алгоритма

За счет уменьшения времени подсчета расстояний между вершинами триангуляции за счет распараллеливания, т.к. асимптотическая сложность алгоритма определяется именно этим этапом.

- Построение нечетких дескрипторов

Зависимость значения качества прогноза от параметров построения дескрипторов будет непрерывной, и для нахождения оптимального набора параметров можно применить метод градиентного спуска. Этой теме посвящена работа [54].

- Построение масштабируемых дескрипторов

Ввести изменяемый параметр – масштаб, от которого зависит количество особых точек. Масштабируемая сегментация молекулы на основе использования комплекса Морса-Смейла описана в работе [31] (см. рисунок 14)

5. Результаты

5.1. Изменяемые параметры алгоритма

На даный момент в алгоритме есть следующие изменяемые параметры:

1. Выбор алфавита дескрипторов.

Можно выбрать A- алфавит особых точек, A^2 - алфавит пар особых точек, A^3- алфавит троек особых точек. Влияет на формирование MD-матрицы.

2. Количество интервалов, на которые разбивается отрезок расстояний между ОТ $[0;d_{max}]$.

Влияет на формирование алфавита пар ОТ $\boldsymbol{A^2}$ и алфавита троек ОТ $\boldsymbol{A^3}$.

3. Количество дескрипторов в множестве $\tilde{A}=\{a_{i_k}: \varphi(F_{i_k}) \geq \varphi(F_l)\ \forall a_l \notin \tilde{A}, a_l \in A^2\}$.

Влияет на формирование алфавита троек ОТ $\boldsymbol{A^3}$.

4. Количество кластеров k, на которые разбивается множество троек расстояний D_{all}

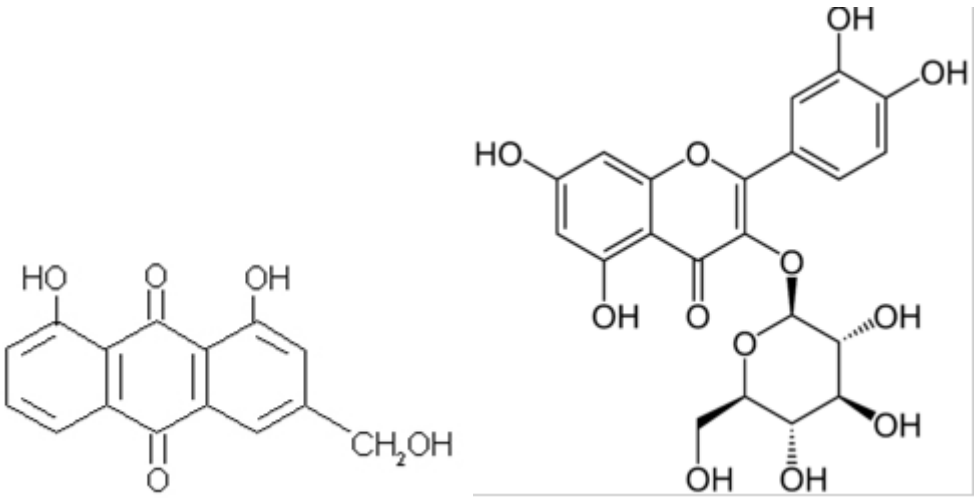
Влияет на формирование алфавита троек ОТ $\boldsymbol{A^3}$.

Приведем результаты тестов на выборках. В качестве функционала качества использовался значения функционала метод перекрестного контроля q^2.

5.2. Выборка гликозидов

Гликозиды - органические соединения, молекулы которых состоят из двух частей: углеводного (пиранозидного или фуранозидного) остатка и неуглеводного фрагмента. В качестве гликозидов в более общем смысле могут рассматриваться и углеводы, состоящие из двух или более моносахаридных остатков. Преимущественно кристаллические, реже аморфные вещества, хорошо растворимые в воде и спирте. Гликозиды представляют собой обширную группу органических веществ, встречающихся в растительном (реже в животном) мире и/или получаемых синтетическим путём. Соединения нужно было

разбить на 2 класса - *активные* и *неактивные*. Приведем химические формулы некоторых гликозидов (рис. 28).



Алоэ – эмодин Гликозид кверцетин

Рис. 30 Химические формулы гликозидов

Выборка состоит из 110 соединений.

	Качество прогноза	Выбросы
1	0.7962	2
2	0.8454	0
3	0.8113	4
4	0.8504	3
5	0.8504	3
6	0.7909	0
7	0.8055	2
8	0.8545	0

Таблица 1. Влияние дискретизации расстояний между особами точками на качество прогноза (выборка гликозидов)

Видно, что нет определенной зависимости качества прогноза от количества интервалов разбиения расстояния, что говорит в пользу построения нечетких дескрипторов.

5.3. Выборка токсичных соединений

Химическая токсичность может вызывать множество биологически опасных эффектов, таких как повреждение генов, или даже привести смерти человека или животных. Около 120000 химических соединений будут протестированы в ближайшие 10 лет. Для этого потребуется до 45 миллионов лабораторных животных. Альтернативой может послужить прогноз токсичности на основе моделей построенных на уже протестированных соединениях. Исследуемая нами выборка поделена на две части. Первая из 644 молекул

предназначена для построения моделей, а вторая (449 соединений) для их проверки. Активность в отличие от предыдущих выборок не является дискретной. Ниже приведена таблица с результатами.

	Качество прогноза
1	0.49
2	0.48
3	0.44
4	0.51
5	0.52
6	0.45
7	0.49
8	0.51

Таблица 2. Влияние дискретизации расстояний между особами точками на качество прогноза(выборка токсичных соединений)

Из таблицы видно, что качестово погноза не очень хорошее, что говорит о том, что используемых в построении дескрипторов свойств (электростатический потенциал) недостаточно для предсказания токсичности. Также видно, что нет определенной зависимости качества прогноза от количества интервалов разбиения расстояния, что говорит в пользу построения нечетких дескрипторов.

6. Заключение.

В дипломной работе был разработан и программно реализован алгоритм построения структурных 3d дескрипторов, учитывающих пространственную форму и свойства молекулы. Так же было проведено тестирование алгоритма, проверенна эффективность дескрипторов, отмечена пригодность полученных моделей для прогнозирования активности новых соединений.

Приведем также результаты прогноза на исследованных выборках:

1. Выборка гликозидов

Лучший результат – 0.8504

2. Выборка токсичных соединений

Лучший результат - 0.52

7. Список литературы

1. Wiener H. Structural determination of paraffin boiling points. J Am Chem Soc 1947, vol.69, pp.17–20
2. Kumskov M.I., Zyryanov I.L., Svitan'ko I.V. A New Method for Representing Spatial Electronic Structures of Molecules in the Problem of Structure-Biological Activity Relationship. Pattern Recognition and Image Analysis, 1995, n.3, pp.477-484.
3. Кумсков М.И., Пономарева Л.А., Смоленский Е.А., Митюшев Д.Ф., Зефиров Н.С. Метод автоматического формирования структурных дескрипторов органических соединений для количественных корреляций «структура-активность». Изв. РАН, серия Химическая, 1994, с.1391-1393.
4. Randic M. On Characterization of Molecular Branching. Journal of the American Chemical Society, 1975, vo.97, pp.6609-6615.
5. Carhart R et al. Atom Pairs as Molecular Features in Structure-Activity Studies: Definition and Applications. J. Chem. Inf. Comput. Sci.; 1985; 25(2) pp 64–73
6. Makeev G.M., Kumskov M.I., Svitan'ko I.V., Zyryanov I.L. Recognition of Spatial Molecular Shapes of Biologically Active Substances for Classification of Their Properties. Pattern Recognition and Image Analysis, 1996, v.6, n.4. p.795-808.
7. Кумсков М.И., Смоленский Е.А., Пономарева Л.А., Митюшев Д.Ф., Зефиров Н.С. Системы структурных дескрипторов для решения задач «структура-активность». Доклады Академии Наук, 1994, 336, п.1., с.64-66.
8. Кохов В. А. Метод количественного определения сходства графов на основе структурных спектров // Известия РАН, Техническая Кибернетика. - 1994. - №5. - С.,143-159.
9. Michel Sanner, Arthur J. Olson, Jean Claude Spehner (1996). *Reduced Surface: an Efficient Way to Compute Molecular Surfaces*. Biopolymers, Vol 38, (3), 305-320. http://mgltools.scripps.edu/downloads#msms
10. http://accelrys.com/products/discovery-studio/visualization-download.php
11. Matthieu Chavent, Bruno Levy, Bernard Maigret "MetaMol: High quality visualization of Molecular Skin Surface"
12. SANDER P., SNYDER J., GORTLER S., HOPPE H.:"Texture mapping progressive meshes." In Proceedings of ACM SIGGRAPH (2001), pp. 409–416.
13. SHLAFMAN S., TAL A., KATZ S.: "Metamorphosis of polyhedral surfaces using decomposition." Computer Graphics forum 21, 3 (2002). Proceedings Eurographics 2002.

14. E. Kalogerakis, A. Hertzmann, K. Singh "Learning 3D Mesh Segmentation and Labeling", TOG 29{3}, Siggraph 2010, ACM Transactions on Graphics 29(3), July 2010.
15. Guillaume Lavoue ´, Florent Dupont, Atilla Baskurt "A new CAD mesh segmentation method, based on curvature tensor analysis" Computer-Aided Design 37 (2005) 975–987.
16. Ariel Shamir "A survey on Mesh Segmentation Techniques" COMPUTER GRAPHICS forum Volume 27 (2008), number 6 pp. 1539–1556.
17. M. Attene S. Katz M. Mortara G. Patan´ e M. Spagnuolo A. Tal "Mesh segmentation – A comparative study" IEEE International Conference on Shape Modeling and Applications 2006 (SMI'06)
18. Vijay Natarajan, Patrice Koehl, Yusu Wang, Bernd Hamann "Visual Analysis of Biomolecular Surfaces Mathematics and Visualization", 2008, V, 237-255.
19. Thitiwan Srinark and Chandra Kambhamettu "A NOVEL METHOD FOR 3D SURFACE MESH SEGMENTATION" Proceedings of the 6th IASTED International Conference on Computers, Graphics, and Imaging.
20. T. E. Exner, M. Keil, J. Brickmann. "Pattern recognition strategies for molecular surfaces. "I. Pattern generation using fuzzy set theory. J. Comput. Chem., 23:1176–1187, 2002.
21. W. Heiden and J. Brickmann. "Segmentation of protein surfaces using fuzzy logic. "J. Mol. Graphics., 12:106–115, 1994.
22. Tatiana Surazhsky , Evgeny Magid , Octavian Soldea ,Gershon Elber, Ehud Rivlin "A Comparison of Gaussian and Mean Curvatures Estimation Methods on Triangular Meshes Computer", Vision and Image Understanding Volume 107 , Issue 3 (September 2007).
23. SYLVAIN PETITJEAN LORIA-CNRS & INRIA Lorraine "A Survey of Methods for Recovering Quadrics in Triangle Meshes", ACM Computing Surveys (CSUR) Volume 34 , Issue 2 (June 2002) Pages: 211 – 262.
24. Christian Hofbauer, Hans Lohninger, András Aszódi "SURFCOMP: A Novel Graph-Based Approach to Molecular Surface Comparison", *J. Chem. Inf. Comput. Sci.*, 2004, 44 (3), pp 837–847.
25. F. Cazals, F. Chazal, T. Lewiner "Molecular Shape Analysis based upon the Morse-Smale Complex and the Connolly Function" Annual Symposium on Computational Geometry, Proceedings of the nineteenth annual symposium on Computational

geometry, San Diego, California, USA, SESSION: Topology, Pages: 351 – 360, Year of Publication: 2003

26. Cohen-Steiner D, Morvan J. “Restricted delaunay triangulations and normal cycle.” 19th Annual ACM Symposium on Computational Geometry; 2003. p. 237–46.
27. M. L. Connolly. “Measurement of protein surface shape by solid angles.” J. Mol. Graphics, 4, 1986.
28. M. L. Connolly. “Shape complementarity at the hemoglobin a1b1 subunit interface.” Biopolymers, 25, 1986.
29. Mitchell, J.C., Kerr, R., Eyck, L.F.T., 2001. “Rapid atomic density measures for molecular shape characterization.” J. Mol. Graph. Model. 19, 324–329.
30. http://www.mitchell-lab.org/mitchell-lab/download/fadepadre.tar.gz
31. Vijay Natarajan,Yusu Wang, Peer-Timo Bremer,Valerio Pascucci, Bernd Hamann “Segmenting molecular surfaces” Computer Aided Geometric Design 23 (2006) 495–509
32. Gregory Cipriano, George N. Phillips, Michael Gleicher “Multi-Scale Surface Descriptors”, IEEE Trans Vis Comput Graph. 2009 Nov–Dec; 15(6): 1201–1208.
33. J. Lien and N. M. Amato. “Approximate convex decomposition of polyhedra.” Technial Report TR05-001, Texas A&M University, 2005.
34. R. Forman. “Morse theory for cell complexes.” Advances in Mathematics, 134:90–145, 1998.
35. Bremer, P.-T., Edelsbrunner, H., Hamann, B., Pascucci, V., 2004. “A topological hierarchy for functions on triangulated surfaces.” IEEE Trans. Visual. Comput. Graph. 10 (4), 385–396.
36. Edelsbrunner H., Harer J., Zomorodian A., 2003b. “Hierarchical Morse–Smale complexes for piecewise linear 2-manifolds.” Discrete Comput. Geom. 30 (1), 87–107
37. Zheng W., Tropsha A. Novel variable selection quantitative structure-property relationship approach based on the k-nearest-neighbour principle J.Chem.Inf.Comput.Sci., vol.40, pp.185-194, 2000.
38. «A Novel Cubic-Order Algorithm for Approximating Principal Direction Vectors»,JACK GOLDFEATHER(Carleton College) and VICTORIA INTERRANTE(University of Minnesota). ACM Transactions on Graphics (TOG), Volume 23 Issue 1, January 2004
39. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн «Алгоритмы: построение и анализ»,2-е издание, стр.680
40. M. L. Connolly, “Analytical molecular surface calculation”, J. Appl. Crystallogr. 16 (1983) 548–558.

41. http://molekel.cscs.ch/wiki/pmwiki.php
42. http://www.ceng.metu.edu.tr/~tcan/LSMS/
43. С.Д.Штовба «Введение в теорию нечетких множеств и нечеткую логику», 12.2.1.
44. Дж. Ту, Р. Гонсалес «Принципы распознавания образов», изд. «Мир», 1978.
45. I.V. Svitanko, D.A. Devetyarov, D.E. Tcheboukov, M. S. Dolmat, A.M. Zakharov, S.S. Grigoryeva, V.T. Chichua, L.A. Ponomareva, M.I. Kumskov. QSAR Modeling on the Basis of 3D Descriptors Representing the Electrostatic Molecular Surface (Ambergris Fragrances) // Mendeleev Communications. – 2007. – Vol.17, No. 2. – P. 90-91.
46. A. V. Perevoznikov, A. M. Shestov, E. A. Permyakov, and M. I. Kumskov «A Way to Increase the Prediction Quality for the Large Set of Molecular Graphs by Using the *k*–NN Classifier», Pattern Recognition and Image Analysis, 2011, Vol. 21, No. 2, pp. 524–526.
47. A. V. Bekker, A. A. Suleymanov, G. N. Apryshko, and M. I. Kumskov «The Multilevel Adaptive Description of Molecular Graphs in "Structure_Property Problem"», Pattern Recognition and Image Analysis, 2011, Vol. 21, No. 2, pp. 438–441.
48. Richard D. Cramer, III, David E. Patterson, and Jeffrey D. Bunce, J . «Comparative Molecular Field Analysis (CoMFA). 1. Effect of Shape on Binding of Steroids to Carrier Proteins» Am. Chem. SOC. 1988, 110, 5959-5967.
49. Gerhard Klebe «Comparative Molecular Similarity Indices Analysis: CoMSIA», Three-Dimensional Quantitative Structure Activity Relationships, 2002, Volume 3, Part I, 87-104
50.
51. Makeev G.M., Kumskov M.I., Svitan'ko I.V., Zyryanov I.L. Recognition of Spatial Molecular Shapes of Biologically Active Substances for Classification of Their Properties. Pattern Recognition and Image Analysis, 1996, v.6, n.4. p.795-808.
52. Halgren, T.A.:Merck molecular force field, V. JComp Chem, 17, 490-641. (1996)
53. Totrov, M.:Atomic Property Fields: Generalized 3D Pharmacophoric Potential for Automated Ligand Superposition, Pharmacophore Elucidation and 3D QSAR,Chem Biol Drug Des, 71, 15-27(2008)
54. Деветьяров Д.А. Эволюционное построение алфавита дескрипторов, сформированных на основе аппарата нечеткой логики, в задаче «структура-свойство» // Системы управления и информационные технологии. – 2010. – 2010. – № 1.1 (39). – С. 131-134.
55. http://www.vcclab.org/lab/indexhlp/
56. http://www.talete.mi.it/products/dragon_description.htm

57. Kim K. H., Greco G., Novellino E. A Critical Review of Recent CoMFA Applications. 3D QSAR in Drug Design, Kubinyi, H.; Folkers, G.; Martin, Y. C. (Eds), Kluwer Academic Publishers, Great Britain vol. 3, p. 257., 1998.
58. Cho S.J., Tropsha A. Cross-Validated R2-Guided Region Selection for Comparative Molecular Field Analysis: A Simple Method to Achieve Consistent Results. J.Med.Chem., 1995, v.38, pp.1060-1066.

8. Приложение.

Описание основных модулей.

pairs.cpp

Строит MD-матрицу для выборки молекул. Параметры считываются из файла. Пример файла с параметрами:

----

mol_package="D:\pirim\"
mol_prefix="pirim"
mol_number="205"
d_number="7"
folder_for_mol "1"
format "mol"

Все параметры указываются обязательно в кавычках, для mol_package должен быть обратный слэш в конце. mol_package – папка, в которой находятся файлы, содержащие молекулярные поверхности и заряды, mol_prefix – префикс имени файла перед его номером, mol_number – количество молекул в выборке, d_number- количество интервалов разбиения расстояния между особыми точками. Остальные параметры используются для построения молекулярной поверхности.

параметры:

filename – имя файла с параметрами.

Вход

1.txt - фаил с параметрами

Посчитанные молекулярные поверхности и заряды на атомах-
файлы mol_package/srfi.wrl и mol_package/chi.wrl

Выход

mol_prefix.mdd_number– MD- матрица, mdd_number – количество интервалов разбиения дистанции между точками.

mol_prefixi.sng – файл, содержащий метки особых точек и их координат iой молекулы.

mol_prefixi.dist - файл, содержащий списки расстояний между особыми точками iой молекулы.

singular_points.cpp.

Рассчитывает особые точки для одной молекулы .

Параметры:

mol_package - папка, в которой находятся файлы, содержащие молекулярные поверхности и заряды

mol_prefix - – префикс имени файла перед его номером

i - номер молекулы

Вход

mol_package/srfi.wrl - фаил с молекулярной поверхностью для iой молекулы

mol_package/chi.wrl - фаил с зарядами атомов и их координатами для iой молекулы

Выход

labels - массив меток особых точек

points – массив координат особых точек.

Листинг модулей.

pairs.cpp

```
#include <iostream.h>
#include <istream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include<math.h>
#include"singular_points.cpp"
int comp (const void*a,const void*b)
{
      double r=*(double*)a-*(double*)b;
       if (r>0)
              return 1;
       else if (r==0)
                        return 0;
           else
             return -1;
}

int main ()
{
       ifstream fin;
       ofstream fout;
       char buf[100],filename[100];

char**opt,**param,*mol_package,*mol_prefix,sng[]=".sng",dst[]=".dst",d[]=".d",md[]=".md"
;
       double**points,*temp,*dist_all,*borders;
        int*labels,**MD;
        int size,size_all=0,usl=0;
       int beg,end,i,mol_number,d_number;
       FILE*f;

       temp=rzeros(temp,3);
       opt=new char*[4];
      param=new char*[4];

       for (int j=0;j<4;j++)
       {
                opt[j]=new char[12];
        param[j]=new char[100];
      }

       strcpy(filename,"1.txt");
       strcpy(opt[0],"mol_package");//нужен слэш в конце
       opt[1]="mol_prefix";
      opt[2]="mol_number";
      opt[3]="d_number";
        fin.open(filename);
```

```
  cout<<filename<<"\n";
  if(fin.fail())
  {
        cerr<<"invalid filename";
        return 0;
}

while (!fin.eof())
  {
        fin.getline(buf,sizeof(buf));

        for (i=0;i<4;i++)
              if (strstr(buf,opt[i])!=0)
                break;
        if (i<4)//остальные параметры не учитываются, они для построения
поверхности
          {
             for (beg=0;beg<100;beg++)
              if (buf[beg]=='"')
                break;

        beg++;

        for (end=beg;end<100;end++)
              if (buf[end]=='"')
                break;

        end--;

        for (int j=beg;j<=end;j++)
          param[i][j-beg]=buf[j];

              param[i][end-beg+1]='\0';
        }
}

fin.close();
mol_package=new char[sizeof(param[0])];
mol_prefix=new char[sizeof(param[1])];
strcpy(mol_package,param[0]);
strcpy(mol_prefix,param[1]);
mol_number=atoi(param[2]);
  d_number=atoi(param[3]);

  cout<<"особые точки посчитаны? (1 - да,0 - нет)\n";
cin>>usl;
if (!usl)
      for (int j=1;j<=mol_number;j++)
      {
            cout<<j<<"\n" ;
        singular_points(points,labels,&size,mol_package,mol_prefix,j);
        size_all+=(size*(size-1))/2;
```

```
                filename[0]='\0';
                sprintf(filename,"%s%s%d%s",mol_package,mol_prefix,j,sng);
                cout<<filename<<"\n";
                fout.open(filename);

                if(fout.fail())
                {
                 cerr<<"invalid filename";
                 return 0;
        }
                fout<<size<<"\n";

                for (int k=0;k<size;k++)
                {
                        for (int l=0;l<3;l++)
                                fout<<points[k][l]<<" ";

                        fout<<labels[k]<<"\n";
                }

                fout.close();
                filename[0]='\0';
                 sprintf(filename,"%s%s%d%s",mol_package,mol_prefix,j,dst);
                 cout<<filename<<"\n";
                fout.open(filename);

                 if(fout.fail())
                 {
                                        cerr<<"invalid filename";
                                        return 0;
        }

        fout<<(size*(size-1))/2<<"\n";

        for (int k=0;k<size-1;k++)
                for (int l=k+1;l<size;l++)
                 {
                                if (labels[l]>labels[k])
                              fout<<labels[k]<<" "<<labels[l]<<" ";
              else
                fout<<labels[l]<<" "<<labels[k]<<" ";

                                minus(temp,points[l],3,points[k],3);

                                fout<<norma(temp,3)<<"\n";
                                zeros(temp,3);
        }

                        fout.close();
                        destroy(points,size);
                        delete[] labels;
}
```

```
else
    for (int j=1;j<=mol_number;j++)
        {
            cout<<j<<"\n";

            filename[0]='\0';
            sprintf(filename,"%s%s%d%s",mol_package,mol_prefix,j,dst);
            cout<<filename<<"\n";
            f=fopen(filename,"r");
        fscanf(f,"%d",&size);
        size_all+=size;
        fclose(f);
        }

  dist_all=rzeros(dist_all,size_all);
  size_all=0;

  for (int j=1;j<=mol_number;j++)
        {
            filename[0]='\0';

            filename[0]='\0';
            sprintf(filename,"%s%s%d%s",mol_package,mol_prefix,j,dst);
            cout<<filename<<"\n";
            f=fopen(filename,"r");
      fscanf(f,"%d",&size);

        int l1,l2;
        for (int k=0;k<size;k++)
            fscanf(f,"%d %d %lf",&l1,&l2,&dist_all[size_all+k]);

            size_all+=size;
        fclose(f);
        }

      qsort(dist_all,size_all,sizeof(double),(*comp));
      borders=rzeros(borders,d_number-1);

      for (int j=0;j<d_number-1;j++)
            borders[j]=dist_all[(int)floor((j+1)*size_all/d_number)];

   filename[0]='\0';
sprintf(filename,"%s%s%s%d",mol_package,mol_prefix,d,d_number);
cout<<filename<<"\n";
fout.open(filename);

   if (fout.fail())
   {
                cerr<<"invalid filename";
                return 0;
   }
```

```
        for (int j=0;j<d_number-1;j++)
            fout<<borders[j]<<"\n";

        delete[] dist_all;
              MD=rzeros(MD,mol_number,36*d_number);
      //cin>>i;

      for (int j=1;j<=mol_number;j++)
      {
                    filename[0]='\0';
                sprintf(filename,"%s%s%d%s",mol_package,mol_prefix,j,dst);
                cout<<filename<<"\n";
                f=fopen(filename,"r");
        fscanf(f,"%d",&size);

                int l1,l2,i=0;
                double r;

                    for (int k=0;k<size;k++)
                    {
                            fscanf(f,"%d %d %lf",&l1,&l2,&r);
                            i=0;
                while ((i<d_number-1)&&(r>borders[i])) i++;
                MD[j-1][l1-1+6*(l2-1)+36*i]++;
                    }

                    fclose(f);
            }
     filename[0]='\0';
   sprintf(filename,"%s%s%s%d",mol_package,mol_prefix,md,d_number);
   cout<<filename<<"\n";
     print_f(MD,mol_number,36*d_number,filename);

     delete[] mol_package;
     delete[] mol_prefix;
     delete[]temp;
     delete[]borders;
     destroy(MD,mol_number);
     cin>>i;
     return 1;
}
```

singular_points.cpp

```
#include <iostream.h>
#include <istream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include <math.h>
#include "curvature.cpp"
#include "segmentation.cpp"
```

```
#include "marking.cpp"
#include "surface_read.cpp"



void singular_points(double**&points,int*&labels,int *size,const char*mol_package,const
char*mol_prefix,const int mol_number)
{
    double **Vertices,**Normals;
   double **H;
    int      **Triangles;
    int **neighbours_vv;
    int **neighbours_tt;
    int *colour_matrix,*Segments,*numbers,*centers;
    int max_size=250,i;
    char filename[200],filename1[200],srf[]="srf",ch[]="ch",wrl[]=".wrl",txt[]=".txt";
    sprintf(filename,"%s%s%d%s",mol_package,srf,mol_number,wrl);
    cout<<filename<<"\n";
    surface_read(filename,Vertices, Triangles, Normals);
    int vn=_msize(Vertices)/sizeof(Vertices[0]);
    int fn=_msize(Triangles)/sizeof(Triangles[0]);

    H=rzeros(H,vn,2);
    colour_matrix=rzeros(colour_matrix,vn);
    Segments=rzeros(Segments,vn);
    numbers=rzeros(numbers,3);

    attributes(Vertices,vn,Triangles,fn,neighbours_vv,neighbours_tt);

    curvature_cubic(H,Vertices,vn,Normals,neighbours_vv);

    for (int i=0;i<vn;i++)
            if (H[i][1]>=0)//средняя кривизна
              colour_matrix[i]=1;

    curvature_parabolloid(H,Vertices,vn,Normals,neighbours_vv);



    for (int i=0;i<vn;i++)
            if (colour_matrix[i]==0)
              if (H[i][0]>=0)
                    colour_matrix[i]=-1;
              else
                     colour_matrix[i]=0;



    //1 - впадина,-1 - выпуклый, 0 - седло

segmentation(Segments,numbers,Vertices,vn,Triangles,fn,neighbours_vv,neighbours_tt,colour_
matrix,max_size) ;
```

```
        //print_f(numbers,3,"numbers.txt");
        //print_f(Segments,vn,"Segments.txt");
        //print_f(Vertices,vn,3,"Vertices.txt");
        //print_f(Triangles,fn,3,"Triangles.txt");

        labels=rzeros(labels,numbers[2]);
        centers=rzeros(centers,numbers[2]);
        sprintf(filename1,"%s%s%d%s",mol_package,ch,mol_number,txt);
        marking(labels, centers,filename1,Segments,numbers,Vertices,vn);
        points=rzeros(points,numbers[2],3);
        for (int i=0;i<numbers[2];i++)
                for (int j=0;j<3;j++)
                        points[i][j]=Vertices[centers[i]][j];

        *size=numbers[2];
       destroy1(Vertices,vn);
       destroy1(Triangles,fn);
       destroy1(Normals,vn);
       destroy(H,vn);

       destroy(neighbours_vv,vn);
       destroy(neighbours_tt,fn);
       delete[]colour_matrix;
       delete[]Segments;
       delete[]numbers;
       delete[]centers;

       //cin>>i;
       return ;
   }
```

surface_read.cpp

```
   #include <iostream.h>
   #include <istream.h>
   #include <fstream.h>
   #include <string.h>
   #include <stdlib.h>
   #include <malloc.h>
   #include<math.h>

   void surface_read(const char*filename, double**&Vertices, int**&Triangles,
double**&Normals)
   {
        Vertices=(double**)malloc(sizeof(double*)) ;
        Triangles=(int**)malloc(sizeof(int*));

        int n=1;
        char buf[60];
        ifstream f;

        f.open(filename);
        if(f.fail())
```

```
  {
            cerr<<"invalid filename";
            return;
}
f.getline(buf,sizeof(buf));

while (strstr(buf, "Triangle")==0&&(!(f.eof())))
                  f>>buf;



while (strstr(buf, "point")==0&&!f.eof())
                  f.getline(buf,sizeof(buf));

int i=0;

  while(f.rdstate()==0)
  {
            Vertices[i]=(double*)malloc(3*sizeof(double)) ;

    for  (int j=0;j<3;j++)
                     f>>(Vertices[i][j]);


                  f>>buf;
                  i++;
                  if (i+1>n)
                  {
        n+=1000;
                     Vertices=(double**)realloc(Vertices,n*sizeof(double*));
    }
}

int vn=i-1;
  Vertices=(double**)realloc(Vertices,vn*sizeof(double*));
  f.clear();

while (strstr(buf, "# Normal definition")==0&&!f.eof())
                  f.getline(buf,sizeof(buf));

i=0;
f.getline(buf,sizeof(buf));
f.getline(buf,sizeof(buf));
Normals=(double**)malloc(vn*sizeof(double*));

  for (int i=0;i<vn;i++)
  {
                  Normals[i]=(double*)malloc(3*sizeof(double));

    for  (int j=0;j<3;j++)
                     f>>(Normals[i][j]);
                  f>>buf;
```

```
    }

      f.clear();

      while (strstr(buf, "IndexedFaceSet")==0&&!f.eof())
                    f.getline(buf,sizeof(buf));

    while (strstr(buf, "coordIndex [")==0&&!f.eof())
                    f.getline(buf,sizeof(buf));

    i=0;
    n=1;
      while(f.rdstate()==0)
      {

                    Triangles[i]=(int*)malloc(3*sizeof(int)) ;;
                    char c;
       for  (int j=0;j<3;j++)
                    {
                      f>>(Triangles[i][j]);
                      f>>c;
                    }

                    f>>buf;
                    i++;
                    if (i+1>n)
                    {
            n+=1000;
                      Triangles=(int**)realloc(Triangles,n*sizeof(int*));
       }
    }

    f.clear();
    int fn=i-1;
      Triangles=(int**)realloc(Triangles,fn*sizeof(int*));
      f.close();
      return;
  }
```

curvature.cpp

```
#include <iostream.h>
#include <istream.h>
#include <fstream.h>
#include <string.h>
#include <stdlib.h>
#include <malloc.h>
#include<math.h>
#include "matrix_double.cpp"
#include "matrix_int.cpp"

  void attributes(double const*const*Vertices,const int vn,int const*const* Triangles,const int
fn,int **&neighbours_vv,int **&neighbours_tt)
```

```
{
	int k=0;
	int* neighbours_number,**neighbours_vt;
	neighbours_number=rzeros(neighbours_number,vn);
	neighbours_tt=rzeros(neighbours_tt,fn,3);
	neighbours_vt=new int*[vn];
	neighbours_vv=new int*[vn];

	for (int i=0;i<fn;i++)
		for (int j=0;j<3;j++)
			neighbours_number[Triangles[i][j]]++;


	for (int i=0;i<vn;i++)
	{
		neighbours_vt[i]= new int[neighbours_number[i]+1];
		neighbours_vt[i][0]=0;
	}
	delete[] neighbours_number;
	for (int i=0;i<fn;i++)
		for (int j=0;j<3;j++)
		 {
			neighbours_vt[Triangles[i][j]][0]++;

			neighbours_vt[Triangles[i][j]][neighbours_vt[Triangles[i][j]][0]]=i;
		 }

	for (int i=0;i<vn;i++)
	{
		neighbours_vv[i]= new int[neighbours_vt[i][0]+1];
		neighbours_vv[i][0]=neighbours_vt[i][0];
	}
	for (int i=0;i<vn;i++)
	{
		int m=0;
		int s=0;
		int e=0;
		for (int j=1;j<neighbours_vt[i][0];j++)
		{
			m=j;
			s=0;
			while ((s!=2)&&(m!=neighbours_vt[i][0]))
			{
	s=0;
	m++;

	for (int r=0;r<3;r++)
	   for (int l=0;l<3;l++)
	      if (Triangles[neighbours_vt[i][j]][r]==Triangles[neighbours_vt[i][m]][l])
	      {
				s++;
		 if(Triangles[neighbours_vt[i][m]][l]!=i)
```

```
                    e=Triangles[neighbours_vt[i][m]][l];
                                        }
                    }
              k=neighbours_vt[i][m];
                    neighbours_vt[i][m]=neighbours_vt[i][j+1];
             neighbours_vt[i][j+1]=k;
             neighbours_vv[i][j]=e;
            }

            for (int k=0;k<3;k++)
                for (int l=0;l<3;l++)
                    if
(Triangles[neighbours_vt[i][neighbours_vt[i][0]]][k]==Triangles[neighbours_vt[i][1]][l]&&T
riangles[neighbours_vt[i][1]][l]!=i)
                    e=Triangles[neighbours_vt[i][1]][l];

          neighbours_vv[i][neighbours_vt[i][0]]=e;
        }

            int *temp=new int[fn];

            for (int i=0;i<fn;i++)
                    temp[i]=-1;

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

                    int s=0,k=0,l=0;
             for (int j=1;j<=neighbours_vt[i][0];j++)
             {
                k=neighbours_vt[i][j];
                if (j>1)
                  l=neighbours_vt[i][j-1];
                    else
                  l=neighbours_vt[i][neighbours_vt[i][0]];

                    s=0;
                for (int m=0;m<3;m++)
                    if (neighbours_tt[k][m]==l)
                    s++;

                    if (s==0)
                    {
                    temp[k]++;
                    temp[l]++;
                    neighbours_tt[k][temp[k]]=l;
                    neighbours_tt[l][temp[l]]=k;
                            }
                    }
            }
            destroy(neighbours_vt,vn);
            delete[] temp;
```

```
        return;
    }


    void curvature_cubic(double **H,double const*const*Vertices,const int vn,double
const*const*Normals,int const*const*neighbours_vv)
    {
        double **E,**temp_m;
        double *temp_v,*r1,*r2,*r3,*ans;

          E=rzeros(E,3,3);
          temp_m=rzeros(temp_m,3,3);
          temp_v=rzeros(temp_v,3);
          r1=rzeros(r1,3);
          r2=rzeros(r2,3);
          r3=rzeros(r3,3);
          ans=rzeros(ans,7);

          int max=0;

          for (int i=0;i<vn;i++)
                  if (neighbours_vv[i][0]>max)
                          max=neighbours_vv[i][0];

        double**neighbours_v,**neighbours_n,**R,*z;

        neighbours_v=rzeros(neighbours_v,max,3);
        neighbours_n=rzeros(neighbours_n,max,3);
        R=rzeros(R,3*max,7);
        z=rzeros(z,3*max);

        for (int i=0;i<3;i++)
                        for (int j=0;j<3;j++)
                            if (i!=j)
                                        E[i][j]=0;
                            else
                              E[i][j]=1;

                for (int i=0;i<vn;i++)
         {
                multiply(temp_m,Normals[i],3,Normals[i],3);
                        minus(temp_m,3,3,E,3,3);
                        multiply(temp_m,3,3,-1);
                        temp_v[0]=1;
                        temp_v[1]=0;
                        temp_v[2]=0;
                multiply(r1,temp_v,3,temp_m,3,3);
                multiply(r1,3,1/norma(r1,3));
                stroka(r3,Normals,vn,3,i);

                r2[0]=r3[1]*r1[2]-r3[2]*r1[1];
```

```
        r2[1]=-(r3[0]*r1[2]-r3[2]*r1[0]);
        r2[2]=r3[0]*r1[1]-r3[1]*r1[0];;

   put_stolbec(temp_m,3,3,r1,3,0);
   put_stolbec(temp_m,3,3,r2,3,1);
   put_stolbec(temp_m,3,3,r3,3,2);

        for (int j=1;j<=neighbours_vv[i][0];j++)
           {
          minus(temp_v,Vertices[neighbours_vv[i][j]],3,Vertices[i],3);
          multiply(neighbours_v[j-1],temp_v,3,temp_m,3,3);
          multiply(neighbours_n[j-1],Normals[neighbours_vv[i][j]],3,temp_m,3,3);
           }

for(int j=0;j<neighbours_vv[i][0];j++)
{
   R[j][0]=pow(neighbours_v[j][0],2)/2;
   R[j][1]=neighbours_v[j][0]*neighbours_v[j][1];
   R[j][2]=pow(neighbours_v[j][1],2)/2;
                R[j][3]=pow(neighbours_v[j][0],3);
   R[j][4]=pow(neighbours_v[j][0],2)*neighbours_v[j][1];
   R[j][5]=neighbours_v[j][1]*pow(neighbours_v[j][1],2);
   R[j][6]=pow(neighbours_v[j][1],3);

   R[neighbours_vv[i][0]+j][0]=neighbours_v[j][0];
   R[neighbours_vv[i][0]+j][1]=neighbours_v[j][1];
   R[neighbours_vv[i][0]+j][2]=0;
   R[neighbours_vv[i][0]+j][3]=3*pow(neighbours_v[j][0],2);
   R[neighbours_vv[i][0]+j][4]=2*neighbours_v[j][0]*neighbours_v[j][1];
   R[neighbours_vv[i][0]+j][5]=pow(neighbours_v[j][1],2);
   R[neighbours_vv[i][0]+j][6]=0;

   R[j+2*neighbours_vv[i][0]][0]=0;
   R[j+2*neighbours_vv[i][0]][1]=neighbours_v[j][0];
   R[j+2*neighbours_vv[i][0]][2]=neighbours_v[j][1];
   R[j+2*neighbours_vv[i][0]][3]=0;
   R[j+2*neighbours_vv[i][0]][4]=pow(neighbours_v[j][0],2);
   R[j+2*neighbours_vv[i][0]][5]=2*neighbours_v[j][0]*neighbours_v[j][1];
   R[j+2*neighbours_vv[i][0]][6]=3*neighbours_v[j][1];

   z[j]=neighbours_v[j][2];

   z[neighbours_vv[i][0]+j]=-neighbours_n[j][0]/neighbours_n[j][2];
   z[neighbours_vv[i][0]*2+j]=-neighbours_n[j][1]/neighbours_n[j][2];
}

least_squares(ans,R,3*neighbours_vv[i][0],7,z,3*neighbours_vv[i][0]);
//print(ans,3);
H[i][0]=4*ans[0]*ans[2]-ans[1]*ans[1];
 H[i][1]=ans[0]+ans[2];
 }
```

```
    for (int j=0;j<max;j++)
    {
                delete[] neighbours_v[j];
                delete[] neighbours_n[j];
    }
    for (int j=0;j<3*max;j++)
         delete[] R[j];

    delete[] neighbours_v;
    delete[] neighbours_n;
    delete[] R;

    for (int i=0;i<3;i++)
         {
                        delete[] E[i];
                        delete[] temp_m[i];
       }

    delete[] ans;
    delete[] E;
    delete[] temp_m;
    delete[] temp_v;
    delete[] r1;
    delete[] r2;
    delete[] r3;
    delete[] z;
    return;
  }

   void curvature_parabolloid(double **H,double const*const*Vertices,const int vn,double
const*const*Normals,int const*const*neighbours_vv)
   {
        double **E=new double*[3],**temp_m=new double*[3];
        double *temp_v=new double[3],*r1=new double[3],*r2=new double[3],*r3=new
double[3];
          int max=0;

          for (int i=0;i<vn;i++)
                if (neighbours_vv[i][0]>max)
                     max=neighbours_vv[i][0];

        double**neighbours=new double*[max],**R=new double*[max],*z=new double[max];

          for (int j=0;j<max;j++)
          {
                neighbours[j]=new double[3];
                R[j]=new double[3];
          }


          for (int i=0;i<3;i++)
          {
```

```
                    E[i]=new double[3];
                    temp_m[i]=new double[3];
    }

    for (int i=0;i<3;i++)
                    for (int j=0;j<3;j++)
                        if (i!=j)
                                E[i][j]=0;
                        else
                          E[i][j]=1;

            for (int i=0;i<vn;i++)
     {
            multiply(temp_m,Normals[i],3,Normals[i],3);
                    minus(temp_m,3,3,E,3,3);
                    multiply(temp_m,3,3,-1);
                    temp_v[0]=1;
                    temp_v[1]=0;
                    temp_v[2]=0;
            multiply(r1,temp_v,3,temp_m,3,3);
            multiply(r1,3,1/norma(r1,3));
            stroka(r3,Normals,vn,3,i);

            r2[0]=r3[1]*r1[2]-r3[2]*r1[1];
            r2[1]=-(r3[0]*r1[2]-r3[2]*r1[0]);
            r2[2]=r3[0]*r1[1]-r3[1]*r1[0];;

       put_stolbec(temp_m,3,3,r1,3,0);
       put_stolbec(temp_m,3,3,r2,3,1);
       put_stolbec(temp_m,3,3,r3,3,2);

            for (int j=1;j<=neighbours_vv[i][0];j++)
               {
              minus(temp_v,Vertices[neighbours_vv[i][j]],3,Vertices[i],3);
              multiply(neighbours[j-1],temp_v,3,temp_m,3,3);
               }

     for(int j=0;j<neighbours_vv[i][0];j++)
     {
        R[j][0]=neighbours[j][0]*neighbours[j][0];
        R[j][1]=neighbours[j][0]*neighbours[j][1];
        R[j][2]=neighbours[j][1]*neighbours[j][1];
        z[j]=neighbours[j][2];
     }

     least_squares(r2,R,neighbours_vv[i][0],3,z,neighbours_vv[i][0]);

     H[i][0]=4*r2[0]*r2[2]-r2[1]*r2[1];
      H[i][1]=r2[0]+r2[2];
      }

for (int j=0;j<max;j++)
```

```
    {
            delete[] neighbours[j];
            delete[] R[j];
    }

    delete[] neighbours;
    delete[] R;

    for (int i=0;i<3;i++)
        {
                delete[] E[i];
                delete[] temp_m[i];
       }

    delete[] E;
    delete[] temp_m;
    delete[] temp_v;
    delete[] r1;
    delete[] r2;
    delete[] r3;
    delete[] z;
    return;
  }
```

segmentation.cpp

```
  #include <iostream.h>
  #include <istream.h>
  #include <fstream.h>
  #include <string.h>
  #include <stdlib.h>
  #include <malloc.h>
  #include <math.h>
  //#include "matrix_double.cpp"
  //#include "matrix_int.cpp"
  void grow_region(int *Segments,int *number,const int i,const int k, int const*const*
neighbours_tt,int const* list)
  {
      Segments[i]=k;
      (*number)++;
      for(int j=0;j<3;j++)
      {
            int m=neighbours_tt[i][j];
            if (list[m]==1&&Segments[m]==0)
                  grow_region(Segments,number,m,k,neighbours_tt,list);
     }
     return;
  }


  int segmentation_c(int *Segments,int const*const* neighbours_tt,int const*list,const int
fn)//выделяет связные сегменты вершин
  {
```

```
    int k=0,*sn;
    int *exc,l=0;
    sn=rzeros(sn,fn);
    for (int i=0;i<fn;i++)
            if (list[i]==1&&Segments[i]==0)
            {
              k++;
              grow_region(Segments,&sn[k-1],i,k,neighbours_tt,list);
     }


  exc=rzeros(exc,k+1);
  //удаление маленьких сегментов
  for (int i=1;i<=k;i++)
       if (sn[i-1]>5)
       {
              l++;
              exc[i]=l;
    }

  k=l;

  for (int i=0;i<fn;i++)
       Segments[i]=exc[Segments[i]];

    delete[] exc;
    delete[] sn;
    //sn=(int*)realloc(sn,k);
    return k;
}
//---------------------------------------------------------------------------
 //к-средних
int center(double const*const*Vertices,const int vn,int const *list,const int l)
{
    //ищем элемент с минимальной суммой расстояний до среднего
    //арифметического координат элементов кластера

    //c-индекс найденного центра
    //Vertice_Matrix - список координат всех вершин триангуляции
    //list - массив с номерами вершин, входящих в кластер
    double *d,*x,m=0;
    int v_min=0;
    d=rzeros(d,l);
    //ищем элемент с минимальной суммой расстояний до среднего
    //арифметического координат элементов кластеров
    x=rzeros(x,3);
    for (int j=0;j<l;j++)
        for (int k=0;k<3;k++)
           x[k]+=Vertices[list[j]][k]/l;

    for (int j=0;j<l;j++)
    {
```

```
            for (int k=0;k<3;k++)
                d[j]+=pow(Vertices[list[j]][k]-x[k],2);
            d[j]=sqrt(d[j]);
        }
        for (int j=0;j<l;j++)
                if (d[j]>m)
                {
                  m=d[j];
                  v_min=j;
        }
       delete[] x;
       delete[] d;
        return list[v_min];
   }

    void k_means(int* Segments,const int l,double const*const*Vertices,double
const*const*Distances,const int vn,int const*list,const int vn_1)
    {

        int s=0;
        int *seed,v_min=0;//центры кластера
        int **segments,*sn;
        int r=0;
        int count=0,s_min;
        double d_min;

        s=(int)floor(vn_1/l);
        //cout<<"l="<<l<<"\n";
        //среднее количество элементов в кластере
        sn=rzeros(sn,l);
        segments=rzeros(segments,l,vn_1);
        seed=rzeros(seed,l);

        for (int k=0;k<l;k++)
        {
           int j=0;
           while(j<s)
           {
              if(list[r]>0)
                 j++;
              r++;
           }
           seed[k]=r-1;
        }

        //количество изменений центров кластеров во время итерации
        s=1;
        //список кластеров с вершинами им принадлежащими
        //в іой сроке на 1 месте количество элементов в кластере, дальше список
        //вершин, принадлежащих кластеру

        while(s>0&&count<1000)
```

```
{

 zeros(sn,l);
   count++;
   s=0;
   //относим iую вершину к тому кластеру, к центру которого она ближе



   zeros(sn,l);
   //zeros(segments,l,vn_1);
   for (int i=0;i<vn;i++)
   {
      if(list[i]>0)
      {
         d_min=10000;
         s_min=0;
         for (int j=0;j<l;j++)
            if(Distances[i][seed[j]]<d_min)
            {
               d_min=Distances[i][seed[j]];
               s_min=j;
                                 }

         sn[s_min]++;
         segments[s_min][sn[s_min]-1]=i;
    }
 }
   //пересчитываем центры вершин
   for (int i=0;i<l;i++)
   {
      v_min=center(Vertices,vn,segments[i],sn[i]);
        //смотрим, изменился ли центр с предыдущей итерации
      if(v_min!=seed[i])
      {
            s++;
            seed[i]=v_min;
      }
   }
}

 for (int i=0;i<l;i++)
    for (int j=0;j<sn[i];j++)
       Segments[segments[i][j]]=i+1;

 destroy(segments,l);
 delete[] sn;
 delete[] seed;
 return;
}
```

```
int k_means_1(int* Segments, int l,double const*const*Vertices,double
const*const*Distances,const int vn)
{
//k_means_1 находит компоненты связности среди вершин на
//основе Distance_Matrix

//Segment_Matrix - массив, на іом месте номер кластера, к которому
//принадлежит іая вершина
//l-количество кластеров
//Distance_Matrix-содержит кратчаюшие расстояния между вершинами, для
//которых list больше 0, если вершины i,j в разных компонентах связанности,
//то Distance_Matrix(i,j)=0

    int count=0;
    int *list_1,*list;
    int k=0;
    int m=0;
    int *S;

    S=rzeros(S,vn);
    list_1=rzeros(list_1,vn);
    list=rones(list,vn);

    for (int i=0;i<vn;i++)
    //для каждой компоненты связности запускаем к-средних
       if(list[i]>0)
       {
          zeros(list_1,vn);
          zeros(S,vn);
          list_1[i]=1;
          list[i]=0;
          //count - общее количество кластеров на предыдущем шаге
          //m - кол-во кластеров на предыдущем шаге
          count+=m;
          k=1;
          //%list_1(j)=1, если jая вершина находится в одной компоненте
          //%связности с іой
          //%те вершины, которые мы кластеризуем, убираем из списка list
          for (int j=0;j<vn;j++)
             if(Distances[i][j]>0)
             {
                list_1[j]=1;
                list[j]=0;
                k++;
             }
               //cout<<"k="<<k<<"\n";
          m=(int)floor(l*k/vn);

          //%m-число кластеров на этом шаге
          if(m!=0)
                    {
             k_means(S,m,Vertices,Distances,vn,list_1,k);
```

```
            //%пересчитываем номера кластеров
            for (int j=0;j<vn;j++)
                if(S[j]>0)
                    S[j]+=count;

            plus(Segments,vn,S,vn);
          }
        }
        //cout<<"----------\n";
    l=count+m;

    delete[] S;
    delete[] list_1;
    delete[] list;
    return l;
}


//-------------------------------------------------
void min_heapify(double**Q,int*num,const int i,const int l)
{
//%min_heapify - опускает элемент с іым номером вниз по пирамиде Q,если он
//%нарушает упорядоченность пирамиды
//%соответствущим образом изменяет пит

//%Q - массив, на 1 месте расстояние до іой вершины, на 2 - номер
//%вершины
//%num(i) - под каким номером іая вершина находится в Q
//%i - номер элемента, который опускать
//%Очередь Q организована в виде пирамиды, l - длина очереди
  int m=-1;

  if((2*i<l)&&(Q[2*i][0]<Q[i][0]))
    m=2*i;
  else
    m=i;

  if(2*i+1<l&&Q[2*i+1][0]<Q[m][0])
    m=2*i+1;

  if(i!=m)
  {
    num[(int)Q[m][1]]=i;
    num[(int)Q[i][1]]=m;
    Q[l][0]=Q[m][0];
    Q[l][1]=Q[m][1];
    Q[m][0]=Q[i][0];
    Q[m][1]=Q[i][1];
    Q[i][0]=Q[l][0];
    Q[i][1]=Q[l][1];
    min_heapify(Q,num,m,l);
  }
```

```
  return;
}

void extract_min(double*m,double**Q,int *num,int *l)
{
//%extract_min - достать минимальный элемент из очереди, соответствующим
//%образом изменить num и Q

//%m - найденное минимальное значение
//%Q - массив, на 1 месте расстояние до iой вершины, на 2 - номер
//%вершины
//%num(i) - под каким номером iая вершина находится в Q
//%Очередь Q организована в виде пирамиды, l - длина очереди
      m[0]=Q[0][0];
      m[1]=Q[0][1];
      num[(int)Q[0][1]]=-1;
      num[(int)Q[*(l)-1][1]]=0;
      Q[0][0]=Q[*(l)-1][0];
      Q[0][1]=Q[*(l)-1][1];
      (*l)--;
      min_heapify(Q,num,0,*l);
      return;
}

void decrease_key(double**Q,int*num, int i,const double k,const int l)
{
//%decrease_key - присваивает Q(i,1) значение k, если k меньше,
//%соответствующим образом изменяет num и Q
//
//%Q - массив, на 1 месте расстояние до iой вершины, на 2 - номер
//%вершины
//%num(i) - под каким номером iая вершина находится в Q
//%Очередь Q организована в виде пирамиды, len - длина очереди
//%i - номер элемента, ключ которого надо изменить
//%k - новое значение ключа

      if(Q[i][0]>k)
      {
        Q[i][0]=k;
        int m=(int)floor(i/2);
        while(m>=0&&Q[i][0]<Q[m][0])
        {
           num[(int)Q[i][1]]=m;
           num[(int)Q[m][1]]=i;
           Q[l][0]=Q[m][0];
           Q[l][1]=Q[m][1];
           Q[m][0]=Q[i][0];
           Q[m][1]=Q[i][1];
           Q[i][0]=Q[l][0];
           Q[i][1]=Q[l][1];
           i=m;
           m=(int)floor(i/2);
```

```
        }
    }

    return;
}


void distance_piram(double**Distances,double const*const*Vertices,int
const*const*neighbour_matrix_vv,const int vn)
{
//%distance - вычисляет матрицу расстояний

//%Distances-содержит кратчаюшие расстояния между вершинами, если вершины i,j в
разных компонентах связанности,
//%то Distances(i,j)=0
//%Vertices - список координат вершин
//%neighbour_matrix_vv-iая строка содержит на
//%1ом месте количество элементов в строке, дальше список вершин, соседних к
//%iой, идущие друг за другом
//%list-список вершин, для которых рассчитываем

    int k=0;
    double **adjacency_matrix;
    int *num,len;
    double **Q,*d_min=new double[2];
    int d,l,r;

    adjacency_matrix=rzeros( adjacency_matrix,vn,vn);
    num=rzeros(num,vn);
    //
    //zeros(Distances,vn,vn);
    //%вычисляем матрицу смежности

    for (int i=0;i<vn;i++)
        for (int j=1;j<=neighbour_matrix_vv[i][0];j++)
        {
            k=neighbour_matrix_vv[i][j];

            if(adjacency_matrix[i][k]<=0)
            {
                                    adjacency_matrix[i][k]=sqrt(pow(Vertices[i][0]-
Vertices[k][0],2)+pow(Vertices[i][1]-Vertices[k][1],2)+pow(Vertices[i][2]-Vertices[k][2],2));
                    adjacency_matrix[k][i]=adjacency_matrix[i][k];
            }
        }

    //%вычисляем кратчайший путь
    //%Q - очередь вершин

    Q=rzeros(Q,vn+1,2);

    for (int i=0;i<vn;i++)
```

```
{
    //%если вершина в списке, формируем очередь из вершин,для которых
    //%list>0, ставим i  на 1 место
    //%Q - массив, на 1 месте расстояние до iой вершины, на 2 - номер
    //%вершины
    //%num(i) - под каким номером iая вершина находится в Q
    //%Очередь Q организована в виде пирамиды, len - длина очереди

    for (int j=0;j<vn;j++)
   {
          Q[j][0]=10000;
          Q[j][1]=j;
          num[j]=j;
               }
               Q[i][1]=0;
  num[0]=i;
  Q[0][1]=i;
  num[i]=0;
  Q[0][0]=0;

    len=vn;
    while(len>0)
               {
    //%находим вершину с минимальным расстоянием до iой вершины из
    //%очереди
      extract_min(d_min,Q,num,&len);
                    d=(int)d_min[1] ;

    //%пересчитываем расстояния для соседних вершин
      for (int j=1;j<=neighbour_matrix_vv[d][0];j++)
      {
        l=neighbour_matrix_vv[d][j];
        //cout<<"len="<<len<<" "<<(int)floor(len/2)<<"\n";
        if(num[l]!=-1)
          decrease_key(Q,num,num[l],d_min[0]+adjacency_matrix[l][d],len);


      }

      if(d_min[0]==10000)
        d_min[0]=0;

      Distances[i][d]=d_min[0];
      Distances[d][i]=d_min[0];
    }
  }

      delete[] d_min;
      delete[] num;

      destroy(Q,vn);
      destroy(adjacency_matrix,vn);
```

```
            return;

    }
    //-----------------------------------------

    void mapping(int*map,int*map_1,double**Vertices_1,const int vn_1,double
const*const*Vertices, int const*list,const int vn)
    {
    //%перенумеровывает несегментированные вершины и записывает соответствие в
    //%map
        int count=-1;
        for (int i=0;i<vn;i++)
            if(list[i]>0)
            {

                count++;
                map[count]=i;
                map_1[i]=count;

                for (int j=0;j<3;j++)
                {
                    Vertices_1[count][j]=Vertices[i][j];
                }
            }

        return;
    }

    int ** neighbours(int const*const*neighbours_vv,const int vn,int const*map_1,const int vn_1)
    {
        int*numbers,k;
        numbers=rzeros(numbers,vn_1);

        for (int i=0;i<vn;i++)
            if(map_1[i]>=0)
                for (int j=1;j<=neighbours_vv[i][0];j++)
                {
                    k=neighbours_vv[i][j];
                    if(map_1[k]>=0)
                        numbers[map_1[i]]++;
                }

        int** neighbours_vv_1=new int*[vn_1];

        for (int i=0;i<vn_1;i++)
        {
            neighbours_vv_1[i]=new int[numbers[i]+1];
            neighbours_vv_1[i][0]=0;
        }

        delete[] numbers;
```

```
        for (int i=0;i<vn;i++)
            if(map_1[i]>=0)
              for (int j=1;j<=neighbours_vv[i][0];j++)
              {
                 k=neighbours_vv[i][j];
                 if(map_1[k]>=0)
                 {
                    neighbours_vv_1[map_1[i]][0]++;
                    neighbours_vv_1[map_1[i]][neighbours_vv_1[map_1[i]][0]]=map_1[k];
                 }
              }

       return neighbours_vv_1;
    }

   int segment_cut(int*Segments_v,double const*const*Vertices,const int vn,int
const*const*neighbours_vv,const int max_size,const int i,int k1)
   {
                        //cout<<"dghdftgh\n";
                        double**Distances_1,**Vertices_1;
            int vn_1=0,*Segments,*Segments_1,*S,*map,*map_1,**neighbours_vv_1;
                        S=rzeros(S,vn);
                        Segments_1=rzeros(Segments_1,vn);
            for (int j=0;j<vn;j++)
               if(Segments_v[j]==i)
               {
                  S[j]=1;
                  vn_1++;
                           }
            Distances_1=rzeros(Distances_1,vn_1,vn_1);
                        Segments=rzeros(Segments,vn_1);
                        map=rzeros(map,vn_1);
                Vertices_1=rzeros(Vertices_1,vn_1,3);
                map_1=rzeros(map_1,vn);
                minus(map_1,vn,-1);
            mapping(map,map_1,Vertices_1,vn_1,Vertices, S,vn);
            neighbours_vv_1=neighbours(neighbours_vv,vn,map_1,vn_1);
            //%перенумеровывает несегментированные вершины и записывает
соответствие в
                        distance_piram(Distances_1,Vertices_1,neighbours_vv_1,vn_1);
            int k=(int)(k1/max_size)+1;
            k=k_means_1(Segments, k,Vertices_1,Distances_1,vn_1);
            //%кластеров может получиться меньше, чем к, потому что слишком
маленькие компоненты связности не сегментируются

            for (int j=0;j<vn_1;j++)
               Segments_1[map[j]]=Segments[j];

            for (int j=0;j<vn;j++)
               if(Segments_1[j]>1)
                 Segments_v[j]=k1+Segments_1[j]-1;
```

```
            k1+=k-1;
            delete[]Segments;
                        delete[]Segments_1;
                        delete[]S;
                        delete[]map;
                        delete[]map_1;

                        for (int j=0;j<vn_1;j++)
                        {
                                delete[] neighbours_vv_1[j];
                                delete[]Distances_1[j];
              delete[]Vertices_1[j];
                        }

                        delete[] neighbours_vv_1;
                    delete[]Distances_1;
            delete[]Vertices_1;
            return k1;
    }
    //-----------------------------
    //segmentacia
    void segmentation(int*Segments_v,int *numbers,double const*const*Vertices,const int vn,int
const*const* Triangles,const int fn,int const*const*neighbours_vv,int
const*const*neighbours_tt,int*colour_matrix,const int max_size)
    {
          int k1,k2,k,*list_t,*sv,*Segments_t,*Segments_1;
          double**Distances;
          double**Vertices_1;
          int vn_1=0,*list_v,*map,*map_1,**neighbours_vv_1;
          Segments_t=rzeros(Segments_t,fn);
          list_t=rzeros(list_t,fn);
          list_v=rzeros(list_v,vn);

          for (int i=0;i<vn;i++)
          {
                        int s=0;
                        for (int j=1;j<=neighbours_vv[i][0];j++)
                        {
                                if (colour_matrix[i]!=colour_matrix[neighbours_vv[i][j]])
                                  s++;
              }
                        if (s==neighbours_vv[i][0])
                                colour_matrix[i]=colour_matrix[neighbours_vv[i][1]];
          }
          //определение типа треугольника
          for (int i=0;i<fn;i++)
                    if
(colour_matrix[Triangles[i][0]]==colour_matrix[Triangles[i][1]]&&colour_matrix[Triangles[
i][0]]==colour_matrix[Triangles[i][2]])
                        list_t[i]=colour_matrix[Triangles[i][0]];

          //сегментация
```

```
k1=segmentation_c(Segments_t,neighbours_tt,list_t,fn);

sv=rzeros(sv,k1);
//print(Segments_t,fn);
for (int i=0;i<fn;i++)
        if (Segments_t[i]>0)
          for (int j=0;j<3;j++)
                        if (Segments_v[Triangles[i][j]]==0)
                        {
                           Segments_v[Triangles[i][j]]=Segments_t[i];
                                sv[Segments_t[i]-1]++;
                }

for (int i=0;i<k1;i++)
        if (sv[i]>max_size)
           k1=segment_cut(Segments_v,Vertices,vn,neighbours_vv,max_size,i+1,k1);

delete[] sv;
sv=0;
multiply(list_t,fn,-1);
zeros(Segments_t,fn);
k2=segmentation_c(Segments_t,neighbours_tt,list_t,fn);
sv=rzeros(sv,k2);
//cout<<k2<<"\n";
for (int i=0;i<fn;i++)
        if (Segments_t[i]>0)
          for (int j=0;j<3;j++)
                        if (Segments_v[Triangles[i][j]]==0)
                        {
                           Segments_v[Triangles[i][j]]=Segments_t[i]+k1;
                           sv[Segments_t[i]-1]++;
                }

for (int i=0;i<k2;i++)
        if (sv[i]>max_size)
        {

k2=segment_cut(Segments_v,Vertices,vn,neighbours_vv,max_size,i+1+k1,k1+k2);
                k2-=k1;
      }

k=(int)((k1+k2)/2);



for (int i=0;i<vn;i++)
        if (Segments_v[i]==0)
        {
                list_v[i]=1;
                vn_1++;
        }
```

```
    Distances=rzeros(Distances,vn_1,vn_1);
    Segments_1=rzeros(Segments_1,vn_1);
    map=rzeros(map,vn_1);
    Vertices_1=rzeros(Vertices_1,vn_1,3);
    map_1=rzeros(map_1,vn);

    mapping(map,map_1,Vertices_1,vn_1,Vertices, list_v,vn);
    neighbours_vv_1=neighbours(neighbours_vv,vn,map_1,vn_1);

    //%перенумеровывает несегментированные вершины и записывает соответствие в
    distance_piram(Distances,Vertices_1,neighbours_vv_1,vn_1);
    //print_f(Distances,vn_1,vn_1,"Distances.txt");
    k=k_means_1(Segments_1,k,Vertices_1,Distances,vn_1);
    cout<<k1<<"\n";
    cout<<k2<<"\n";
    cout<<k<<"\n";
    for (int i=0;i<vn_1;i++)
       if (Segments_1[i]>0)
         Segments_v[map[i]]=Segments_1[i]+k1+k2;

       delete[]Segments_t;
       delete[]Segments_1;
       delete[]list_v;
       delete[]map;
       delete[]map_1;
       delete[] list_t;
       delete[] sv;

       destroy(neighbours_vv_1,vn_1);
       destroy(Distances,vn_1);
       destroy(Vertices_1,vn_1);

       numbers[0]=k1;
       numbers[1]=k1+k2;
       numbers[2]=k1+k2+k;
       return;
   }
```

marking.cpp

```
   #include <iostream.h>
   #include <istream.h>
   #include <fstream.h>
   #include <string.h>
   #include <stdlib.h>
   #include <malloc.h>
   #include <math.h>
   void marking(int*labels, int*centers,char*filename,const int* Segments,const
int*numbers,double const*const*Vertices,const int vn)
   {
        double*p,**coord,*charges,*temp;
      int l,*num,**S,k=numbers[2];
      ifstream f;
```

```
cout<<k<<"\n";
p=rzeros(p,k);
temp=rzeros(temp,3);
num=rzeros(num,k);
S=new int*[k];

  for (int i=0;i<vn;i++)
            if (Segments[i]>0)
               num[Segments[i]-1]++;

 for (int i=0;i<k;i++)
           S[i]=rzeros(S[i],num[i]);

 zeros(num,k);

  for (int i=0;i<vn;i++)
            if (Segments[i]>0)
            {
               num[Segments[i]-1]++;
               S[Segments[i]-1][num[Segments[i]-1]-1]=i;
     }

 for (int i=0;i<k;i++)
          centers[i]=center(Vertices,vn,S[i],num[i]);

  delete[] num;
  destroy(S,k);

  f.open(filename);

  if(f.fail())
   {
            cerr<<"invalid filename";
            return;
}
f>>l;

 coord=rzeros(coord,l,3);
 charges=rzeros(charges,l);

for (int i=0;i<l;i++)
{
            for (int j=0;j<3;j++)
                     f>>coord[i][j];

            f>>charges[i];
   }

   f.close();

   for (int i=0;i<k;i++)
            for (int j=0;j<l;j++)
```

```
                {
                        //cout<<i-k<<"\n";
                    minus(temp,coord[j],3,Vertices[centers[i]],3);
                    p[i]+=charges[j]/norma(temp,3);
                    zeros(temp,3);
                }

        //cout<<"1\n";
          for (int i=0;i<k;i++)
          {


                int j=0;
                while(i>numbers[j])
                        j++;

               labels[i]=2*j+1;
               if (p[i]>0)
                        labels[i]+=1;

          }

          delete[] p;
          destroy(coord,l);
          delete[] charges;
          delete[]temp;
          return;
    }
```

matrix_int.cpp

```
    #include <iostream.h>
    #include <istream.h>
    #include <fstream.h>
    #include <string.h>
    #include <math.h>
    //на первом месте - возвращаемое значение, дальше - сначала объект, за ним его
размерность
    //умножение
    void multiply(int** C,int const*const*A,const int m,const int n1, int const*const*B,const int
n2,const int k)
    {

                if (n1!=n2)
                        return ;

                int n=n1;
                for (int i=0;i<m;i++)
                        for (int j=0;j<k;j++)
                        {
                                        C[i][j]=0;
                                        for (int l=0;l<n;l++)
                                            C[i][j]+=A[i][l]*B[l][j];
```

```
                }

        return;
}

void multiply(int** C,const int*A, const int n1, const int *B, const int n2)
{
        if (n1!=n2)
                return ;

        int n=n1;
        for (int i=0;i<n;i++)
                for (int j=0;j<n;j++)
                C[i][j]=A[i]*B[j];

        return ;
}

void multiply(int* C,int const*const*A, const int m,const int n1, const int *B,const int n2)
{
        if (n1!=n2)
                return;

        int n=n1;

        for (int i=0;i<m;i++)
        {
    C[i]=0;
  for (int l=0;l<n;l++)
                                C[i]+=A[i][l]*B[l];
  }

        return;
}

void multiply(int* C,const int*B,const int n2, int const*const*A, const int n1,const int m)
{
        if (n1!=n2)
                return ;

        int n=n1;

        for (int i=0;i<m;i++)
        {
    C[i]=0;
  for (int l=0;l<n;l++)
                                C[i]+=A[l][i]*B[l];
  }

        return ;
}
```

```
void multiply(int* a,const int *v,const int n,const int c)
{
        for(int i=0;i<n;i++)
            a[i]=v[i]*c;

    return ;
}

void multiply(int** C,int const*const*A,const int m,const int n,const int c)
{
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=A[i][j]*c;

        return ;
}

void multiply(int*a,const int c,const int *v,const int n)
{
        for(int i=0;i<n;i++)
            a[i]=v[i]*c;

    return;
}

void multiply(int** C,const int c,int const*const*A,const int m,const int n)
{
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=A[i][j]*c;

        return ;
}
//*=
void multiply(int**A, int m,int n1,int const*const*B,int n2,int k)
{

        if (n1!=n2)
                return ;
        if (n1!=k)
                return ;
        int n=n1;
        int **C=new int*[m];
        for (int i=0;i<m;i++)
        {
                C[i]=new int[n];
                for (int j=0;j<n;j++)
                {
                                C[i][j]=0;
                                for (int l=0;l<n;l++)
                                    C[i][j]+=A[i][l]*B[l][j];
                 }
```

```
        }
            for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                    A[i][j]=C[i][j];

        for (int i=0;i<m;i++)
            delete[] C[i];

            delete C;
            return;
}



 void multiply(int *v,const int n,const int c)
{
            for(int i=0;i<n;i++)
                v[i]=v[i]*c;

      return ;
}

void multiply(int **A,const int m,const int n,const int c)
{
            for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                    A[i][j]=A[i][j]*c;

            return ;
}


//сложение
void plus(int** C,int const*const*A,const int m1,const int n1, int const*const*B,const int
m2,const int n2)
{
            if (n1!=n2||m1!=m2)
                return ;

            int n=n1;
            int m=m1;

            for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                    C[i][j]=A[i][j]+B[i][j];

            return ;
}

void plus(int* C,const int*A, const int n1, const int *B, const int n2)
{
            if (n1!=n2)
```

```
                return;

        int n=n1;

        for (int i=0;i<n;i++)
                C[i]=A[i]+B[i];

        return;
}

void plus(int* a,const int *v,const int n,const int c)
{
        for(int i=0;i<n;i++)
            a[i]=v[i]+c;

    return ;
}

void plus(int** C,int const*const*A,const int m,const int n,const int c)
{
        for (int i=0;i<n;i++)
                C[i]=new int[n];

        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=A[i][j]+c;

        return ;
}

void plusc(int* a,const int c,const int *v,const int n)
{
        for(int i=0;i<n;i++)
            a[i]=v[i]+c;

    return ;
}

void plus(int** C,const int c,int const*const*A,const int m, const int n)
{
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=A[i][j]+c;

        return;
}
//+=

void plus( int**A,const int m1,const int n1, int const*const*B,const int m2,const int n2)
{
        if (n1!=n2||m1!=m2)
                return ;
```

```
		int n=n1;
		int m=m1;

		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				A[i][j]+=B[i][j];

		return;
	}

	void  plus( int*A,const int n1, const int *B,const int n2)
	{
		if (n1!=n2)
			return ;

		int n=n1;

		for (int i=0;i<n;i++)
			A[i]+=B[i];

		return;
	}

	void  plus(int *v,const int n,const int c)
	{
		for(int i=0;i<n;i++)
			v[i]=v[i]+c;
	return;
	}

	void plus(int **A,const int m,const int n,const int c)
	{
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				A[i][j]+=c;


		return;
	}


	//минус
	void minus(int** C,int const*const*A,const int m1,const int n1, int const*const*B,const int
m2,const int n2)
	{
		if (n1!=n2||m1!=m2)
			return ;

		int n=n1;
		int m=m1;
```

```
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				C[i][j]=A[i][j]-B[i][j];

		return ;
}

void minus(int* C,const int*A, const int n1, const int *B, const int n2)
{
		if (n1!=n2)
			return;

		int n=n1;

		for (int i=0;i<n;i++)
			C[i]=A[i]-B[i];

		return;
}

void minus(int* a,const int *v,const int n,const int c)
{
		for(int i=0;i<n;i++)
		  a[i]=v[i]-c;

	return ;
}

void minus(int** C,int const*const*A,const int m,const int n,const int c)
{
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				C[i][j]=A[i][j]-c;

		return ;
}

void minusc(int* a,const int c,const int *v,const int n)
{
		for(int i=0;i<n;i++)
		  a[i]=-v[i]+c;

	return ;
}

void minus(int** C,const int c,int const*const*A,const int m, const int n)
{
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				C[i][j]=-A[i][j]+c;

		return;
```

```
}

//-=

void minus( int**A,const int m1,const int n1, int const*const*B,const int m2,const int n2)
{
		if (n1!=n2||m1!=m2)
			return ;

		int n=n1;
		int m=m1;

		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
					A[i][j]-=B[i][j];

		return;
}

void  minus( int*A,const int n1, const int *B,const int n2)
{

		if (n1!=n2)
			return ;

		int n=n1;

		for (int i=0;i<n;i++)
			A[i]-=B[i];

		return;
}

void  minus(int *v,const int n,const int c)
{
		for(int i=0;i<n;i++)
		   v[i]-=c;
	return;
}

void minus(int **A,const int m,const int n,const int c)
{
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				A[i][j]-=c;


		return;
}


//-------------------------------------------------------
```

```
void transpose_matrix(int **C,int const*const*A,const int m,const int n)
{
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				C[j][i]=A[i][j];

		return ;
}

int determinant(int const*const*A,const int m,const int n)
{
	if (m!=n||m!=2)
		return 0;

	return A[0][0]*A[1][1]-A[0][1]*A[1][0];
}

double norma(const int *v,const int n)
{
		double a=0;
		for(int i=0;i<n;i++)
		  a+=v[i]*v[i];
		return sqrt(a);
}

int scalar_product(const int *v1,const int n1,const int *v2,const int n2)
{
		if (n1!=n2)
			return 0;

		int n=n1;
		int C=0;
		for (int i=0;i<n;i++)
			C+=v1[i]*v2[i];

		return C;
}

void stroka(int*C,int const*const*A,const int m,const int n,const int i)
{
	for (int j=0;j<n;j++)
				C[j]=A[i][j];

	return ;
}

void stolbec(int* C,int const*const*A,const int m,const int n,const int i)
{
	for (int j=0;j<m;j++)
				C[j]=A[j][i];
```

```
    return;
}

void put_stroka(int**A,const int m,const int n1,const int *C,const int n2,const int i)
{
      if(n1!=n2)
        return;

    int n=n1;

      for (int j=0;j<n;j++)
                              A[i][j]=C[j];

    return ;
}

void put_stolbec(int**A,const int n1,const int m,const int *C,const int n2,const int i)
{
      if(n1!=n2)
        return;

    int n=n1;

      for (int j=0;j<n;j++)
                              A[j][i]=C[j];

    return ;
}
//
int** rzeros(int**A,const int m,const int n)
{
     A=new int*[m];
     for (int i=0;i<m;i++)
     {
            A[i]=new int[n];
            for (int j=0;j<n;j++)
                  A[i][j]=0;
   }
   return A;
}

int* rzeros(int*A,const int n)
{
     A=new int[n];
     for (int j=0;j<n;j++)
                  A[j]=0;

   return A;
}
//запись нулей в массив и вектор
void zeros(int**A,const int n,const int m)
{
```

```
      for (int i=0;i<m;i++)
      {

            for (int j=0;j<n;j++)
                  A[i][j]=0;
   }
   return;
}

void zeros(int*A,const int n)
{

      for (int j=0;j<n;j++)
                  A[j]=0;

   return;
}

//создание массива и вектора и запись в них единиц
int** rones(int**A,const int m,const int n)
{
      A=new int*[m];
      for (int i=0;i<m;i++)
      {
            A[i]=new int[n];
            for (int j=0;j<n;j++)
                  A[i][j]=1;
   }
   return A;
}

int* rones(int*A,const int n)
{
      A=new int[n];
      for (int j=0;j<n;j++)
                  A[j]=1;

   return A;
}

//запись единиц в массив и вектор
void ones(int**A,const int m,const int n)
{

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

            for (int j=0;j<n;j++)
                  A[i][j]=1;
   }
   return;
```

```
}

void ones(int*A,const int n)
{

	for (int j=0;j<n;j++)
				A[j]=1;

    return;
}
//печать

void print(int const*v,int n)
{
	for (int i=0;i<n;i++)
		cout<<v[i]<<"\n";

	return;
}

void print(int const*const*A,int m,int n)
{
	for (int i=0;i<m;i++)
	{
		for (int j=0;j<n;j++)
			cout<<A[i][j]<<" ";
		cout<<"\n";
    }
		return;
}

void print_f(int const*v,int n,const char*s)
{
	FILE *f;
	f=fopen(s,"w");
	fprintf(f,"%d\n",n);
	for (int i=0;i<n;i++)
		fprintf(f,"%d\n",v[i]);

	fclose(f);
		return;
}

void print_f(int const*const*A,int m,int n,const char*s)
{
	FILE *f;
	f=fopen(s,"w");
	fprintf(f,"%d\n",m);
	fprintf(f,"%d\n",n);
	for (int i=0;i<m;i++)
	{
		for (int j=0;j<n;j++)
```

```
                fprintf(f,"%d ",A[i][j]);
            fprintf(f,"\n");
        }
        fclose(f);
            return;
    }
    //удаление


    void destroy(int **A,int m)
    {
        for (int i=0;i<m;i++)
        {
            delete[] A[i];
            A[i]=0;
        }

        delete[] A;
        A=0;
            return;
    }

    void destroy1(int **A,int m)
    {
        for (int i=0;i<m;i++)
        {
            free(A[i]);
            A[i]=0;
        }

        free(A);
        A=0;
            return;
    }
```

matrix_double.cpp

```
    #include <iostream.h>
    #include <istream.h>
    #include <fstream.h>
    #include <string.h>
    #include <math.h>
    //на первом месте - возвращаемое значение, дальше - сначала объект, за ним его
размерность
    //умножение
    void multiply(double** C,double const*const*A,const int m,const int n1, double
const*const*B,const int n2,const int k)
    {

            if (n1!=n2)
                return ;

            int n=n1;
```

```
        for (int i=0;i<m;i++)
                for (int j=0;j<k;j++)
                {
                                C[i][j]=0;
                                for (int l=0;l<n;l++)
                                    C[i][j]+=A[i][l]*B[l][j];
                }

        return;
}

void multiply(double** C,const double*A, const int n1, const double *B, const int n2)
{
        if (n1!=n2)
                return ;

        int n=n1;
        for (int i=0;i<n;i++)
                for (int j=0;j<n;j++)
               C[i][j]=A[i]*B[j];

        return ;
}

void multiply(double* C,double const*const*A, const int m,const int n1, const double
*B,const int n2)
{
        if (n1!=n2)
                return;

        int n=n1;

        for (int i=0;i<m;i++)
        {
      C[i]=0;
    for (int l=0;l<n;l++)
                                C[i]+=A[i][l]*B[l];
    }

        return;
}

void multiply(double* C,const double*B,const int n2, double const*const*A, const int
n1,const int m)
{
        if (n1!=n2)
                return ;

        int n=n1;

        for (int i=0;i<m;i++)
        {
```

```
            C[i]=0;
          for (int l=0;l<n;l++)
                                        C[i]+=A[l][i]*B[l];
          }

                return ;
}

void multiply(double* a,const double *v,const int n,const double c)
{
               for(int i=0;i<n;i++)
                  a[i]=v[i]*c;

      return ;
}

void multiply(double** C,double const*const*A,const int m,const int n,const double c)
{
               for (int i=0;i<m;i++)
                        for (int j=0;j<n;j++)
                                 C[i][j]=A[i][j]*c;

               return ;
}

void multiply(double*a,const double c,const double *v,const int n)
{
               for(int i=0;i<n;i++)
                  a[i]=v[i]*c;

      return;
}

void multiply(double** C,const double c,double const*const*A,const int m,const int n)
{
               for (int i=0;i<m;i++)
                        for (int j=0;j<n;j++)
                                 C[i][j]=A[i][j]*c;

               return ;
}
//*=
void multiply(double**A, int m,int n1,double const*const*B,int n2,int k)
{

               if (n1!=n2)
                        return ;
               if (n1!=k)
                        return ;
               int n=n1;
               double **C=new double*[m];
               for (int i=0;i<m;i++)
```

```
                {
                        C[i]=new double[n];
                        for (int j=0;j<n;j++)
                        {
                                        C[i][j]=0;
                                        for (int l=0;l<n;l++)
                                            C[i][j]+=A[i][l]*B[l][j];
                        }
        }
            for (int i=0;i<m;i++)
                    for (int j=0;j<n;j++)
                                        A[i][j]=C[i][j];

        for (int i=0;i<m;i++)
            delete[] C[i];

                delete C;
                return;
}



 void multiply(double *v,const int n,const double c)
{
                for(int i=0;i<n;i++)
                    v[i]=v[i]*c;

        return ;
}

void multiply(double **A,const int m,const int n,const double c)
{
                for (int i=0;i<m;i++)
                        for (int j=0;j<n;j++)
                                A[i][j]=A[i][j]*c;

                return ;
}


//сложение
void plus(double** C,double const*const*A,const int m1,const int n1, double
const*const*B,const int m2,const int n2)
{
                if (n1!=n2||m1!=m2)
                        return ;

                int n=n1;
                int m=m1;

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

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

        return ;
}

void plus(double* C,const double*A, const int n1, const double *B, const int n2)
{
        if (n1!=n2)
                return;

        int n=n1;

        for (int i=0;i<n;i++)
                C[i]=A[i]+B[i];

        return;
}

void plus(double* a,const double *v,const int n,const double c)
{
        for(int i=0;i<n;i++)
           a[i]=v[i]+c;

    return ;
}

void plus(double** C,double const*const*A,const int m,const int n,const double c)
{
        for (int i=0;i<n;i++)
                C[i]=new double[n];

        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=A[i][j]+c;

        return ;
}

void plus(double* a,const double c,const double *v,const int n)
{
        for(int i=0;i<n;i++)
           a[i]=v[i]+c;

    return ;
}

void plus(double** C,const double c,double const*const*A,const int m, const int n)
{
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=A[i][j]+c;
```

```
		return;
	}
	//+=

	void plus( double**A,const int m1,const int n1, double const*const*B,const int m2,const int
n2)
	{
		if (n1!=n2||m1!=m2)
			return ;

		int n=n1;
		int m=m1;

		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
					A[i][j]+=B[i][j];

		return;
	}

	void  plus( double*A,const int n1, const double *B,const int n2)
	{
		if (n1!=n2)
			return ;

		int n=n1;

		for (int i=0;i<n;i++)
			A[i]+=B[i];

		return;
	}

	void  plus(double *v,const int n,const double c)
	{
		for(int i=0;i<n;i++)
			v[i]=v[i]+c;
	return;
	}

	void plus(double **A,const int m,const int n,const double c)
	{
		for (int i=0;i<m;i++)
			for (int j=0;j<n;j++)
				A[i][j]+=c;


		return;
	}


	//минус
```

```
void minus(double** C,double const*const*A,const int m1,const int n1, double
const*const*B,const int m2,const int n2)
{
        if (n1!=n2||m1!=m2)
            return ;

        int n=n1;
        int m=m1;

        for (int i=0;i<m;i++)
            for (int j=0;j<n;j++)
                C[i][j]=A[i][j]-B[i][j];

        return ;
}

void minus(double* C,const double*A, const int n1, const double *B, const int n2)
{
        if (n1!=n2)
            return;

        int n=n1;

        for (int i=0;i<n;i++)
            C[i]=A[i]-B[i];

        return;
}

void minus(double* a,const double *v,const int n,const double c)
{
       for(int i=0;i<n;i++)
          a[i]=v[i]-c;

    return ;
}

void minus(double** C,double const*const*A,const int m,const int n,const double c)
{
        for (int i=0;i<m;i++)
            for (int j=0;j<n;j++)
                C[i][j]=A[i][j]-c;

        return ;
}

void minus(double* a,const double c,const double *v,const int n)
{
       for(int i=0;i<n;i++)
          a[i]=-v[i]+c;

    return ;
```

```
}

void minus(double** C,const double c,double const*const*A,const int m, const int n)
{
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[i][j]=-A[i][j]+c;

        return;
}

//-=

void minus( double**A,const int m1,const int n1, double const*const*B,const int m2,const int
n2)
{
        if (n1!=n2||m1!=m2)
                return ;

        int n=n1;
        int m=m1;

        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        A[i][j]-=B[i][j];

        return;
}

void  minus( double*A,const int n1, const double *B,const int n2)
{

        if (n1!=n2)
                return ;

        int n=n1;

        for (int i=0;i<n;i++)
                A[i]-=B[i];

        return;
}

void  minus(double *v,const int n,const double c)
{
        for(int i=0;i<n;i++)
                v[i]-=c;
        return;
}

void minus(double **A,const int m,const int n,const double c)
{
```

```
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        A[i][j]-=c;


        return;
}


//-----------------------------------------------------------

void transpose_matrix(double **C,double const*const*A,const int m,const int n)
{
        for (int i=0;i<m;i++)
                for (int j=0;j<n;j++)
                        C[j][i]=A[i][j];

        return ;
}

double determinant(double const*const*A,const int m,const int n)
{
    if (m!=n||m!=2)
            return 0;

    return A[0][0]*A[1][1]-A[0][1]*A[1][0];
}

double norma(const double *v,const int n)
{
        double a=0;
        for(int i=0;i<n;i++)
          a+=v[i]*v[i];
        return sqrt(a);
}

double scalar_product(const double *v1,const int n1,const double *v2,const int n2)
{
        if (n1!=n2)
                return 0;

        int n=n1;
        double C=0;
        for (int i=0;i<n;i++)
                C+=v1[i]*v2[i];

        return C;
}

void stroka(double*C,double const*const*A,const int m,const int n,const int i)
{
    for (int j=0;j<n;j++)
```

```
                                C[j]=A[i][j];

        return ;
}

void stolbec(double* C,double const*const*A,const int m,const int n,const int i)
{
        for (int j=0;j<m;j++)
                                C[j]=A[j][i];

        return;
}

void put_stroka(double**A,const int m,const int n1,const double *C,const int n2,const int i)
{
        if(n1!=n2)
          return;

        int n=n1;

        for (int j=0;j<n;j++)
                                A[i][j]=C[j];

        return ;
}

void put_stolbec(double**A,const int n1,const int m,const double *C,const int n2,const int i)
{
        if(n1!=n2)
          return;

        int n=n1;

        for (int j=0;j<n;j++)
                                A[j][i]=C[j];

        return ;
}
//обратная матрица
double ** obratnaya_matrica(double **E,double const*const*A,const int m,const int n)
{
        if (m!=n)
          return 0;

          double **temp_m=new double*[m],*temp_v=new double[m],*temp_v1;
          for (int i=0;i<n;i++)
                        temp_m[i]=new double[m];


                for (int i=0;i<m;i++)
                        for (int j=0;j<n;j++)
                                temp_m[i][j]=A[i][j];
```

```
    for (int i=0;i<m;i++)
                    for (int j=0;j<n;j++)
                        if (i!=j)
                                    E[i][j]=0;
                        else
                          E[i][j]=1;

            for (int i=0;i<n;i++)
            {
                    double max=0;
                    int n_max=0;
                  for (int j=i;j<n;j++)
                            if (fabs(temp_m[i][j])>max)
                            {
                              max=fabs(temp_m[i][j]);
                              n_max=j;
                          }

                    temp_v1=temp_m[i];
                    temp_m[i]=temp_m[n_max];
                    temp_m[n_max]=temp_v1;
                    temp_v1=E[i];
                    E[i]=E[n_max];
                    E[n_max]=temp_v1;

                    multiply(temp_m[i],m,1/temp_m[i][i]);
                    multiply(E[i],m,1/temp_m[i][i]);

                    for (int j=0;j<n;j++)
                        if (i!=j)
                        {
                          stroka(temp_v,temp_m,m,m,i);
                          multiply(temp_v,m,temp_m[j][i]);
                                minus(temp_m[j],m,temp_v,m);
                          stroka(temp_v,E,m,m,i);
                          multiply(temp_v,m,temp_m[j][i]);
                                minus(temp_m[j],m,temp_v,m);
            }
        }
          for (int i=0;i<n;i++)
                     delete[] temp_m[i];

            delete[] temp_m;
            delete[] temp_v;
        return E;
    }


    void least_squares(double *ans,double const*const* R,const int m,const int n,double
const*z,const int m1)
    {
```

```
        if(m1!=m) return;
        double **temp_m=new double*[n],**temp_m1=new double*[n];
       double *r=new double[n];
       double **Rt=new double*[n];

       for (int j=0;j<n;j++)
                Rt[j]=new double[m];

                for (int i=0;i<n;i++)
                {
                        temp_m[i]=new double[n];
                        temp_m1[i]=new double[n];
                }
                transpose_matrix(Rt,R,m,n);
        multiply(r,Rt,n,m,z,m);
       multiply(temp_m,Rt,n,m,R,m,n);
       obratnaya_matrica(temp_m1,temp_m,n,n);
       multiply(ans,temp_m1,n,n,r,n);

       for (int j=0;j<n;j++)
        delete[] Rt[j];

                delete[] Rt;

                for (int i=0;i<n;i++)
                {
                                        delete[] temp_m[i];
                                        delete[] temp_m1[i];
         }

                delete[] temp_m;
                delete[] temp_m1;
                delete[] r;
}
//создание массива и вектора и запись в них нулей

double** rzeros(double**A,const int m,const int n)
{
      A=new double*[m];
      for (int i=0;i<m;i++)
      {
                A[i]=new double[n];
                for (int j=0;j<n;j++)
                        A[i][j]=0;
    }

    return A;
}

double* rzeros(double*A,const int n)
{
      A=new double[n];
```

```
	for (int j=0;j<n;j++)
			A[j]=0;

    return A;
}
//запись нулей в массив и вектор

void zeros(double**A,const int n,const int m)
{

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

		for (int j=0;j<n;j++)
			A[i][j]=0;
    }
    return;
}

void zeros(double*A,const int n)
{

	for (int j=0;j<n;j++)
			A[j]=0;

    return;
}
//создание массива и вектора и запись в них единиц

double** rones(double**A,const int m,const int n)
{
	A=new double*[m];
	for (int i=0;i<m;i++)
	{
		A[i]=new double[n];
		for (int j=0;j<n;j++)
			A[i][j]=1;
    }
    return A;
}

double* rones(double*A,const int n)
{
	A=new double[n];
	for (int j=0;j<n;j++)
			A[j]=1;

    return A;
}
//запись единиц в массив и вектор

void ones(double**A,const int m,const int n)
```

```
{

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

		for (int j=0;j<n;j++)
			A[i][j]=1;
    }
    return;
}

void ones(double*A,const int n)
{

	for (int j=0;j<n;j++)
			A[j]=1;

    return;
}

//печать

void print(double const*v,int n)
{
	for (int i=0;i<n;i++)
		cout<<v[i]<<"\n";

		return;
}

void print(double const*const*A,int m,int n)
{
	for (int i=0;i<m;i++)
	{
		for (int j=0;j<n;j++)
			cout<<A[i][j]<<" ";
		cout<<"\n";
    }

		return;
}

void print_f(double const*v,int n,const char*s)
{
	FILE *f;
	f=fopen(s,"w");
	fprintf(f,"%d\n",n);
	for (int i=0;i<n;i++)
		fprintf(f,"%lf\n",v[i]);

	fclose(f);
		return;
```

```
}

void print_f(double const*const*A,int m,int n,const char*s)
{
	FILE *f;
	f=fopen(s,"w");
	fprintf(f,"%d\n",m);
	fprintf(f,"%d\n",n);
	for (int i=0;i<m;i++)
	{
		for (int j=0;j<n;j++)
			fprintf(f,"%lf ",A[i][j]);
		fprintf(f,"\n");
    }
	fclose(f);
		return;
}
//удаление


void destroy(double **A,int m)
{
	for (int i=0;i<m;i++)
	{
		delete[] A[i];
		A[i]=0;
    }

	delete[] A;
	A=0;
		return;
}

void destroy1(double **A,int m)
{
	for (int i=0;i<m;i++)
	{
		free(A[i]);
		A[i]=0;
    }

	free(A);
	A=0;
		return;
}
```