ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ

Вычислительная геометрия зародилась в области проектирования и анализа алгоритмов в конце 1970-х годов. Впоследствии она превратилась в самостоятельную дисциплину со своими журналами, конференциями и большим сообществом активно работающих исследователей. Ее успех как отдельной отрасли научных исследований можно, с одной стороны, объяснить красотой изучаемых задач и их решений, а, с другой, многочисленностью областей применения – компьютерная графика, геоинформационные системы (ГИС), робототехника и другие – в которых геометрические алгоритмы играют важнейшую роль.

Есть много геометрических задач, первые алгоритмы решения которых либо работали очень медленно, либо были трудны для понимания и реализации. В последние годы разработан ряд новых алгоритмических подходов, которые позволили улучшить или упростить прежние решения. В этой книге мы попытались представить современные алгоритмы в виде, доступном широкой аудитории. Книга задумана как учебник по курсу вычислительной геометрии, но может использоваться и для самообразования.

Организация книги. Каждая из шестнадцати глав (кроме введения) начинается с некоторой прикладной задачи. Эта задача переформулируется как чисто геометрическая, а затем решается методами вычислительной геометрии. Сама геометрическая задача, а также идеи и приемы ее решения и составляют тему главы. Выбор приложений продиктован темами, которые мы хотели охватить, и не может считаться исчерпывающим обзором всех применений вычислительной геометрии. Мы ставили перед собой задачу пробудить любопытство у читателя, а не продемонстрировать готовые решения. Вместе с тем, мы полагаем, что знание вычислительной геометрии важно для эффективного решения геометрических задач в самых разных областях. Мы надеемся, что наша книга вызовет интерес не только у алгоритмистов, но и у специалистов-прикладников.

Для большинства рассмотренных геометрических задач мы приводим только одно решение, даже если их существует несколько. В общем случае мы выбирали то решение, которое проще всего понять и реализовать, пусть даже оно не самое эффективное. Мы также постарались включить достаточное число различных технических приемов, например: метод «разделяй и властвуй», заметание плоскостью и рандомизированные алгоритмы. Мы решили не рассматривать всевозможные варианты каждой задачи, нам казалось важнее познакомить читателя со

всеми основными вопросами вычислительной геометрии, чем увязнуть в деталях немногих избранных тем.

Некоторые разделы отдельных глав помечены звездочками. В них описываются улучшенные решения, обобщения или взаимосвязи между разными задачами. Для понимания остального материала они необязательны.

Каждая глава завершается разделом «Замечания». Здесь приводятся ссылки на источники приведенных в данной главе результатов, упоминаются другие решения, обобщения и усовершенствования и дается обзор литературы. Эти разделы можно пропустить, но они содержат полезную информацию для тех, кто хочет узнать больше о теме главы.

В конце каждой главы имеются упражнения. Они варьируются от простых проверочных вопросов на усвоение до более сложных задач, расширяющих пройденный материал. Трудные упражнения, как и те, что относятся к материалу из разделов со звездочкой, помечены звездочкой.

План курса. Хотя главы книги в значительной мере независимы, мы не рекомендуем читать их в произвольном порядке. Например, глава 2 знакомит с алгоритмами заметания плоскости, поэтому читать ее лучше до глав, в которых этот метод используется. Точно так же главу 4 лучше прочитать раньше глав, в которых применяются рандомизированные алгоритмы.

В начальный курс вычислительной геометрии мы советуем включить материал глав 1–10 в указанном порядке. В них рассматриваются понятия и методы, которые, на наш взгляд, должны быть освещены в любом курсе вычислительной геометрии. Если есть желание включить дополнительный материал, то можно взять любые из оставшихся глав.

Предварительные требования. Книгу можно использовать как учебник для курса, читаемого при подготовке бакалавров или магистров, – в зависимости от общей структуры учебного плана в данном учебном заведении. Предполагается, что у читателя имеются базовые знания по проектированию и анализу алгоритмов и структурам данных. Читатель должен быть знаком с нотацией «О большое» и простыми алгоритмами, в том числе с сортировкой, двоичным поиском и сбалансированными деревьями поиска. Не предполагается никаких знаний о предметной области и почти никаких по геометрии. При анализе рандомизированных алгоритмов используется элементарная теория вероятностей.

Реализации. Алгоритмы в этой книге представлены на псевдокоде высокого уровня, но описаны достаточно детально, для того чтобы их можно было без труда реализовать. В частности, мы старались всюду обращать внимание на обработку вырожденных случаев, которые зачастую вызывают трудности, когда дело доходит до реализации.

Мы полагаем весьма полезным самостоятельно реализовать несколько алгоритмов, т. к. это позволит составить представление об их практической сложности. Каждую главу можно рассматривать как программный проект. В зависимости от

располагаемого времени читатель может реализовать только сами геометрические алгоритмы или приложение в целом.

Для реализации геометрических алгоритмов нужны базовые типы данных: точки, прямые, многоугольники и т. д. и процедуры для операций над ними. Корректная реализация этих процедур – непростое дело, требующее много времени. И хотя сделать это один раз никому не помешает, полезно все же иметь библиотеку, содержащую базовые типы данных и процедуры. Ссылки на такие библиотеки приведены на нашем сайте.

Веб-сайт. Этой книге сопутствует веб-сайт, на котором имеются списки опечаток, найденных в каждом издании, все рисунки, псевдокод всех алгоритмов, а также другие материалы. Его адрес:

```
http://www.cs.uu.nl/geobook/
```

На сайте указан адрес, по которому можно присылать замеченные опечатки, равно как и любые замечания по поводу книги.

О третьем издании. В третье издание включены два важных дополнения: в главе 7, посвященной диаграммам Вороного, обсуждаются диаграмма Вороного отрезков прямых и дальняя диаграмма Вороного. А в главу 12 включен раздел о деревьях двоичного разбиения пространства для сцен малой плотности – как введение в реалистичные практические модели. Кроме того, исправлено множество мелких и ряд серьезных ошибок (см. список опечаток во второй издании, опубликованный на сайте). Мы также обновили замечания к каждой главе, включив ссылки на последние результаты и литературу по теме. Мы старались не изменять нумерацию разделов и упражнений, чтобы студенты могли и дальше пользоваться вторым изданием книги.

Благодарности. Написание учебника – длительный процесс, даже когда у него четыре автора. Многие помогали нам при подготовке самого первого издания полезными советами о том, что включать в книгу, прочтением черновых вариантов глав, рекомендациями по внесению изменений и исправлению ошибок. Еще больше народу присылали отзывы и находили ошибки после выхода первых двух изданий. Мы благодарим всех, а в особенности Панкаджа Агарвала (Pankaj Agarwal), Гельмута Альта (Helmut Alt), Маршалла Берна (Marshall Bern), Джита Бозе (Jit Bose), Хейзел Эверетт (Hazel Everett), Джеральда Фарина (Gerald Farin), Стива Форчун (Steve Fortune), Геерта-Яна Гиземана (Geert-Jan Giezeman), Мордехая Голина (Mordecai Golin), Дэна Гальперина (Dan Halperin), Ричарда Карпа (Richard Karp), Мэттью Катца (Matthew Katz), Клару Кедем (Klara Kedem), Нельсона Макса (Nelson Max), Джозефа С. Б. Митчелла (Joseph S. B. Mitchell), Рене ван Оострума (René van Oostrum), Гюнтера Роте (Günter Rote), Генри Шапиро (Henry Shapiro), Свена Скюма (Sven Skyum), Джека Сноейинка (Jack Snoeyink), Герта Вегтера (Gert Vegter), Петера Видмайера (Peter Widmayer), Чии Япа (Chee Yap) и Гюнтера Циглера (Günther Ziegler). Мы также благодарны сотрудникам издатель-

ства Springer-Verlag за советы и помощь в ходе работы над первым и последующими изданиями книги и ее переводами на другие языки (на данный момент – японский, китайский и польский).

Наконец, мы выражаем признательность за поддержку Нидерландской организации научных исследований (Netherlands' Organization for Scientific Research – N.W.O.) и Корейскому исследовательскому фонду (Korea Research Foundation – KRF).

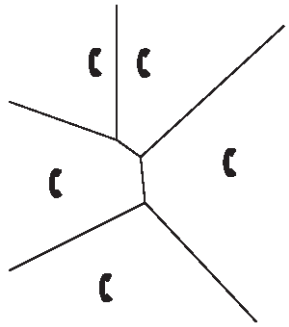
Январь 2008

Марк де Берг
Отфрид Чеонг
Марк ван Кревельд
Марк Овермарс

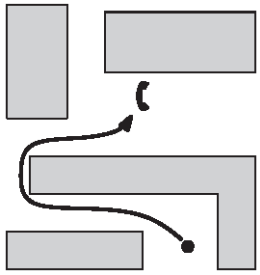
ГЛАВА 1.
Вычислительная геометрия

Введение

Представьте, что вы идете по территории университетского городка (кампуса) и внезапно вспоминаете, что должны срочно позвонить. На территории много телефонных будок и, естественно, вам нужна ближайшая. Но какая из них ближайшая? Хорошо бы иметь карту, на которой можно найти ближайшую будку, в какой бы точке кампуса вы ни находились. На такой карте было бы показано разбиение кампуса на области и для каждой из них – ближайшая телефонная будка. Как выглядят такие области? И как их построить?



Сама по себе поставленная проблема не особенно важна, но это демонстрация фундаментальной геометрической концепции, играющей роль во многих приложениях. Описанное разбиение кампуса называется диаграммой Вороного, она является предметом главы 7. Эту идею можно использовать для моделирования торговых зон городов, управления движением роботов и даже для описания и моделирования роста кристаллов. Для вычисления таких геометрических структур, как диаграммы Вороного, нужны геометрические алгоритмы. Эти алгоритмы и составляют содержание данной книги.



Другой пример. Допустим, вы нашли ближайшую телефонную будку. С картой кампуса в руках вы, надо полагать, без особого труда найдете достаточно короткий путь в обход стен и других препятствий. Но вот запрограммировать робота для решения той же задачи будет посложнее. И в этом случае задача имеет геометрическую природу: при заданном наборе геометрических препятствий найти кратчайший путь между двумя точками, избегающий столкновений с препятствиями. Решение этой задачи *планирования движения* чрезвычайно важно для робототехники. Главы 13 и 15 посвящены геометрическим алгоритмам, используемым при планировании движения.

Третий пример. Допустим, что у вас не одна карта, а две: на одной нанесены различные здания, в т. ч. и телефонные будки, а на другой – дороги на территории кампуса. Чтобы спланировать маршрут к будке, мы должны *наложить* карты друг на друга, т. е. объединить содержащуюся в них информацию. Наложение карт – одна из основных операций в геоинформационных системах. Для этого необходимо находить положение объектов одной карты на другой, вычислять пересечения различных множеств и т. д. Эта задача рассматривается в главе 2.

И это всего лишь три примера геометрических задач, для решения которых необходимы тщательно спроектированные геометрические алгоритмы. Вычислительная геометрия как наука, в рамках которой рассматриваются такие задачи, зародилась в 1970-х годах. Ее можно определить как систематическое изучение алгоритмов и структур данных, имеющих отношение к геометрическим объектам, с упором на поиск точных алгоритмов, работающих асимптотически быстро. Многих ученых заинтриговали трудности, связанные с геометрическими задачами. Путь от постановки задачи до эффективного и элегантного решения часто оказывался долгим и был усеян множеством трудностей и неоптимальных промежуточных результатов. Сегодня в нашем распоряжении имеется богатый набор эффективных геометрических алгоритмов, относительно простых для понимания и реализации.

В этой книге описываются важнейшие понятия, методы, алгоритмы и структуры данных вычислительной геометрии в виде, который, как мы надеемся, понравится читателям, интересующимся приложениями этой науки. В начале каждой главы ставится реальная задача, для решения которой необходимы геометрические алгоритмы. Чтобы продемонстрировать широкую применимость вычислительной геометрии, мы брали задачи из разных областей: робототехника, компьютерная графика, САПР/АСУП (CAD/CAM) и геоинформационные системы.

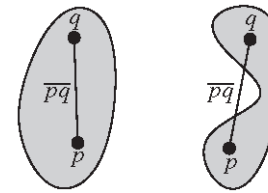
Вы не найдете здесь готовых решений задач в различных прикладных областях. В каждой главе рассматривается одна концепция вычислительной геометрии, а приложения служат лишь для пояснения и иллюстрации концепций и в качестве примеров моделирования инженерной задачи и поиска ее точного решения.

1.1. Пример: выпуклые оболочки

Хорошее решение алгоритмической задачи геометрического характера обычно покоится на двух столпах. Первый – ясное понимание геометрических свойств задачи, второй – правильное применение алгоритмов и структур данных. Если вы не понимаете геометрию задачи, то никакой алгоритм не поможет решить ее эффективно. С другой стороны, даже если геометрия задачи вам совершенно ясна, эффективно решить без знакомства с подходящими алгоритмическими приемами будет трудно. В этой книге вы найдете объяснения наиболее важных геометрических концепций и алгоритмических приемов.

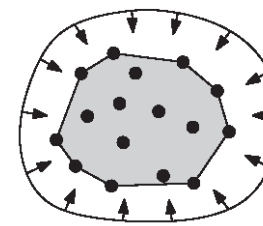
Для иллюстрации проблем, возникающих при разработке геометрического алгоритма, в этом разделе мы рассмотрим одну из первых задач, изучаемых в курсе вычислительной геометрии: построение выпуклых оболочек на плоскости. Оста-

вим в стороне вопрос о том, зачем это нужно; если вам интересно, прочитайте введение к главе 11, посвященной выпуклым оболочкам в трехмерном пространстве.



Множество S на плоскости называется *выпуклым*, если для любых двух точек $p,q \in S$ весь отрезок $\overline{pq}$ принадлежит S. Выпуклой оболочкой $\mathcal{CH}(S)$ множества S называется наименьшее выпуклое множество, содержащее S. Точнее, это пересечение всех выпуклых множеств, содержащих S.

Изучим задачу о вычислении выпуклой оболочки конечного множества P, состоящего из n точек на плоскости. Чтобы наглядно представить, как выглядит выпуклая оболочка, поставим мысленный эксперимент. Допустим, что в каждой точке вбит гвоздик, возьмем эластичную резинку, окружим ей все гвоздики и отпустим. Резинка плотно охватит гвоздики, стремясь к минимальной длине. Область внутри резинки и будет выпуклой оболочкой P. Это подводит нас к другому определению выпуклой оболочки конечного множества P точек на плоскости как однозначно определенного выпуклого многоугольника, вершинами которого являются точки, принадлежащие P, и который содержит все точки P. Разумеется, необходимо строго доказать, что это определение корректно – т. е. что такой многоугольник действительно единственный – и что оно эквивалентно предыдущему, но в этой вводной главе мы опустим эту деталь.



Как вычислить выпуклую оболочку? Прежде чем отвечать на этот вопрос, нужно задать себе другой: что вообще означает выражение «вычислить выпуклую оболочку»? Как мы видели, выпуклая оболочка множества P – это выпуклый многоугольник. Многоугольник естественно представлять путем перечисления его вершин по часовой стрелке, начав с любой вершины. Таким образом, мы хотим решить такую задачу: для заданного множества $P = \{p_1, p_2, ..., p_n\}$ точек на плоскости найти список, содержащий те точки P, которые являются вершинами $\mathcal{CH}(P)$, в порядке обхода по часовой стрелке.

вход = множество точек:
$p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8, p_9$
выход = представление выпуклой оболочки:
p_4, p_5, p_8, p_2, p_9

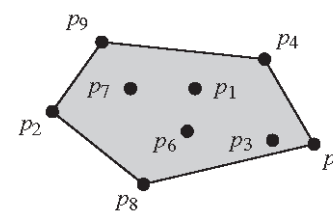
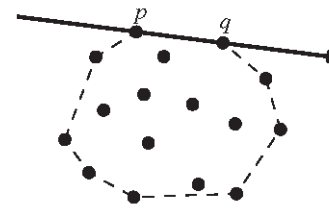


Рис. 1.1. Вычисление выпуклой оболочки

Первое определение выпуклой оболочки бесполезно для проектирования алгоритма ее вычисления. В нем говорится о пересечении всех выпуклых множеств, содержащих P, а таких бесконечно много. Тот факт, что $\mathcal{CH}(P)$ – выпуклый много-



угольник, более полезен. Посмотрим, каковы стороны $\mathcal{CH}(P)$. Обе вершины p и q его стороны являются точками P, и если мы проведем прямую через p и q, выбрав ее направление, так чтобы $\mathcal{CH}(P)$ оказалась справа, то все точки P должны находиться справа от этой прямой. Обратное тоже верно: если все точки $P\backslash\{p,q\}$ лежат справа от направленной прямой, проходящей через p и q, то отрезок $\overline{pq}$ является стороной $\mathcal{CH}(P)$.

Теперь, когда мы стали лучше понимать геометрию задачи, можно переходить к разработке алгоритма. Опишем его в виде псевдокода, как и все последующие алгоритмы в этой книге.

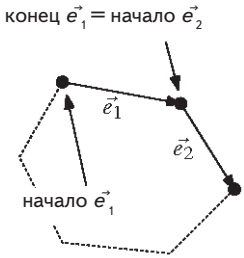
Алгоритм SlowConvexHull(P)
Вход. Множество P точек на плоскости.
Выход. Список $\mathcal{L}$, содержащий вершины $\mathcal{CH}(P)$ в порядке обхода по часовой стрелке.

1. $E \leftarrow \emptyset$.
2. **for** всех упорядоченных пар $(p,q) \in P \times P$ таких, что p не равно q
3. **do** *valid* ← **true**
4. **for** всех точек $r \in P$, не равных p или q
5. **do if** r находится слева от направленной прямой из p в q
6. **then** *valid* ← **false**.
7. **if** *valid* **then** добавить направленный отрезок $\vec{pq}$ в E.
8. По множеству сторон E построить список $\mathcal{L}$ вершин, входящих в $\mathcal{CH}(P)$, отсортировав его в порядке обхода по часовой стрелке.

Два шага этого алгоритма, возможно, нуждаются в пояснении.

Сначала строка 5: как проверить, с какой стороны направленной прямой лежит точка? Это одна из примитивных операций, необходимых в большинстве геометрических алгоритмов. В этой книге мы будем предполагать, что такие операции где-то реализованы. Ясно, что их можно выполнить за постоянное время, так что фактическая реализация не оказывает влияние на асимптотическую сложность. Мы вовсе не хотим сказать, что подобные примитивные операции не важны или тривиальны. Напротив, их не так-то просто реализовать правильно, а от качества реализации зависит фактическое время работы. Но, к счастью, в наши дни имеются соответствующие библиотеки. Поэтому не будем домать голову над проверкой в строке 5, а просто предположим, что существует функция, которая производит такую проверку за постоянное время.

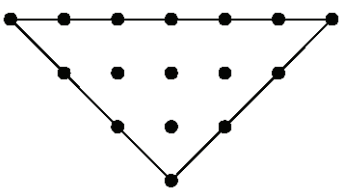
Также требует пояснения последний шаг алгоритма. В цикле в строках 2–7 мы строим множество E сторон выпуклой оболочки. По E можно следующим образом построить список $\mathcal{L}$. Элементы E представляют собой направленные отрезки, поэтому можно говорить о начальной и конечной вершине стороны. Поскольку любая сторона направлена так, что все остальные точки P лежат справа от нее, то



при перечислении вершин в порядке по часовой стрелке конечная вершина оказывается после начальной. Теперь удалим из E произвольную сторону $\vec{e}_1$. Поместим начало $\vec{e}_1$ в список $\mathcal{L}$, сделав его первой точкой, тогда второй точкой будет конец $\vec{e}_1$. Найдем в E сторону $\vec{e}_2$, для которой началом является конец $\vec{e}_1$, удалим ее из E и добавим в $\mathcal{L}$ ее конец. Далее, найдем сторону $\vec{e}_3$, началом которой является конец $\vec{e}_2$, удалим ее из E и добавим конец в $\mathcal{L}$. Будем продолжать, пока в E не останется всего одна сторона; ее конец обязательно совпадет с началом $\vec{e}_1$, а это уже и так первая точка в списке $\mathcal{L}$. Простая реализация этой процедуры занимает время $O(n^2)$. Его можно без особого труда сократить до $O(n \log n)$, но время, необходимое для работы остальной части алгоритма, все равно доминирует.

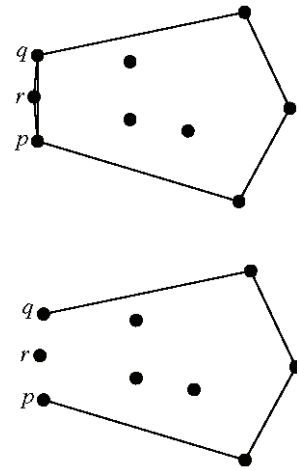
Проанализировать временную сложность алгоритма SlowConvexHull нетрудно. Мы проверяем $n^2 - n$ пар точек. Для каждой пары проверяется, все ли из оставшихся $n - 2$ точек лежат справа от прямой. Общее время составляет $O(n^3)$. Время выполнения последнего шага равно $O(n^2)$, так что полное время равно $O(n^3)$. Алгоритм с кубическим временем выполнения слишком медленный и может использоваться только для очень небольших входных наборов. Проблема в том, что мы даже не пытались применить сколько-нибудь изощренные приемы проектирования алгоритмов, а действовали «грубой силой». Но прежде чем искать решение получше, полезно будет сделать несколько наблюдений.

Мы несколько поторопились при выводе критерия принадлежности пары p, q к границе выпуклой оболочки $\mathcal{CH}(P)$. Точка r необязательно лежит слева или справа от прямой, проходящей через p и q, она может лежать и на этой прямой. И что тогда? Это так называемый *вырожденный случай*, или просто *вырожденность*. При первоначальном обдумывании задачи такие ситуации обычно игнорируются, чтобы не затемнять геометрические свойства задачи. Однако же на практике они встречаются. Например, если создавать точки, случайно щелкая мышью в разных местах экрана, то координатами всех точек будут небольшие целые числа, и с высокой вероятностью какие-то три точки окажутся на одной прямой.



Чтобы алгоритм был корректным и в вырожденном случае, мы должны переформулировать критерий следующим образом: направленный отрезок $\vec{pq}$ принадлежит $\mathcal{CH}(P)$ тогда и только тогда, когда все остальные точки $r \in P$ либо лежат строго справа от направленной прямой, проходящей через p и q, либо принадлежат открытому интервалу $\overline{pq}$. (Мы предполагаем, что в P нет совпадающих точек.) И строку 5 алгоритма следует заменить этой более сложной проверкой.

Мы проигнорировали еще один важный момент, от которого может зависеть правильность результата алгоритма. Мы неявно предполагали, что можем как-то узнать, лежит ли точка слева или справа от данной прямой. Но такое предположение не всегда справедливо: если координаты заданы числами с плавающей точкой



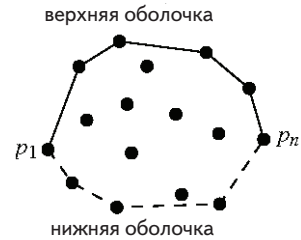
и вычисления тоже производятся над числами с плавающей точкой, то ошибки округления могут исказить результат проверки.

Пусть есть три почти коллинеарные точки p, q, r, а все остальные расположены далеко справа от них. В нашем алгоритме проверяются пары (p,q), (r,q) и (p,r). Поскольку эти точки почти коллинеарны, то из-за ошибок округления мы можем прийти к выводу, что r лежит справа от прямой pq, p лежит справа от прямой rq, а q лежит справа от прямой pr. Разумеется, геометрически такое невозможно, но арифметика с плавающей точкой об этом ничего не знает! В таком случае алгоритм включит все три отрезка в выпуклую оболочку. Еще хуже, если все три проверки дают разные ответы, и тогда алгоритм отвергнет все три отрезка, что приведет к разрыву выпуклой оболочки. А это станет причиной серьезной ошибки при построении отсортированного списка вершин выпуклой оболочки на последнем шаге алгоритма. Ведь там мы предполагаем, что для каждой вершины выпуклой оболочки существует ровно одна сторона, начинающаяся в ней, и ровно одна сторона, в ней заканчивающаяся. Но из-за ошибок округления может оказаться, что в некоторой вершине p начинаются две или ни одной стороны. Тогда программа, реализующая наш простой алгоритм, скорее всего, завершится аварийно, потому что мы не предусмотрели возможности столь противоречивых данных на входе.

И хотя мы доказали, что алгоритм правилен и обрабатывает все особые случаи, он не является *устойчивым*: небольшие погрешности в вычислениях могут приводить к неожиданным отказам. Проблема в том, что корректность доказана в предположении, что операции над вещественными числами выполняются точно.

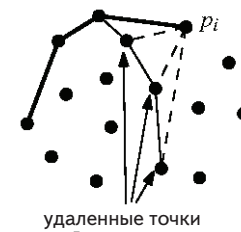
Вот мы и спроектировали свой первый геометрический алгоритм. Он вычисляет выпуклую оболочку множества точек на плоскости. Однако работает он очень медленно – за время $O(n^3)$, обрабатывает вырожденные случаи неэлегантно и не является устойчивым. Нужно продолжать попытки.

Для этого применим стандартный прием проектирования: разработаем *инкрементный алгоритм*. Это означает, что мы будем добавлять точки в P по одной, всякий раз изменяя ранее найденное решение. Придадим этому подходу геометрический привкус, добавляя точки слева направо. Итак, сначала отсортируем точки по координате x, получив при этом последовательность p_1, ..., p_n, а затем будет добавлять их в таком порядке. Поскольку точки просматриваются слева направо, было бы удобно, если бы вершины выпуклой оболочки в порядке следования вдоль ее границы тоже были упорядочены слева направо. Но это не так. Поэтому сначала мы вычислим только те вершины выпуклой оболочки, которые лежат в *верхней оболочке*, т. е. части выпуклой оболочки от самой левой точки p_1 до самой правой точки p_n при обходе по часовой стрелке. Другими словами, верхняя оболочка содержит стороны выпуклой оболочки, ограничивающие ее сверху. А на втором про-



ходе, выполняемом справа налево, вычислим оставшуюся часть выпуклой оболочки – *нижнюю оболочку*.

Основной шаг инкрементного алгоритма – обновление верхней оболочки после добавления точки p_i. Иначе говоря, уже известна верхняя оболочка точек $p_1, ..., p_{i-1}$, и требуется вычислить верхнюю оболочку точек $p_1, ..., p_i$. Опишем, как это можно сделать. При обходе границы многоугольника по часовой стрелке мы совершаем поворот в каждой вершине. Если многоугольник произвольный, то поворот может быть как направо, так и налево, но в случае выпуклого прямоугольника возможны только повороты направо. Раз так, то добавление p_i можно выполнить следующим образом. Обозначим $\mathcal{L}_{\text{upper}}$ список вершин верхней оболочки в порядке слева направо. Сначала добавляем p_i в $\mathcal{L}_{\text{upper}}$. Это допустимо, потому что p_i – самая правая из добавленных до сих пор точек, поэтому она должна принадлежать верхней оболочке. Затем проверяем, верно ли, что последние три точки в $\mathcal{L}_{\text{upper}}$ образуют поворот направо. Если это так, то больше ничего не нужно; $\mathcal{L}_{\text{upper}}$ содержит вершины верхней оболочки $p_1, ..., p_i$, и мы можем переходить к следующей точке p_{i+1}. Но если три последние точки образуют поворот налево, то среднюю нужно из верхней оболочки удалить. Но это еще не все: может оказаться, что новые последние три точки тоже не образуют поворот направо, и тогда нужно опять удалить среднюю. Так следует продолжать до тех пор, пока не окажется, что три последние точки образуют поворот направо или не останется всего две точки.



Вот теперь можно записать алгоритм на псевдокоде, в котором вычисляется верхняя и нижняя оболочка. Нижняя вычисляется так же, как верхняя, только точки просматриваются справа налево.

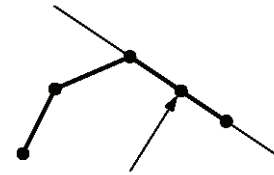
Алгоритм ConvexHull(P)
Вход. Множество P точек на плоскости.
Выход. Список, содержащий вершины $\mathcal{CH}(P)$ в порядке обхода по часовой стрелке.

1. Отсортировать точки по координате x, получив в результате последовательность $p_1, ..., p_n$.
2. Поместить точки p_1 и p_2 в список $\mathcal{L}_{\text{upper}}$, сделав p_1 первой.
3. **for** $i \leftarrow 3$ **to** n
4. **do** добавить p_i в конец $\mathcal{L}_{\text{upper}}$.
5. **while** $\mathcal{L}_{\text{upper}}$ содержит более двух точек и последние три точки $\mathcal{L}_{\text{upper}}$ не образуют поворот направо
6. **do** удалить среднюю из трех последних точек из $\mathcal{L}_{\text{upper}}$.
7. Поместить точки p_n и p_{n-1} в список $\mathcal{L}_{\text{lower}}$, сделав p_n первой.
8. **for** $i \leftarrow n - 2$ **downto** 1
9. **do** добавить p_i в конец $\mathcal{L}_{\text{lower}}$.

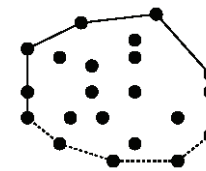
10. **while** $\mathcal{L}_{\text{lower}}$ содержит более двух точек и последние три точки $\mathcal{L}_{\text{lower}}$ не образуют поворот направо
11. **do** удалить среднюю из трех последних точек из $\mathcal{L}_{\text{lower}}$.
12. Удалить первую и последнюю точки из $\mathcal{L}_{\text{lower}}$, чтобы не дублировать точки, в которых верхняя и нижняя оболочки соединяются.
13. Добавить $\mathcal{L}_{\text{lower}}$ в конец $\mathcal{L}_{\text{upper}}$ и назвать получившийся список $\mathcal{L}$.
14. **return** $\mathcal{L}$

Но и на этот раз, приглядевшись внимательнее, мы обнаружим, что описанный алгоритм некорректен. Мы неявно предположили, что координаты x всех точек различны. Но если это предположение неверно, то упорядоченность точек по x не вполне определена. К счастью, это не очень серьезная проблема. Нужно лишь уточнить определение порядка: использовать не просто упорядоченность по координате x, а лексикографический порядок. Это означает, что сначала мы сортируем по x, а если координаты x двух точек совпадают, то дополнительно по y.

Еще один особый случай, на который мы не обратили внимания, – ситуация, когда три точки, для которых мы решаем вопрос о направлении поворота, на самом деле лежат на одной прямой. В этом случае средняя точка не должна принадлежать выпуклой оболочке, поэтому коллинеарные точки следует рассматривать так, будто они образуют поворот налево. Иными словами, проверка должна возвращать true, если три точки образуют поворот направо, и false в противном случае. (Отметим, что это проще проверки, которую приходилось производить в предыдущем алгоритме, когда встречались коллинеарные точки.)


нет поворота направо

С такими модификациями алгоритм правильно вычисляет выпуклую оболочку: на первом проходе вычисляется верхняя оболочка, которая теперь определяется как часть выпуклой оболочки от лексикографически наименьшей вершины до лексикографически наибольшей, а на втором – оставшаяся часть выпуклой оболочки.



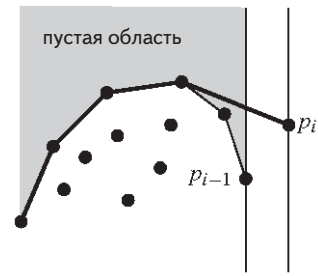
Как ведет себя наш алгоритм при наличии ошибок округления? Из-за таких ошибок из выпуклой оболочки может быть либо удалена точка, которая там должна присутствовать, либо не удалена точка, которая на самом деле находится внутри нее. Но структурная целостность алгоритма при этом не страдает: он все равно вычисляет замкнутый многоугольник. Ведь на выходе получается список точек, который можно интерпретировать как перечисление вершин многоугольника, и любые три точки образуют поворот направо или – в случае ошибок округления – почти поворот направо. Более того, ни одна точка, принадлежащая P, не может оказаться далеко за пределами вычисленной оболочки. Единственная проблема может возникнуть, когда три точки расположены очень близко друг к другу, и поворот, который в действительности является резким поворотом налево, интерпретируется как поворот направо. Это может привести к появлению зубца в

результирующем многоугольнике. Решить проблему можно, если рассматривать очень близкие точки как одну, например, с помощью предварительного округления. Конечно, результат при этом может получиться не вполне точным, но он, по крайней мере, имеет смысл, а рассчитывать на точный результат при неточных арифметических вычислениях в любом случае не приходится. Для многих приложений этого достаточно. Но все же разумно проявлять осмотрительность при реализации главной проверки, чтобы по возможности избежать погрешностей.

В заключение докажем следующую теорему.

Теорема 1.1. *Выпуклую оболочку множества n точек на плоскости можно вычислить за время $O(n \log n)$.*

Доказательство. Мы должны доказать корректность вычисления верхней оболочки; для нижней оболочки рассуждения точно такие же. Доказательство проведем индукцией по количеству уже рассмотренных точек. До начала цикла **for** список $\mathcal{L}_{\text{upper}}$ содержит точки p_1 и p_2, которые образуют тривиальную верхнюю оболочку $\{p_1, p_2\}$. Предположим теперь, что $\mathcal{L}_{\text{upper}}$ содержит вершины верхней оболочки $\{p_1, ..., p_{i-1}\}$ и рассмотрим добавление точки p_i. После выполнения цикла **while** мы, по предположению индукции, знаем, что точки $\mathcal{L}_{\text{upper}}$ образуют ломаную, которая поворачивает только направо. Кроме того, ломаная начинается в лексикографически наименьшей точке множества $\{p_1, ..., p_i\}$ и заканчивается в лексикографически наибольшей, а именно в точке p_i. Если мы сможем доказать, что все точки множества $\{p_1, ..., p_i\}$, не вошедшие в $\mathcal{L}_{\text{upper}}$, находятся под ломаной, то будет доказано, что $\mathcal{L}_{\text{upper}}$ содержит правильные точки. По предположению индукции, до добавления p_i не существовало точки выше ломаной. Поскольку старая ломаная лежит под новой, то выше новой ломаной точка могла бы оказаться лишь в одном случае: если она находится в вертикальной полосе между p_{i-1} и p_i. Но это невозможно, потому что такая точка должна была бы находиться между p_{i-1} и p_i в лексикографическом порядке. (Вам следует проверить, что это рассуждение сохраняет силу, даже если координаты x точки p_i и p_{i-1} или еще какой-то одинаковы.)



Для доказательства оценки сложности заметим, что лексикографическую сортировку точек можно выполнить за время $O(n \log n)$. Далее рассмотрим вычисление верхней оболочки. Число итераций цикла **for** линейно зависит от n. Остается вопрос о том, сколько раз выполняется внутренний цикл **while**. Для каждого выполнениния цикла **for** цикл **while** выполняется хотя бы один раз. При любом дополнительном выполнении одна точка удаляется из выпуклой оболочки. Поскольку каждую точку можно удалить не более одного раза, то общее число дополнительных выполнений по всем итерациям цикла **for** ограничено сверху значением n. Точно так же, вычисление нижней оболочки занимает время $O(n)$. А из-за сортировки общее время вычисления выпуклой оболочки составляет $O(n \log n)$.

Окончательный алгоритм вычисления выпуклой оболочки легко описать и реализовать. Для него нужны только две вещи: лексикографическая сортировка и проверка того, что три точки образуют поворот направо. Из первоначальной постановки задачи вовсе не было очевидно, что существует такое простое и эффективное решение.

1.2. Вырожденность и устойчивость

В предыдущем разделе мы видели, что в разработке геометрического алгоритма часто можно выделить три этапа.

На первом этапе мы игнорируем все, что мешает разобраться в геометрических концепциях, с которыми приходится иметь дело. Иногда помехой являются коллинеарные точки, иногда – вертикальные отрезки. При первой попытке спроектировать или понять алгоритм на такие вырожденные случаи обычно лучше не обращать внимания.

На втором этапе нужно подправить спроектированный на первом этапе алгоритм, чтобы он правильно работал в вырожденных случаях. Начинающие пытаются справиться с этой проблемой, вставляя в алгоритм обработку огромного числа разных случаев. Часто есть способ лучше. Еще раз изучив геометрию задачи, мы как правило можем включить особые случаи в общий. Например, в алгоритме вычисления выпуклой оболочке для обработки точек с одинаковыми координатами x нам понадобилось всего лишь применить лексикографическую сортировку вместо сортировки только по x. В большинстве алгоритмов, рассматриваемых в этой книге, мы стараемся применить такой интегрированный подход к особым случаям. Тем не менее, при первом чтении лучше о таких случаях не задумываться. И лишь поняв, как алгоритм работает в общем случае, переходить к вырожденным.

Читая литературу по вычислительной геометрии, вы обнаружите, что многие авторы игнорируют особые случая, часто формулируя специальные предположения о выходных данных. Например, в задаче о выпуклой оболочке мы могли бы поступить так, заранее предположив, что никакие три точки не лежат на одной прямой и координаты x всех точек различны. С теоретической точки зрения, такие предположения обычно оправданы: задача состоит в установлении вычислительной сложности алгоритма, а вырожденные случаи, при всей муторности обработки, почти никогда не увеличивают асимптотическую сложность. Но сложность реализации они, безусловно, увеличивают. Многие исследователи отчетливо понимают, что их предположения об *общем положении* в практических приложениях не удовлетворяются, и обычно было бы оптимально интегрировать особые случаи в общий алгоритм. Кроме того, существуют общие методы – так называемые *схемы символических возмущений* – которые позволяют игнорировать особые случаи при проектировании и реализации и, тем не менее, получать алгоритм, который правильно работает в вырожденных ситуациях.

Третий этап – собственно реализация. Теперь необходимо подумать о таких примитивных операциях, как проверка того, лежит ли точка слева, справа или на

прямой. В случае удачи найдется готовая библиотека, содержащая такие операции, иначе реализовывать их придется самостоятельно.

Еще одна проблема, возникающая на этапе реализации, – нарушение допущения о точности арифметических операций при работе с вещественными числами. Тут необходимо ясно представлять последствия. Неустойчивость зачастую является причиной разочарования при реализации геометрических алгоритмов. Решать проблему устойчивости нелегко. Одно из возможных решений – использовать пакет программ, предоставляющий точную арифметику (на основе целых, рациональных или даже алгебраических чисел, в зависимости от характера задачи). Или изменить алгоритм, так чтобы он сам обнаруживал противоречия и предпринимал корректирующие меры для предотвращения краха программы. В таком случае не гарантируется, что алгоритм порождает правильный результат, и важно установить, какими свойствами результат все же обладает. Именно так мы поступили в предыдущем разделе при разработке алгоритма вычисления выпуклой оболочки: результат может и не быть выпуклым многоугольником, но мы уверены в том, что его структура правильна, а нарушения выпуклости минимальны. Наконец, на основе входных данных можно предсказать, какая точность представления чисел необходима для правильного решения задачи.

Какой подход лучше, зависит от приложения. Если быстродействие – не самое главное, то предпочтительнее точная арифметика. Но бывает и так, что точность результата не слишком важна. Например, при отображении выпуклой оболочки множества точек небольшие отклонения от точной формы оболочки, скорее всего, будут незаметны. В таком случае можно воспользоваться тщательно продуманной реализацией на основе арифметики с плавающей точкой.

Далее нас будет интересовать, главным образом, этап проектирования геометрических алгоритмов, а о проблемах, возникающих на этапе реализации, мы говорить почти не будем.

1.3. Области применения

Как уже отмечалось, для каждой геометрической концепции, алгоритма или структуры данных, упоминаемых в книге, мы выбрали тот или иной иллюстративный пример. Основными областями применения вычислительной геометрии являются компьютерная графика, робототехника, геоинформационные системы и САПР/АСУП. Для читателей, не знакомых с этими дисциплинами, ниже приводится краткое введение и перечисляются некоторые возникающие в них геометрические задачи.

Компьютерная графика. Предметом компьютерной графики является создание изображений моделируемой сцены для отображения на экране компьютера, принтере или другом устройстве ввода-вывода. Сцена может быть как простым двумерным рисунком, состоящим из прямых, многоугольников и других примитивных объектов, так и реалистичной объемной сценой, включающей источники света, текстуры и т. д. В последнем случае для отображения сцены зачастую тре-

буется нарисовать миллионы многоугольников или фрагментов искривленных поверхностей.

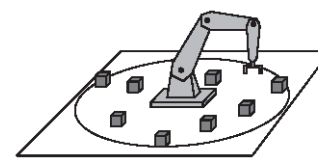
Поскольку сцена состоит из геометрических объектов, неудивительно, что геометрические алгоритмы играют важную роль в компьютерной графике.

Для двумерной графики типичны следующие задачи: найти пересечение некоторых примитивов, определить, на какой примитив указывает курсор мыши, вычислить подмножество примитивов, находящихся в данной области. Методы решения этих задач рассматриваются в главах 6, 10 и 16.

В трехмерном случае геометрические задачи усложняются. Важнейшим этапом отображения трехмерной сцены является удаление невидимых поверхностей: требуется определить, какая часть сцены видна из данной точки, или, иными словами, отбросить части, скрытые другими объектами. В главе 12 мы изучим один подход к решению этой задачи.

Для создания реалистичных сцен необходимо принимать в расчет освещение. Это порождает много новых проблем, например вычисление теней. Поэтому для синтеза реалистичных изображений необходимы сложные приемы, такие, как трассировка лучей и метод излучательности. Во всех этих ситуациях возникают геометрические задачи.

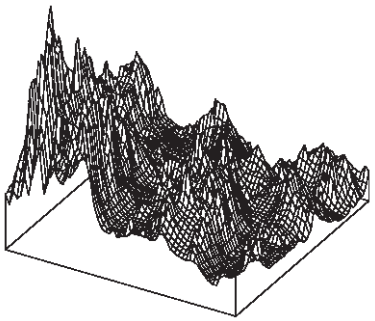
Робототехника. Предметом робототехники является проектирование и применение роботов. Поскольку роботы – это геометрические объекты, действующие в трехмерном пространстве – реальном мире – понятно, что геометрические задачи возникают повсеместно. В начале этой главы мы уже упоминали о задаче планирования движения, суть которой – вычислить траекторию движения робота в пространстве с препятствиями. В главах 13 и 15 мы изучим несколько простых примеров планирования движения. Но планирование движения – это лишь одна сторона более общей задачи планирования выполняемых задач. Мы хотим, чтобы роботу можно было поставить высокоуровневую задачу – «пропылесосить комнату» – рассчитывая на то, что робот сам найдет оптимальный способ ее выполнения. Сюда входит и планирование движения, и планирование порядка выполнения подзадач, и многое другое.



Другие геометрические задачи возникают при проектировании роботов и производственных ячеек, в которых робот должен работать. Большинство промышленных роботов – роботизированные руки (манипуляторы), закрепленные на неподвижном основании. Детали, обрабатываемые рукой робота, следует подавать так, чтобы робот мог легко ухватить их. Некоторые детали в процессе обработки должны быть неподвижны. Иногда деталь необходимо предварительно правильно ориентировать в пространстве. Все это геометрические задачи, иногда с кинематической составляющей. К ним применимы некоторые из описанных в книге алгоритмов. Например, задачу о наименьшем описанном круге, рассматриваемую в разделе 4.7, можно использовать для оптимального размещения руки робота.

Геоинформационные системы. В геоинформационной системе (ГИС) хранятся географические данные, например: формы границ стран, высота гор, русла рек, типы растительности в разных регионах, плотность населения, количество осадков. Там же могут храниться данные о строениях, возведенных человеком: городах, автомобильных и железных дорогах, линиях электропередач и газопроводах. ГИС можно использовать для получения сведений о регионах и, в частности, о взаимосвязях между данными разных типов. Например, биолога может интересовать связь между средним количеством осадков и существованием определенных растений, а инженера-строителя – проходит ли газовая труба под строительной площадкой, где планируется проводить выемку грунта.

Поскольку для большинства ГИС представляют интерес свойства точек и областей на поверхности Земли, геометрические задачи возникают здесь в изобилии. К тому же, объем данных настолько велик, что алгоритмы должны быть очень эффективны. Ниже мы скажем о том, какие относящиеся к ГИС задачи рассматриваются в этой книге.



Первая проблема – как хранить географические данные. Допустим, мы хотим разработать бортовую навигационную систему, которая в каждый момент времени показывает водителю местоположение автомобиля. Для этого необходимо хранить гигантскую карту дорог и другие данные. В каждый момент мы должны уметь определять местоположение автомобиля на карте и быстро выделять небольшой фрагмент карты для показа на экране бортового компьютера. Для этого нужны эффективные структуры данных. В главах 6, 10 и 16 описывается, какие решения предлагает вычислительная геометрия.

Информация о высоте в некоторых гористых местностях обычно доступна лишь для выборочных точек, а для всех остальных ее нужно получать с помощью интерполяции. Но по каким выборочным точкам интерполировать? Ответ на этот вопрос вы найдете в главе 9.

Комбинирование данных разных типов – одна из самых важных операций в ГИС. Например, часто требуется проверить, какие дома находятся в лесу, найти все мосты, т. е. точки пересечения дорог с реками, или подобрать хорошее место под новое поле для гольфа – немного холмистое, относительно дешевое и не слишком далеко от указанного города. Обычно в ГИС данные разных типов хранятся в отдельных картах. Чтобы объединить данные, мы должны наложить друг на друга разные карты. В главе 2 рассматривается проблема, возникающая при вычислении наложения.

Наконец, упомянем тот же пример, что в начале главы: поиск ближайшей телефонной будки (или больницы, или еще какого-то предприятия или заведения). Для этого необходимо вычислить диаграмму Вороного, структуру, которая будет детально изучена в главе 7.

САПР/АСУП. Системы автоматизированного проектирования (САПР) занимаются проектированием изделий с помощью компьютеров. Изделия могут быть самыми разными: печатные платы, детали машин, мебель и даже здания со всей «начинкой». В любом случае результатом является геометрический объект, и потому неудивительно, что возникают разнообразные геометрические задачи. В частности, САПР должны уметь вычислять пересечение и объединение объектов, раскладывать объекты и их границы на более простые фигуры и визуализировать спроектированное изделие.

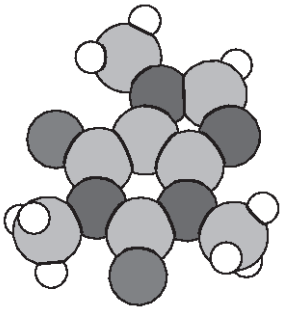
Чтобы понять, удовлетворяет ли изделие техническим условиям, необходимо провести испытания. Часто для этого не нужно создавать опытный образец, достаточно моделирования. В главе 14 решается задача, возникающая при моделировании тепловыделения печатной платы.

Спроектированное и испытанное изделие необходимо изготовить. Тут на помощь приходят автоматизированные системы управления производством (АСУП). В АСУП тоже возникает множество геометрических задач. Одна из них рассматривается в главе 4.

Недавно зародилось новое направление – *проектирование с учетом требований сборки* (design for assembly), когда технологические решения о порядке сборки принимаются уже на этапе проектирования. САПР, поддерживающая эту технологию, должна предоставлять проектировщику средства для проверки осуществимости проекта, т. е. отвечать на вопросы вида: можно ли без труда изготовить изделие при имеющемся процессе производства? Для ответа на многие подобные вопросы приходится использовать геометрические алгоритмы.

Другие области применения. Существует еще много областей, где возникают геометрические задачи, решаемые с помощью геометрических алгоритмов и структур данных.

Например, в молекулярном моделировании молекулу часто представляют в виде набора пересекающихся шаров в пространстве, по одному шару на каждый атом. Типичные вопросы – как вычислить объединение шаров-атомов для получения поверхности молекулы или как узнать, в каких точках две молекулы могут соприкасаться.



молекула кофеина

Еще одна область – распознавание образов. Возьмем, к примеру, систему оптического распознавания символов. Такая система сканирует бумажный текст, стремясь выделить в нем отдельные символы. Основной шаг – сопоставление изображения символа с коллекцией хранимых символов и поиск наилучшего соответствия. Это приводит к геометрической задаче: определить, насколько похожи два заданных геометрических объекта.

Даже в областях, на первый взгляд далеких от геометрии, можно с успехом применить геометрические алгоритмы, если удастся переформулировать задачу в геометрических терминах. Например, в главе 5 мы увидим, что записи базы данных

можно интерпретировать как точки в многомерном пространстве, и представим геометрическую структуру данных, позволяющую эффективно отвечать на некоторые запросы к базе.

Мы надеемся, что приведенное выше перечисление геометрических задач убедило вас в том, что вычислительная геометрия играет важную роль во многих компьютерных дисциплинах. Алгоритмы, структуры данных и методы, описанные в этой книге, снабдят вас инструментарием для успешного решения таких задач.

1.4. Замечания

Каждая глава этой книги заканчивается разделом «Замечания». В этих разделах указаны источники описанных в главе результатов, намечены направления обобщения и улучшения и приведены библиографические ссылки. Их можно пропустить, но те, кто хочет узнать больше о материале, изложенном в главе, найдут в них полезную информацию. Дополнительные сведения можно найти также в книгах «Handbook of Computational Geometry» [331] и «Handbook of Discrete and Computational Geometry» [191].

В этой главе была подробно рассмотрена одна геометрическая задача: вычисление выпуклой оболочки множества точек на плоскости. Эта классическая задача вычислительной геометрии, которой посвящена обширная литература. Описанный в этой главе алгоритм часто называют *сканированием Грэхема*, он основан на одном из самых первых алгоритмов (Graham [192]), модифицированном в работе Andrew [17]. Это лишь один из многих алгоритмов порядка $O(n \log n)$, предложенных для решения этой задачи. В работе Preparata, Hong [322] описан метод «разделяй и властвуй». Существует также инкрементный метод, который вставляет точки по одной за время $O(\log n)$ на одну вставку [321]. Овермарс и ван Леувен обобщили этот метод, так что точки можно вставлять и удалять за время $O(\log^2 n)$ [305]. Другие результаты о динамических выпуклых оболочках были получены в работах Hershberger, Suri [211], Chan [83] и Brodal, Jacob [73].

Хотя для этой задачи известна нижняя граница – $\Omega(n \log n)$ [393], многие авторы пытались улучшить этот результат. Это имеет смысл, поскольку во многих приложениях число точек, принадлежащих выпуклой оболочке сравнительно мало, а в оценке нижней границе предполагается, что почти все точки лежат на выпуклой оболочке. Поэтому полезно поискать алгоритмы, время работы которых зависит от сложности выпуклой оболочки. В работе Jarvis [221] предложен алгоритм «заворачивания», часто называемый обходом Джарвиса, который вычисляет выпуклую оболочку за время $O(h \cdot n)$, где h – сложность выпуклой оболочки. Такую же производительность в худшем случае дает алгоритм Овермарса и ван Леувена [303], основанный на более ранних работах Bykat [79], Eddy [156] и Green, Silverman [193]. Достоинством этого алгоритма является то, что ожидаемое время работы линейно для многих распределений точек. Наконец, в работе Kirkpatrick, Seidel [238] этот результат улучшен до $O(n \log h)$, а недавно Чан [82] открыл гораздо более простой алгоритм с такой же асимптотикой.

Выпуклую оболочку можно определить в пространстве любого числа измерений. Как мы увидим в главе 11, в трехмерном пространстве выпуклую оболочку по-прежнему можно вычислить за время $O(n \log n)$. Но если размерность больше 3, то зависимость сложности выпуклой оболочки от числа точек перестает быть линейной. Подробности см. в замечаниях к главе 11.

За прошедшие годы было предложено несколько общих методов обработки особых случаев. Такие схемы символического возмущения немного изменяют входные данные, так чтобы вырожденность пропала. Однако возмущение производится лишь символически. Эта техника была впервые описана в работе Edelsbrunner, Mücke [164], а затем усовершенствована в работах Yap [397] и Emiris, Canny [172, 171]. Метод символического возмущения освобождает программиста от заботы о вырожденных случаях, но имеет ряд недостатков: применение библиотеки символического возмущения замедляет работу алгоритма, и иногда приходится восстанавливать «настоящий результат» по «возмущенному результату», что не всегда легко. Из-за этих недостатков в работе Burnikel et al. [78] утверждается, что проще (с точки зрения затрат труда программиста) и эффективнее (с точки зрения времени работы) разбираться с вырожденными случаями вручную.

Тема устойчивости геометрических алгоритмов в последнее время вызывает пристальный интерес. Большинство геометрических сравнений можно описать как вычисление знака некоторого определителя. Один из способов борьбы с неточностью арифметики с плавающей точкой при вычислении знака – выбрать небольшое пороговое значение ε и сказать, что определитель равен нулю, если результат вычисления меньше ε. При наивной реализации это может приводить к противоречиям (например, для трех точек a, b, c может получиться, что $a = b$, $b = c$, но $a \neq c$), из-за которых программа завершится аварийно. В работе Guibas et al. [198] показано, что комбинирование этого подхода с интервальной арифметикой и обратным анализом ошибок может дать устойчивый алгоритм. Другой вариант – использование точной арифметики. В этом случае мы вычисляем столько разрядов определителя, сколько необходимо для определения его знака. Это замедляет вычисления, но были разработаны методы, позволяющие свести замедление к минимуму [182, 256, 395]. Помимо общих подходов, во многих статьях описываются устойчивые методы решения конкретных задач [34, 37, 81, 145, 180, 181, 219, 279].

Мы привели краткий обзор областей применения, из которых черпали примеры, чтобы показать, для чего нужны различные геометрические понятия и алгоритмы, описанные в книге. Ниже перечислены ссылки на некоторые учебники, из которых можно почерпнуть дополнительные сведения об этих предметных областях. Разумеется, по каждой из них есть гораздо больше хороших книг, чем мы смогли упомянуть.

По компьютерной графике есть масса книг. Книга Foley et al. [179] очень подробна и обычно считается одним из лучших источников по этой теме. Другие хорошие книги: Shirley et al. [359] и Watt [381]. Подробный обзор на тему робототехники и задачи планирования движения можно найти в книге Choset et al. [127], а также в более старых книгах Latombe [243] и Hopcroft, Schwartz, Sharir [217].

Дополнительные сведения о геометрических аспектах робототехники есть в книге Selig [348].

Существует немало книг о геоинформационных системах, но в большинстве из них алгоритмическим вопросам уделяется немного внимания. Отметим несколько учебников общего типа: DeMers [140], Longley et al. [257], Worboys, Duckham [392]. Структуры для представления пространственных данных подробно описаны в книге Samet [335].

Книги Faux, Pratt [175], Mortenson [285] и Hoffmann [216] – хорошие вводные учебники по САПР/АСУП и геометрическому моделированию.

1.5. Упражнения

1.1. Выпуклая оболочка множества S определяется как пересечение всех выпуклых множеств, содержащих S. Мы отмечали также, что выпуклая оболочка является выпуклым множеством с наименьшим периметром. Требуется доказать, что эти определения эквивалентны.

a. Докажите, что пересечение двух выпуклых множеств выпукло. Отсюда следует, что пересечение любого конечного семейства выпуклых множеств выпукло.

b. Докажите, что многоугольник $\mathcal{P}$ с наименьшим периметром, содержащий множество точек P, является выпуклым множеством.

c. Докажите, что любое выпуклое множество, содержащее множество точек P, содержит и многоугольник $\mathcal{P}$ с наименьшим периметром.

1.2. Пусть P – множество точек на плоскости. Обозначим $\mathcal{P}$ выпуклый многоугольник, вершинами которого являются точки из P и который содержит все точки P. Докажите, что такой многоугольник определен однозначно и что он является пересечением всех выпуклых множеств, содержащих P.

1.3. Пусть E – неотсортированное множество n отрезков, являющихся сторонами выпуклого многоугольника. Опишите алгоритм сложности $O(n \log n)$, который вычисляет по E список всех вершин многоугольника, отсортированный в порядке обхода по часовой стрелке.

1.4. В алгоритме вычисления выпуклой оболочки мы должны уметь проверять, лежит ли точка r слева или справа от направленной прямой, проходящей через точки p и q. Обозначим $p = (p_x, p_y)$, $q = (q_x, q_y)$, $r = (r_x, r_y)$.

a. Докажите, что знак определителя

$$D = \begin{vmatrix} 1 & p_x & p_y \\ 1 & q_x & q_y \\ 1 & r_x & r_y \end{vmatrix}$$

определяет, лежит ли r слева или справа от прямой.

b. Докажите, что $|D|$ равен удвоенной площади треугольника с вершинами в точках p, q, r.

c. Почему этот способ удобен для реализации основной проверки в алгоритме ConvexHull? Приведите аргументы для случаев, когда координатами являются целые числа и числа с плавающей точкой.

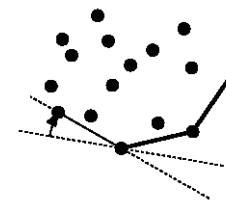
1.5. Проверьте, что алгоритм ConvexHull с указанными выше модификациями правильно вычисляет выпуклую оболочку даже вырожденного множества точек. Рассмотрите, к примеру, случай, когда все точки лежат на одной вертикальной прямой.

1.6. Во многих ситуациях требуется вычислять выпуклую оболочку множества объектов, отличных от точек.

a. Пусть S – множество n отрезков прямых на плоскости. Докажите, что выпуклая оболочка S в точности совпадает с выпуклой оболочкой множества, состоящего из $2n$ концевых точек отрезков.

b.* Пусть $\mathcal{P}$ – невыпуклый многоугольник. Опишите алгоритм, который вычисляет выпуклую оболочку $\mathcal{P}$ за время $O(n)$. *Подсказка*: воспользуйтесь вариантом алгоритма ConvexHull, в котором вершины рассматриваются не в лексикографическом, а в другом порядке.

1.7. Рассмотрим альтернативный подход к вычислению выпуклой оболочки множества точек на плоскости. Начнем с самой правой точки, это будет первая точка p_1 выпуклой оболочки. Теперь представим себе вертикальную прямую и будем вращать ее по часовой стрелке, пока не встретим некоторую точку p_2. Это будет вторая точка выпуклой оболочки. Продолжим вращать прямую, но теперь относительно p_2, пока не встретится точка p_3. И так до тех пор, пока снова не наткнемся на p_1.



a. Запишите этот алгоритм на псевдокоде.

b. Какие вырожденные случаи могут встретиться и как их обрабатывать?

c. Докажите, что этот алгоритм правильно вычисляет выпуклую оболочку.

d. Докажите, что этот алгоритм можно реализовать за время $O(n \cdot h)$, где h – сложность выпуклой оболочки.

e. Какие проблемы могут возникнуть при вычислениях с плавающей точкой?

1.8. Алгоритм сложности $O(n \log n)$ для вычисления выпуклой оболочки множества n точек на плоскости, описанный в этой главе, основан на идее инкрементного построения: добавлять точки по одной и после каждого добавления пересчитывать оболочку. В этом упражнении мы разработаем алгоритм, основанный на другой идее: разделяй и властвуй.

a. Пусть $\mathcal{P}_1$ и $\mathcal{P}_2$ – непересекающиеся выпуклые многоугольники, в совокупности имеющие n вершин. Придумайте алгоритм, который вычисляет выпуклую оболочку $\mathcal{P}_1 \cup \mathcal{P}_2$ за время $O(n)$.

b. Примените алгоритм из части a для разработки алгоритма типа «разделяй и властвуй», вычисляющего выпуклую оболочку множества n точек на плоскости за время $O(n \log n)$.

1.9. Предположим, что уже имеется подпрограмма ConvexHull для вычисления выпуклой оболочки множества точек на плоскости. На выходе она порождает список вершин выпуклой оболочки в порядке обхода по часовой стрелке. Пусть теперь $S = \{x_1, x_2, ..., x_n\}$ – множество n чисел. Покажите, что S можно отсортировать за время $O(n)$ плюс время, необходимое для одного вызова ConvexHull. Поскольку нижняя граница сложности алгоритма сортировки асимптотически равна $\Omega(n \log n)$, то отсюда следует, что и нижняя граница сложности алгоритма вычисления выпуклой оболочки – тоже $\Omega(n \log n)$. Поэтому алгоритм, описанный в этой главе, асимптотически оптимален.

1.10. Пусть S – множество n единичных окружностей на плоскости (возможно, пересекающихся). Мы хотим вычислить выпуклую оболочку S.

a. Покажите, что граница выпуклой оболочки S состоит из отрезков прямых и дуг окружностей из S.
b. Покажите, что ни одна окружность не может встречаться в границе выпуклой оболочки более одного раза.
c. Обозначим S' множество центров окружностей, принадлежащих S. Покажите, что окружность из S встречается в выпуклой оболочке тогда и только тогда, когда ее центр принадлежит выпуклой оболочке S'.
d. Придумайте алгоритм сложности $O(n \log n)$ для вычисления выпуклой оболочки S.
e.* Придумайте алгоритм сложности $O(n \log n)$ для случая, когда окружности, принадлежащие S, имеют разные радиусы.