Содержание

Об авторе

Нинчуань Сяо – доцент географического факультета Университета штата Огайо. Он читал разнообразные курсы по картографии, ГИС и пространственному анализу и моделированию. С 2009 по 2012 год занимал пост председателя специальной группы по пространственному анализу и моделированию Ассоциации американских географов. Исследования д-ра Сяо посвящены разработке действенных и эффективных вычислительных методов построения карт и анализа пространственно-временных данных в различных предметных областях, в т. ч. пространственной оптимизации, систем поддержки принятия решений о пространственном размещении, моделировании окружающей среды и экологической обстановки, пространственных моделях распространения эпидемий. Его работы публиковались в ведущих журналах по географии и ГИС. Его текущие проекты связаны с проектированием и реализацией новаторских подходов к анализу и нанесению на карту больших данных из социальных сетей и других онлайновых ресурсов, а также с разработкой поисковых алгоритмов для решения задач пространственного агрегирования. Он также работает в составе междисциплинарных групп над проектами отображения на картах влияния мобильности населения на распространение инфекционных заболеваний и построения моделей влияния окружающей среды на социальную динамику в Северном Камеруне.

Предисловие

Географические информационные системы (геоинформационные системы, ГИС) приобретают все большее значение, помогая нам понять сложную социальную, экономическую и природную динамику в ситуациях, где ключевую роль играют пространственные компоненты. В сравнительно короткой истории развития теории и приложений ГИС часто можно наблюдать процесс отчуждения ГИС как «механизма» от ее пользователей. В некоторых случаях ГИС свелась к черному ящику, который используется для порождения приятных глазу карт для различных целей. Эта тенденция уже проникла в наши учебные программы, в которых значительная доля времени уделяется тому, как научить студентов пользоваться графическими интерфейсами программных пакетов ГИС. Преодоление этой тенденции представляет серьезный вызов для исследователей и преподавателей ГИС.

В данной книге мы будем говорить о важнейших алгоритмах ГИС, являющихся фундаментом многих операций над пространственными данными. Научная дисциплина ГИС всегда была весьма разветвленной, и во многих учебниках общие вопросы рассматриваются поверхностно, лишая студентов возможности полностью вникнуть в концептуальные основы ГИС. Алгоритмы ГИС часто представляют разными способами с использованием различных структур данных, а отсутствие единого представления затрудняет понимание сути алгоритмов. Студенты, посещающие курсы ГИС, специализируются в разных областях знания и не всегда хорошо знакомы с терминами, используемыми при традиционном формальном описании алгоритмов. Из-за этого преподавание алгоритмов на курсах ГИС стало больной темой. Но это не должно служить оправданием исключению алгоритмов из курса. Проследив, как пространственные данные подаются на вход алгоритма и как алгоритм используется для обработки данных и получения конечного результата, мы сможем гораздо лучше понять два важных аспекта ГИС: что на самом деле представляют собой геопространственные данные и как эти данные в действительности обрабатываются.

В этой книге рассматриваются алгоритмы, критические для реализации некоторых базовых функций ГИС. Однако наша цель не в том, чтобы предъявить исчерпывающий длинный перечень алгоритмов. Мы включили только те алгоритмы, которые используются в большинстве современных ГИС или оказали существенное влияние на разработку текущих алгоритмов; поэтому выбор тем может показаться субъективным. Мы исповедуем минималистский взгляд на геопространственные данные, считая их данными о местоположении в пространстве, т. е. фокусируем внимание на координатах как атомарной единице географической информации, допуская, что в большинстве своем геопространственные данные можно рассматривать как наборы точек. Это позволяет в значительной степени избежать ненужных дискуссий о различии векторного и растрового представлений, сведя обсуждение к фундаментальной модели данных. Начав с этого места, мы приступим к изучению многообразных алгоритмов ГИС, которые помогают в выполнении базовых функций, как то: измерение важных пространственных свойств, например расстояния,

включение нескольких источников данных с помощью слоев, ускорение анализа за счет применения различных методов индексирования. Мы также уделим внимание алгоритмам решения таких задач пространственного анализа и моделирования, как интерполяция, анализ паттернов и принятие решений с помощью моделей оптимизации. Разумеется, все эти функции присутствуют во многих пакетах ГИС, как коммерческих, так и с открытым кодом. Однако наша цель – не дублировать то, что там есть, а показать, как это работает, чтобы мы могли реализовать собственную ГИС или, по крайней мере, некоторые ее функции, не полагаясь на программную систему, в названии которой встречается акроним ГИС. Чтобы обрести такую свободу, мы должны опуститься на уровень, где данные видны, а не скрыты, где процессы представлены в виде кода, а не кнопок, а выходные данные так же прозрачны, как входные.

Это не традиционная книга об алгоритмах. В типичной книге по информатике алгоритмы были бы представлены в виде псевдокода, в котором отражены наиболее важные части, а некоторые детали опущены. Но такой подход не годится для многих студентов, изучающих ГИС, поскольку зачастую теоретические аспекты алгоритмов интересуют их не в первую очередь. Стремление понять алгоритмы ГИС обычно проистекает из желания узнать, «как это работает». Поэтому для описания и реализации алгоритмов мы используем реальный работающий код. В качестве языка мы выбрали Python за его простоту, что позволяет не тратить много времени на изучение программирования как такового. Мы старались не слишком увлекаться мощными «пакетными» модулями Python, а показать, как на самом деле устроены алгоритмы. С той же целью мы с самого начала используем единое представление геопространственных данных и, стало быть, единые структуры данных.

Эта книга не появилась бы без помощи других людей. Я глубоко благодарен сообществу ПО с открытым исходным кодом, которое очень сильно поспособствовало формированию наших представлений о пространственных данных и их использовании, так что в этом смысле книга не состоялась бы без программ с открытым исходным кодом. По возникающим у меня вопросам касательно Python я неоднократно просил совета у пользователей сайта Stack Overflow. Онлайновое сообщество LATeX (http://tex.stackexchange.com и http://en.wikibooks.org/wiki/LaTeX) всегда было готово ответить на вопросы по поводу верстки в бессонные ночи, проведенные в работе над книгой. Некоторые открытые пакеты были особенно важны во многих частях книги, в т. ч. связанные с *k*D-деревьями (http://en.wikipedia.org/wiki/K-d_tree и https://code.google.com/p/python-kdtree/) и кригингом (https://github.com/cjohnson318/geostatsmodels). Выражаю также благодарность студентам многочисленных курсов, прочитанных мною в последние годы в США и Китае. Их отзывы, а иногда и критические замечания позволили мне улучшить реализации многих алгоритмов. Спасибо Мей-По Куань за поддержку и Ричарду Ли, Кэтрин Хоу и Мэттью Олдфилду за внимательное прочтение рукописи. Подробные замечания ряда рецензентов ранних вариантов книги существенно помогли мне улучшить текст. Наконец, я благодарен своей семье за терпение и поддержку на протяжении всего процесса работы над книгой.

Нинчуань Сяо
−82.8650°, 40.0556°
декабрь 2014

От издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге, – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв на нашем сайте www.dmkpress.com, зайдя на страницу книги и оставив комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу dmkpress@gmail.com; при этом укажите название книги в теме письма.

Если вы являетесь экспертом в какой-либо области и заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу http://dmkpress.com/authors/publish_book/ или напишите в издательство по адресу dmkpress@gmail.com.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы обеспечить высокое качество наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг, мы будем очень благодарны, если вы сообщите о ней главному редактору по адресу dmkpress@gmail.com. Сделав это, вы избавите других читателей от недопонимания и поможете нам улучшить последующие издания этой книги.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательства «ДМК Пресс» и SAGE очень серьезно относятся к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконной публикацией какой-либо из наших книг, пожалуйста, пришлите нам ссылку на интернет-ресурс, чтобы мы могли применить санкции.

Ссылку на подозрительные материалы можно прислать по адресу электронной почты dmkpress@gmail.com.

Мы высоко ценим любую помощь по защите наших авторов, благодаря которой мы можем предоставлять вам качественные материалы.

Глава 1

Введение

Алгоритмы[1] проектируются для решения вычислительных задач. Вообще говоря, алгоритм – это процесс, состоящий из ряда четко определенных шагов. Например, чтобы вычислить сумму 18 и 19, нужно решить, как быть с тем, что 8 плюс 9 больше 10, хотя в разных культурах этот факт обрабатывается поразному. Даже в такой простой задаче мы ожидаем, что используемые шаги позволят быстро дать правильный ответ. Есть много задач, более трудных, чем сложение, и для их эффективного и правильного решения нужно проектировать шаги вычислений более тщательно.

При разработке ГИС и их приложений алгоритмы важны чуть ли не в каждой детали. Например, щелкая мышью по карте, мы ожидаем получить от компьютера быстрый ответ, содержащий информацию о точке или области, в которой находится курсор мыши. В этой процедуре, встречающейся практически в любом приложении ГИС, участвует несколько алгоритмов, гарантирующих получение приемлемого ответа. Все начинается с поиска объекта (точки, прямой, многоугольника или пикселя), на который пришелся щелчок. Эффективный алгоритм поиска позволяет быстро сузить интересующую нас область. Попытка решить задачу в лоб означала бы проверку каждого объекта, что в случае большого набора данных сделало бы поиск недопустимо долгим. Для решения этой проблемы придумано много алгоритмов индексирования и опроса пространственных данных. В процессе поиска мы должны проверять, соответствует ли объект данных точке на экране. В случае многоугольников требуется решить, находится ли точка внутри многоугольника, для чего нужен специальный алгоритм, который быстро отвечает «да» или «нет» на этот вопрос. Обычно геопространственные данные поступают из многих источников, и принято приводить их к единой системе координат, чтобы разные наборы данных можно было обрабатывать единообразно. Еще одно типичное следствие наличия нескольких источников данных – организация слоев, позволяющих удобно использовать разнородную информацию.

Исследовать алгоритм можно с разных точек зрения. Очевидно, что алгоритм должен решать задачу правильно. Правильность некоторых алгоритмов доказать легко. Например, ниже в этой главе мы познакомимся с двумя алгоритмами поиска, правильность которых не вызывает сомнений. Другие

---

[1] Само слово «алгоритм» происходит от латинского *algorismus*, которое, в свою очередь, происходит от имени персидского математика, астронома и географа *Аль-Хорезми*, внесшего большой вклад в алгебру и географическую картину мира.

алгоритмы не столь очевидны, для доказательства их правильности необходим формальный анализ. Второе свойство алгоритмов – эффективность, или время работы. Конечно, мы всегда хотим, чтобы алгоритм работал быстро, но существуют теоретические пределы быстродействию алгоритма, зависящие от задачи. Мы будем обсуждать некоторые проблемы такого рода в конце книги при рассмотрении пространственной оптимизации. Помимо правильности и времени работы, при рассмотрении алгоритмов зачастую следует принимать во внимание организацию данных и конкретную реализацию.

1.1. Вычислительные аспекты алгоритмов

Пусть имеется неупорядоченный список n точек. Мы хотим найти в этом списке конкретную точку. Сколько времени для этого понадобится? Это разумный вопрос. Но на фактическое *время* влияет множество разных факторов: выбор языка программирования, квалификация программиста, платформа, количество и быстродействие процессоров и т. д. Более полезный способ оценки времени – количество *шагов*, необходимых для завершения работы, и анализ общей стоимости алгоритма в терминах количества шагов. Конечно, стоимость каждого шага – величина переменная, зависящая от того, что понимается под шагом. Но все это равно более надежный способ описания времени вычислений, потому что можно выделить ряд шагов – например, простые арифметические операции, вычисление логических выражений, доступ к памяти компьютера для выборки данных, присваивание значения переменной, – стоимость которых постоянна. Если мы сможем посчитать, сколько шагов необходимо для выполнения некоторой процедуры, то будем иметь неплохое представление о том, сколько времени эта процедура займет, что особенно полезно при сравнении алгоритмов.

Но вернемся к нашему списку точек. Если точки могут храниться в произвольном порядке, то лучшее, что можно сделать, – перебирать их одну за другой, пока не найдем искомую точку или не дойдем до конца списка. Предположим, что список называется `points`, и мы хотим узнать, включает ли он точку `p0`. Для выполнения поиска можно воспользоваться следующим простым алгоритмом (листинг 1.1).

Листинг 1.1 ❖ Линейный поиск точки `p0` в списке

```
1 для каждой точки p в points:
2     если p совпадает с p0:
3         вернуть p и остановиться
```

Алгоритм в листинге 1.1 называется линейным поиском; мы просто перебираем все точки для поиска нужной информации. Сколько шагов придется сделать? Первая строка – заголовок цикла, который будет выполнен n раз, если искомая точка окажется последней в списке. Стоимость одной итерации цикла постоянна, поскольку список хранится в памяти компьютера, а основные операции в данном случае – доступ к информации по фиксированному адресу в памяти и переход к следующему адресу. Предположим, что стои-

мость этой операции равна c_1 и что мы выполняем ее в цикле не более n раз. Вторая строка – логическое сравнение двух точек. Она тоже выполняется не более n раз, потому что находится внутри цикла. Пусть стоимость сравнения тоже постоянна и равна c_2. В строке 3 просто возвращается значение найденной точки; стоимость этой операции равна c, и выполняется она только один раз. В лучшем случае мы найдем искомую точку уже на первой итерации цикла, поэтому полная стоимость будет равна $c_1 + c_2 + c$, или, упрощая, константе $b + c$. Но в худшем случае нам придется просмотреть все элементы от начала до конца, поэтому полная стоимость будет равна $c_1n + c_2n + c$, или $bn + c$, где b и c – константы, а n – длина списка (размер задачи). В среднем, если список представляет собой случайный набор точек и мы ищем в нем случайную точку, ожидаемая стоимость составит $c_1n/2 + c_2n/2 + c$, или $b'n + c$, и мы знаем, что $b' < b$, т. е. стоимость ниже, чем в худшем случае.

Насколько нам интересны фактические значения b, b' и c в этом анализе? Как они влияют на общую стоимость вычислений? Не слишком сильно, поскольку это константы. Но если складывать их много раз, то влияние будет существенным, а количество сложений в общем случае зависит от размера задачи n. Когда n достигает определенного уровня, влияние самих констант оказывается минимальным, а *рост* полной стоимости вычислений определяется величиной n.

Стоимость некоторых алгоритмов пропорциональна n^2, что сильно отличает их от алгоритмов со стоимостью, пропорциональной n. Например, в листинге 1.2 приведена простая процедура для вычисления наименьшего расстояния между парами точек в списке, содержащем n точек. Здесь первый цикл (строка 2) выполняется n раз, и стоимость каждой его итерации равна t_1, а второй цикл (строка 3) – n^2 раз с той же стоимостью итерации. Код сравнения (строка 4) выполняется n^2 раз, стоимость одного сравнения обозначим t_2. Очевидно, что вычисление расстояния (строка 5) обходится дороже остальных, более простых операций, но все равно постоянно, поскольку входные данные фиксированы, а для выполнения вычислений нужно конечное и не слишком большое количество шагов. Будем считать, что стоимость вычисления одного расстояния равна константе t_3. Поскольку расстояние между точкой и ей самой не вычисляется, то код вычисления расстояния будет выполняться $n^2 - n$ раз, как и сравнение в строке 6 (стоимость которого обозначим t_4). Стоимость присваивания в строке 7 постоянна и равна t_5, а выполняться оно может не более $n^2 - n$ в худшем случае, когда каждое следующее расстояние меньше предыдущего. Последняя строка выполняется только один раз, ее стоимость обозначим c. В итоге полное время работы этого алгоритма равно $t_1n + t_1n^2 + t_2n^2 + t_3(n^2 - n) + t_4(n^2 - n) + t_5(n^2 - n) + c$, или, упрощая, $an^2 + bn + c$. Теперь понятно, что время работы данного алгоритма определяется величиной n^2.

Листинг 1.2 ❖ Линейный поиск для нахождения наименьшего расстояния между парами точек в списке

```
1 пусть mindist - очень большое число
2 для каждой точки p1 в points:
3     для каждой точки p2 в points:
```

```
4         если p1 не совпадает с p2:
5             пусть d – расстояние между p1 и p2
6             если d < mindist:
7                 mindist = d
8 вернуть mindist и остановиться
```

В обоих рассмотренных примерах величина n определяет полную стоимость вычислений, и мы говорим, что стоимость алгоритма линейного поиска имеет порядок n, а стоимость алгоритма нахождения минимального расстояния – порядок n^2. В случае линейного поиска мы также знаем, что при увеличении n полная стоимость ограничена сверху величиной bn. А что можно сказать насчет нижней границы? Мы знаем, что в лучшем случае время выполнения постоянно, т. е. имеет порядок n^0, но в общем случае это не так. Если мы можем найти верхнюю, но не нижнюю границу времени выполнения, то пользуемся нотацией O для обозначения порядка. В нашем примере порядок равен $O(n)$ как в среднем, так и в худшем случае (поскольку не константы определяют полную стоимость). Говорят также, что время работы, или временная сложность алгоритма линейного поиска, равно $O(n)$. Поскольку нотация O относится к верхней границе, т. е. к худшему случаю, имеется в виду временная сложность также в худшем случае.

Существуют алгоритмы, для которых время работы не ограничено сверху. Но нам известна нижняя граница, и тогда используется нотация Ω. Когда говорят, что время работы равно $\Omega(n)$, это значит, что временная сложность алгоритма по порядку величины никак не меньше n, хотя верхняя граница неизвестна. Бывают также алгоритмы, для которых известны и верхняя, и нижняя границы времени работы, тогда используется нотация Θ. Например, время работы $\Theta(n^2)$ означает, что алгоритм в любом случае занимает время порядка n^2. Так обстоит дело для алгоритма нахождения наименьшего расстояния, поскольку всегда выполняется n^2 итераций цикла вне зависимости от результата сравнения в строке 6. Будет точнее сказать, что временная сложность равна $\Theta(n^2)$, а не $O(n^2)$, потому что нам известно, что ее нижняя граница также имеет порядок n^2.

Теперь организуем точки, хранящиеся в списке, в виде дерева, как показано на рис. 1.1. Это двоичное дерево, потому что из каждого узла, начиная с корня, может исходить не более двух ветвей. Здесь в корне дерева хранится точка (6, 7), она показана сверху. Все точки, для которых координата X меньше или равна, чем у точки в корне, хранятся в узлах, находящихся слева от корня, а точки, для которых координата X больше, чем у точки в корне, – в узлах справа от корня. На втором уровне мы видим точки (4, 6) и (9, 4). Для каждой из них слева находятся точки с меньшей или равной координатой Y, а справа – с большей. Спускаясь вниз по дереву, мы попеременно используем координаты X и Y, пока не найдем нужную нам точку или не дойдем до конца ветви (листового узла).

Чтобы воспользоваться этой древовидной структурой для поиска точки, мы начинаем с корня (поиск в дереве всегда начинается с корня) и спускаемся вниз, решая на каждом уровне, по какой ветви идти. Например, чтобы найти точку (1, 7), мы пойдем из корня по левой ветви, потому что координата X искомой точки равна 1, т. е. меньше, чем в корне. Затем мы пойдем из узла

(4, 6) по правой ветви, потому что координата *Y* (7) меньше, чем хранится в текущем узле. Мы пришли в узел (3, 7) на третьем уровне дереве, и из него пойдем по левой ветви, потому что координата *X* искомой точки меньше, чем в узле. Дойдя до узла (2, 9), мы пойдем налево, потому что координата *Y* искомой точки меньше, чем в узле. Резюмируем: зная дерево, мы можем написать алгоритм поиска в нем; он показан в листинге 1.3.

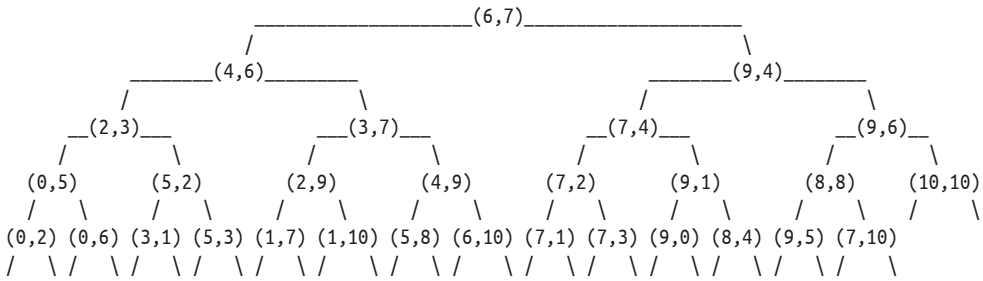

Рис. 1.1 ❖ Дерево, в котором хранится 29 случайно выбранных точек. Каждый узел помечен координатами *X* и *Y*, изменяющимися в диапазоне от 0 до 10

Листинг 1.3 ❖ Двоичный поиск точки `p0` в дереве

```
 1 пусть t – корень дерева
 2 пока t не пусто:
 3   пусть p – точка в узле t
 4   если p совпадает с p0:
 5       вернуть p и остановиться
 6   если t расположен на том уже уровне, что p0:
 7       coordp, coordp0 = координаты X точек p и p0
 8   иначе:
 9       coordp, coordp0 = координаты Y точек p и p0
10   если coordp0 <= coordp:
11       t = левая ветвь t
12   иначе:
13       t = правая ветвь t
```

Эта процедура называется двоичным поиском по дереву. Вспоминая наше обсуждение времени работы, легко видеть, что время работы этого алгоритма определяется количеством итераций цикла `пока` (строка 2), которое зависит от высоты дерева, т. е. числом ребер на пути от корня до самого удаленного листового узла. Высота показанного выше дерева равна 4, оно может содержать до 31 узла (в нашем примере их всего 29). Вообще, в двоичном дереве высоты H можно сохранить до $2^0 + 2^1 + 2^2 + \ldots + 2^H = 2^{H+1} - 1$ элементов данных. Иными словами, если имеется n точек, размещенных во всех узлах идеально сбалансированного двоичного дерева, в котором все листовые узлы находятся на одном и том же уровне, то $2^{H+1} - 1 = n$ и, следовательно, $H = \log_2(n + 1) - 1$. Если задано такое дерево, то время работы алгоритма имеет порядок $\log_2(n + 1)$. Поскольку по порядку величины – это то же самое, что $\log_2 n$, мы говорим, что временная сложность равна $O(\log_2 n)$. Для сбалансированного, но не идеально сбалансированного двоичного дерева, в котором раз-

ность высот листовых узлов не больше 1, временная сложность по-прежнему равна $O(\log_2 n)$, т. е. самый дальний листовой узел имеет высоту H. Но если сбалансированность гарантировать невозможно, то все становится хуже. На рис. 1.2 приведен пример несбалансированного дерева, в котором хранятся те же точки, но высота равна 8. Итак, мы знаем, что алгоритм двоичного поиска эффективнее, чем линейный поиск, потому что $O(\log_2 n) < O(n)$, но фактическое время работы зависит от того, как построено дерево, и может быть больше $O(\log_2 n)$, если дерево не сбалансировано.

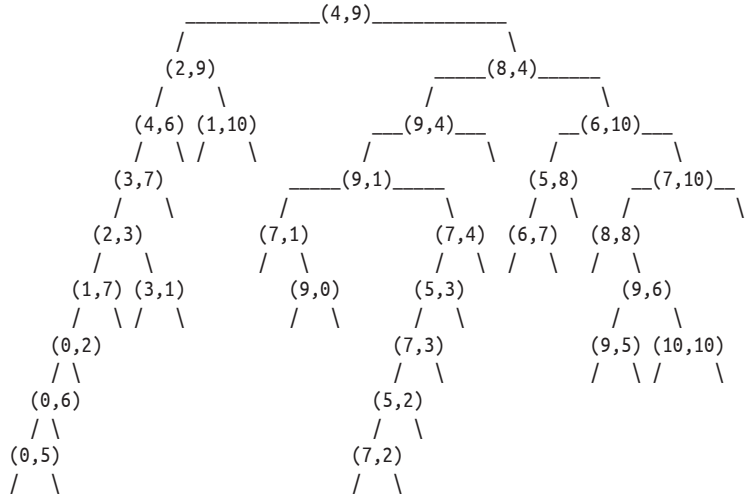

Рис. 1.2 ❖ Несбалансированное дерево, в котором хранится 29 случайных точек

Поскольку хранение точек в сбалансированном двоичном дереве может заметно повысить эффективность поиска, то не поможет ли оно заодно и сократить время вычисления кратчайшего расстояния между двумя точками? Временная сложность рассмотренного выше решения путем полного перебора равна $\Theta(n^2)$, она быстро возрастает с увеличением n. Было бы разумно поискать более эффективный способ нахождения наименьшего из попарных расстояний между точками в списке. Такое решение действительно существует, и ключ к нему – использование древовидных структур. Мы вернемся к этой теме во второй части книги, когда будем обсуждать различные древовидные структуры более подробно. В этой книге мы не будем уделять много внимания теоретическому анализу сложности алгоритмов, а вместо этого ограничимся эмпирическим анализом, выполняя алгоритмы с разными наборами данных.

Разговор о поиске, в особенности о двоичном поиске по дереву, логически подводит нас к вопросу о структуре данных: как хранить и организовывать данные, чтобы повысить эффективность алгоритма? Структура дерева – хороший пример того, как данные, первоначально представленные в виде списка, можно реорганизовать для увеличения производительности поиска. Многие структуры данных зависят от конкретной задачи. Некоторые струк-

туры довольно сложны, но увеличение объема занятой данными памяти часто компенсируется уменьшением времени работы.

1.2. Кодирование

Алгоритмы можно описывать разными способами. В этой главе мы описывали алгоритмы линейного и двоичного поиска словесно. В теоретической работе достаточно формального описания, в котором детально специфицированы все шаги, но исполнимость не является обязательным условием. Такое описание называется псевдокодом, потому что это не настоящий компьютерный код, хотя и очень близко к нему. В данной книге мы избрали более практичный путь – описывать алгоритм на реальном языке программирования, в качестве какового решили взять Python.

У составления компьютерной программы (т. е. кодирования) для описания алгоритма имеется важное преимущество: любой алгоритм можно сразу же выполнить. Поэтому все относящееся к работе алгоритма представлено в самой книге в виде простого текста. Код становится частью текста, и, следовательно, открывается возможность экспериментировать с ним, модифицировать и улучшать. Однако при таком подходе информации может оказаться слишком много, особенно если язык программирования заставляет писать вспомогательный код, без которого программа не работает. Например, во многих языках требуется ставить специальные символы или скобки в конце строки, иначе компилятор считает программу синтаксически некорректной. Добавление этих символов затрудняет чтение и мешает сосредоточиться на основном содержании текста. Мы выбрали в этой книге Python в основном за его простой синтаксис, а также за изобилие популярных, эффективных и хорошо сопровождаемых модулей. Все приведенные в данной книге программы были протестированы для версии Python 2.7, которая была стабильной и широко распространенной на момент написания книги. В большинстве программ используются только базовые средства Python, поэтому велики шансы, что они будут совместимы и с последующими версиями Python.

1.3. Как использовать эту книгу

Основной текст книги разделен на три большие части. Идея в том, чтобы сначала обсудить самые фундаментальные аспекты данных – геометрические, а затем переходить к более сложным темам: индексированию пространственных данных, а также пространственному анализу и моделированию. В конце каждой главы имеется раздел «Примечания», в котором описываются основные работы, связанные с рассмотренным материалом. Мы также включили три приложения, стремясь помочь читателям разобраться в языке программирования Python и в структуре включенных в книгу программ.

Часть I посвящена описанию местоположения, а конкретно координатам, которые необходимы для представления геопространственной информа-

ции. В главе 2 мы рассмотрим несколько алгоритмов для вычисления разных видов расстояния, например расстояния между точками или от точки до прямой. Мы также обсудим вычисление центра тяжести многоугольника и широко распространенный алгоритм, который эффективно отвечает на вопрос, находится ли точка внутри многоугольника. Последняя тема главы 2 – преобразование систем координат, включая картографические проекции. В главе 3 рассматривается традиционная операция ГИС – наложение. Хотя эта тема «старая», фактическое вычисление результата наложения двух многоугольников может быть весьма трудоемким, пусть и не особенно сложным. Многие вопросы, обсуждаемые в этой части книги, связаны с дисциплиной, называемой вычислительной геометрией. Но нас будут интересовать в основном вещи, касающиеся ГИС.

Основной темой части II является идея индексирования пространственных данных. У пространственной информации есть свои особенности. Хотя основной подход к индексированию – разделяй и властвуй – остается в силе и для пространственных данных, из-за наличия двух (а в некоторых случаях и большего числа) измерений приходится проектировать более специализированные алгоритмы. В главе 4 мы введем основные понятия, связанные с индексированием, и сосредоточимся на разработке структуры дерева. Глава 5 посвящена *k*D-деревьям, которые обычно применяются для индексирования данных о точках. В главе 6 рассматривается популярный метод квадродеревьев, или деревьев квадрантов для индексирования точек и растровых данных. В главе 7 в обсуждение включаются прямые и многоугольники.

Часть III посвящена главной теме приложений ГИС: пространственному анализу и моделированию. Сначала в главе 8 мы рассмотрим интерполяцию координат точек и сравним два стандартных метода: обратных взвешенных расстояний и кригинг. Мы также включили алгоритм имитации данных под названием *смещение средней точки*, взятый из литературы по фрактальной геометрии. Глава 9 посвящена анализу пространственных паттернов, в т. ч. вычислению индекса *I* Морана. В главу 10 включены алгоритмы анализа сетей, в частности вычисления кратчайшего пути. Две главы мы посвятили пространственной оптимизации: в главе 11 рассматриваются точные методы, а в главе 12 – некоторые эвристические методы.

Мы также включили три приложения, содержащих технические детали, относящиеся к кодированию. Сразу хотим сказать, что хотя в основном это книга об алгоритмах, кодирование тоже занимает в ней важное место. Поэтому мы добавили краткое введение в язык Python. Далее следует столь же краткое введение в мощную библиотеку GDAL/OGR, точнее в ее интерфейс с Python, и в Python-библиотеку пространственного анализа PySAL. Наша цель – помочь читателю быстро начать работу с этими библиотеками и получить представление о том, как «реальные» наборы данных связаны с обсуждаемыми в книге вопросами (и, конечно, кодом).

Большинство программ, встречающихся в книге, хранятся в отдельных файлах, так что их можно использовать в других программах. В таком случае в заголовке листинга указывается имя файла. Для организации программ мы пользуемся каталогами. В последнем приложении перечислены все встречающиеся Python-программы и наборы данных.

Каждую главу можно было бы легко развернуть в отдельную книгу, где соответствующая тема рассматривается более полно. Таким образом, эту книгу можно рассматривать как обзор тематики алгоритмов ГИС. Лучший способ охватить представленные в книге вопросы во всей широте – кодирование. Сейчас ведется работа над коллективной страницей на Github[1], где читатели смогут делиться мыслями о реализации как включенных в книгу алгоритмов, так и новых алгоритмов, которые в книгу не вошли. Теоретический вывод не находится в фокусе нашего внимания, чего нельзя сказать об эмпирическом анализе. Мы включили в основной текст и в упражнения много экспериментов. Но это не значит, что вы не можете экспериментировать самостоятельно, особенно когда речь идет об инновационных направлениях. Книгу можно будет считать успешной, только если она достигнет двух целей: во-первых, на основе усвоенных алгоритмов и кода читатели смогут разрабатывать собственные инструменты, отвечающие особенностям наборов данных и требованиям приложений, а во-вторых, написание кода станет привычкой при работе с геопространственными данными.

[1] https://github.com/gisalgs.

Часть I

ГЕОМЕТРИЧЕСКИЕ АЛГОРИТМЫ

Глава 2

Базовые геометрические операции

> Представьте себе огромный лист бумаги, на котором Отрезки прямых, Треугольники, Квадраты, Пятиугольники, Шестиугольники и другие фигуры, вместо того чтобы неподвижно оставаться на своих местах, свободно перемещаются по всем направлениям вдоль поверхности, не будучи, однако, в силах ни приподняться над ней, ни опуститься под нее, подобно теням (только твердым и со светящимися краями), и вы получите весьма точное представление о моей стране и моих соотечественниках.
>
> *Эдвин Э. Эбботт «Флатландия. Роман о четвертом измерении»*

Эта главе посвящена основным алгоритмам ГИС, относящимся к геометрическим операциям. Сначала мы познакомимся с вычислением расстояния между двумя точками, а затем перейдем к объектам в пространстве большего числа измерений, в т. ч. алгоритму «точка внутри многоугольника», вычислению расстояния от точки до прямой, нахождению центроида и площади многоугольника. Мы также рассмотрим две картографические проекции и обсудим, как преобразовать геопространственные данные из одной системы координат в другую.

2.1. Точка

Прежде чем начать обсуждение, определим структуру данных для представления точки, которой будем пользоваться на протяжении всей книги. Это класс Python под названием `Point` (листинг 2.1). Нас интересует только двумерный случай, поэтому мы храним в классе лишь координаты *X* и *Y* точки. Мы переопределяем несколько встроенных методов Python, чтобы предоставить удобные средства. Метод `__getitem__` позволяет ссылаться на координаты *X* и *Y*, указывая соответственно индексы 0 и 1. Метод `__len__` возвращает количество измерений (в данном случае два). Мы также считаем

две точки одинаковыми, если их координаты X и Y совпадают (метод `__eq__`), и различными в противном случае (метод `__ne__`). Кроме того, нам нужен метод сравнения точек – точка в левом нижнем квадранте всегда считается «меньше» точки в правом верхнем квадранте. Точнее, если даны две точки p_1 и p_2, то мы считаем, что $p_1 < p_2$, если координата X точки p_1 меньше. Если координаты X двух точек совпадают, то меньшей считается точка, у которой меньше координата Y. Это правило закодировано в переопределенных операторах сравнения `__lt__` (меньше), `__gt__` (больше), `__le__` (меньше или равно) и `__ge__` (больше или равно). Мы также выводим координаты точки при ее печати (методы `__str__` и `__repr__`). Хотя в этой главе описанный способ упорядочения и сравнения не используется, он очень пригодится в последующих главах. Метод `distance` нужен для вычисления евклидова расстояния между двумя точками.

Листинг 2.1 ❖ Структура данных для представления точки (point.py)

```
from math import sqrt
class Point():
    """Класс для представления точки в декартовой системе координат."""
    def __init__(self, x=None, y=None):
        self.x, self.y = x, y
    def __getitem__(self, i):
        if i==0: return self.x
        if i==1: return self.y
        return None
    def __len__(self):
        return 2
    def __eq__(self, other):
        if isinstance(other, Point):
            return self.x==other.x and self.y==other.y
        return NotImplemented
    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return result
        return not result
    def __lt__(self, other):
        if isinstance(other, Point):
            if self.x<other.x:
                return True
            elif self.x==other.x and self.y<other.y:
                return True
            return False
        return NotImplemented
    def __gt__(self, other):
        if isinstance(other, Point):
            if self.x>other.x:
                return True
            elif self.x==other.x and self.y>other.y:
                return True
            return False
```

```
36          return NotImplemented
37      def __ge__(self, other):
38          if isinstance(other, Point):
39              if self > other or self == other:
40                  return True
41              else:
42                  return False
43              return False
44          return NotImplemented
45      def __le__(self, other):
46          if isinstance(other, Point):
47              if self < other or self == other:
48                  return True
49              else:
50                  return False
51              return False
52          return NotImplemented
53      def __str__(self):
54          if type(self.x) is int and type(self.y) is int:
55              return "({0},{1})".format(self.x,self.y)
56          else:
57              return "({0:.1f}, {1:.1f})".format(self.x,self.y)
58      def __repr__(self):
59          if type(self.x) is int and type(self.y) is int:
60              return "({0},{1})".format(self.x,self.y)
61          else:
62              return "({0:.1f}, {1:.1f})".format(self.x,self.y)
63      def distance(self, other):
64          return sqrt((self.x-other.x)**2 + (self.y-other.y)**2)
```

Класс `Point` легко расширить для более гибкого представления точек. Например, мы предполагаем, что X и Y – декартовы координаты (в этом предположении определен метод `distance`), но это ограничение можно ослабить, добавив в класс еще один член `CS`, который задает вид системы координат; тогда расстояние нужно будет вычислять соответственно. Можно создать подкласс, который наследует классу `Point` и определяет точки в пространстве более высокой размерности. Можно добавить в класс время и другие атрибуты.

2.2. Расстояние между двумя точками

Расстояние между двумя точками можно вычислить разными способами, зависящими от предметной области и системы координат, в которой представлены точки. Самый распространенный способ – измерение евклидова расстояния по прямой между двумя точками на декартовой плоскости по формуле

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2},$$

где x_1 и x_2 – абсциссы (X) двух точек, а y_1 и y_2 – их ординаты (Y). Метод `distance` класса `Point` возвращает именно евклидово расстояние между двумя точками (листинг 2.1).

В некоторых ситуациях евклидово расстояние – не самый подходящий способ измерения расстояния между двумя точками, особенно когда речь о городе, где перемещаться из одной точки в другую нужно только по улицам (рис. 2.1). В таком случае используется манхэттенское расстояние:

$$d_1 = |x_1 - x_2| + |y_1 - y_2|.$$

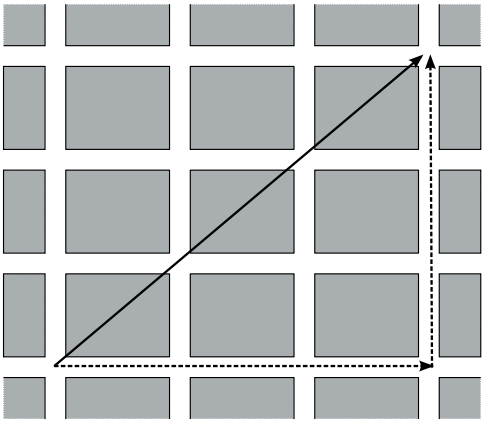

Рис. 2.1 ❖ Евклидово расстояние (сплошная линия) и манхэттенское расстояние (пунктирная линия). Серые прямоугольники представляют городские здания, между которыми проходят дороги

Если координаты точек измерены на поверхности сферы, то приведенные выше формулы не дают правильного расстояния между точками. В этом случае кратчайшее расстояние измеряется по дуге большой окружности, проходящей через две точки. Точнее, если положение точки определяется широтой, или углом с плоскостью экватора (φ), и долготой, или углом между меридианом, проходящим через точку, и нулевым меридианом (λ), то длина дуги большой окружности α (угол) между двумя точками определяется по формуле

$$\cos\alpha = \sin\varphi_1 \sin\varphi_2 + \cos\varphi_1 \cos\varphi_2 \cos\lambda,$$

где $d\lambda = |\lambda_1 - \lambda_2|$ – абсолютная величина разности между долготами двух точек. Но при вычислениях мы пользуемся следующей формулой гаверсинуса, чтобы обойти трудности при работе с отрицательными значениями:

$$a = \sin^2 \frac{d\varphi}{2} + \cos\varphi_1 \cos\varphi_2 \sin^2 \frac{d\lambda}{2},$$

$$c = 2\arcsin\min(1, \sqrt{a}),$$

где функция min возвращает меньшее из двух значений, чтобы избежать последствий ошибок округления, из-за которых a может оказаться больше 1; тогда расстояние можно вычислить по формуле

$$d = cR,$$

где R – радиус сферической Земли (3959 миль, или 6371 км). Важно помнить, что перед применением этой формулы широту и долготу нужно преобразовать в радианы.

Теперь напишем на Python программу для вычисления расстояния между двумя точками на сфере (листинг 2.2). Здесь широта и долгота выражены в градусах, и мы преобразуем их в радианы (строки 13–16). Пользуясь этим кодом, мы можем вычислить расстояние между Колумбусом, штат Огайо (40° с.ш., 83° з.д.) и Пекином (39.91° с.ш., 116.56° в.д.), оно равно 6780 миль (10 911 км).

Листинг 2.2 ❖ Вычисление расстояния по дуге большой окружности (spherical_distance.py)

```
 1 import math
 2 def spdist(lat1, lon1, lat2, lon2):
 3     """
 4     Вычисляет расстояние по дуге большой окружности, зная
 5     широту и долготу двух точек.
 6     Вход
 7       lat1, lon1: широта и долгота первой точки в градусах
 8       lat2, lon2: широта и долгота второй точки в градусах
 9     Выход
10       d: расстояние по дуге большой окружности
11     """
12     D = 3959                          # радиус Земли в милях
13     phi1 = math.radians(lat1)
14     lambda1 = math.radians(lon1)
15     phi2 = math.radians(lat2)
16     lambda2 = math.radians(lon2)
17     dlambda = lambda2 - lambda1
18     dphi = phi2 - phi1
19     sinlat = math.sin(dphi/2.0)
20     sinlong = math.sin(dlambda/2.0)
21     alpha=(sinlat*sinlat) + math.cos(phi1) * \
22         math.cos(phi2) * (sinlong*sinlong)
23     c=2 * math.asin(min(1, math.sqrt(alpha)))
24     d=D*c
25     return d
26
27 if __name__ == "__main__":
28     lat1, lon1 = 40, -83              # Колумбус, штат Огайо
29     lat2, lon2 = 39.91, 116.56        # Пекин
30     print spdist(lat1, lon1, lat2, lon2)
```

2.3. Расстояние от точки до прямой

Пусть $ax + by + c = 0$ – уравнение прямой на плоскости, где a, b, c – константы. Расстояние от точки (x_0, y_0) на этой плоскости до прямой равно

$$\frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

Это можно доказать, вычислив длину перпендикуляра, опущенного из точки (x_0, y_0) на прямую $ax + by + c = 0$. Обозначим (x_1, y_1) точку пересечения прямой и перпендикуляра. Мы знаем, что тангенс угла наклона перпендикулярной прямой равен b/a. Отсюда

$$\frac{y_1 - y_0}{x_1 - x_0} = \frac{b}{a},$$

что дает

$$a(y_1 - y_0) - b(x_1 - x_0) = 0.$$

Возведем обе части равенства в квадрат и выполним простое преобразование:

$$a^2(y_1 - y_0)^2 + b^2(x_1 - x_0)^2 = 2ab(x_1 - x_0)(y_1 - y_0).$$

Прибавим $a^2(x_1 - x_0)^2 + b^2(y_1 - y_0)^2$ к обеим частям. Тогда левую часть можно записать в виде

$$(a^2 + b^2)[(y_1 - y_0)^2 + (x_1 - x_0)^2],$$

а правую часть – в виде

$$[a(x_1 - x_0) + b(y_1 - y_0)]^2 = [ax_1 + by_1 - ax_0 - by_0]^2.$$

Поскольку точка (x_1, y_1) принадлежит также исходной прямой, $ax_1 + by_1 + c = 0$. Следовательно,

$$(a^2 + b^2)[(y_1 - y_0)^2 + (x_1 - x_0)^2] = [ax_1 + by_1 + c]^2,$$

Член $[(y_1 - y_0)^2 + (x_1 - x_0)^2]$ – это квадрат расстояния между (x_0, y_0) и прямой $ax + by + c = 0$. Таким образом, искомое расстояние равно

$$\sqrt{(y_1 - y_0)^2 + (x_1 - x_0)^2} = \frac{|ax_0 + by_0 + c|}{\sqrt{a^2 + b^2}}.$$

В большинстве приложений ГИС легко получить информацию об отрезке, заданном двумя своими концевыми точками. Поэтому необходимо вычислить значения a, b, c в приведенных выше уравнениях по двум точкам. Обозначим их (x_1, y_1) и (x_2, y_2) и положим $dx = x_1 - x_2$, $dy = y_1 - y_2$. Тогда уравнение прямой можно записать в виде с угловым коэффициентом:

$$y = \frac{dy}{dx}x + n,$$

где n – постоянная, которую мы хотим вычислить. Подставляя в это уравнение координаты первой концевой точки, имеем

$$y_1 = \frac{dy}{dx}x_1 + n,$$

откуда

$$n = y_1 - \frac{dy}{dx}x_1.$$

Таким образом, мы получили общий вид уравнения прямой, проходящей через две точки:

$$y = \frac{dy}{dx}x + y_1 - \frac{dy}{dx}x_1.$$

Умножим обе части этого равенства на dx и изменим порядок членов:

$xdy - ydx + y_1dx - x_1dy = 0$.

Отсюда $a = dy$, $b = -dx$, $c = y_1dx - x_1dy$. Зная эти параметры, легко вычислить расстояние от точки до отрезка прямой (листинг 2.3).

Листинг 2.3 ❖ Python-программа для вычисления расстояния между точкой и прямой (point2line.py)

```
import math
from point import *

def point2line(p, p1, p2):
    """
    Вычисляет расстояние между точкой и прямой.
    Вход
      p: точка
      p1 и p2: две точки, определяющие прямую
    Выход
      d: расстояние от точки p до прямой p1p2
    """
    x0 = float(p.x)
    y0 = float(p.y)
    x1 = float(p1.x)
    y1 = float(p1.y)
    x2 = float(p2.x)
    y2 = float(p2.y)
    dx = x1-x2
    dy = y1-y2
    a = dy
    b = -dx
    c = y1*dx - x1*dy
    if a==0 and b==0:                # точки p1 и p2 совпадают
```

```
25          d = math.sqrt((x1-x0)*(x1-x0) + (y1-y0)*(y1-y0))
26      else:
27          d = abs(a*x0+b*y0+c)/math.sqrt(a*a+b*b)
28      return d
29
30 if __name__ == "__main__":
31      p, p1, p2 = Point(10,0), Point(0,100), Point(0,1)
32      print point2line(p, p1, p2)
33      p, p1, p2 = Point(0,10), Point(1000,0.001), Point(-100,0)
34      print point2line(p, p1, p2)
35      p, p1, p2 = Point(0,0), Point(0,10), Point(10,0)
36      print point2line(p, p1, p2)
37      p, p1, p2 = Point(0,0), Point(10,10), Point(10,10)
38      print point2line(p, p1, p2)
```

Мы продемонстрировали применение этого кода на нескольких тестовых примерах (строки 31–38). Печатается следующая информация:

```
10.0
9.9999090909
7.07106781187
14.1421356237
```

2.4. Центроид и площадь многоугольника

Будем представлять многоугольник P с n вершинами в виде последовательности (x_1, y_1), (x_2, y_2), ..., (x_n, y_n), в которую добавим еще одну точку (x_{n+1}, y_{n+1}), чтобы многоугольник был замкнутым. Если в многоугольнике нет дырок, а его граница не имеет самопересечений, то координаты центроида определяются по формулам:

$$x = \frac{1}{6A}\sum_{i=1}^{n}(x_i + x_{i+1})(x_i y_{i+1} - x_{i+1} y_i);$$

$$y = \frac{1}{6A}\sum_{i=1}^{n}(y_i + y_{i+1})(x_i y_{i+1} - x_{i+1} y_i),$$

где A – площадь многоугольника. Если многоугольник выпуклый, то центроид находится внутри него. Но для невыпуклых многоугольников центроид может оказаться снаружи. Хотя понятие центроида тесно связано с центром масс многоугольника, точку, находящуюся снаружи, трудно назвать центром, что вынуждает нас поместить центроид внутрь многоугольника.

Чтобы понять смысл этой формулы, возьмем для примера многоугольник на рис. 2.2 с вершинами (a, b, c, d, e, f, g, a); точка a указана дважды, чтобы замкнуть многоугольник. На каждом отрезке границы можно построить трапецию, площадь которой вычисляется по формуле $(x_2 - x_1)(y_2 + y_1)/2$, где индексы 1 и 2 служат для обозначения двух соседних точек в последовательности. Так, площадь трапеции $abb'a'$ равна $(x_b - x_a)(y_b + y_a)/2$, а площадь трапеции $fgg'f'$ равна $(x_g - x_f)(y_g + y_f)/2$ – эта величина отрицательна. Очевидно, что вычисленные по данной формуле площади трапеций под отрезками ab,

bc, *cd* и *de* (обратите внимание на порядок точек) положительны, а площади трапеций под отрезками *ef*, *fg* и *ga* отрицательны. Поэтому сумма этих величин действительно равна площади многоугольника, поскольку площадь фигуры, ограниченной ломаной $efgaa'e'$, вычитается из площади фигуры, ограниченной ломаной $abcdee'a'$. Таким образом, площадь многоугольника вычисляется по формуле

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1} - x_i)(y_{i+1} + y_i).$$

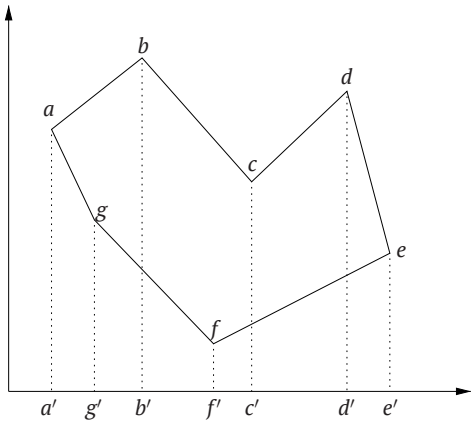

Рис. 2.2 ❖ Использование трапеций для вычисления площади многоугольника. Пунктирные линии проецируют вершины многоугольника на горизонтальную ось

Эта формула основана на разложении многоугольника в последовательность трапеций. Ее можно переписать в таком виде:

$$A = \frac{1}{2}\sum_{i=1}^{n}(x_{i+1}y_i - x_i y_{i+1}).$$

Хотя вторая формула дает такой же результат, как и первая, в прошлом, когда вычисления производились вручную, она была удобнее. В зависимости от порядка точек в последовательности формула площади многоугольника может давать отрицательное значение. Поэтому для получения истинной площади многоугольника нужно взять абсолютную величину.

Python-программа в листинге 2.4 возвращает центроид и площадь многоугольника, вычисленные по приведенным выше формулам. Мы также протестировали программу на примере многоугольника, заданного списком точек.

Листинг 2.4 ❖ Python-программа для вычисления площади и центроида многоугольника (centroid.py)

```
1 from point import *
2
3 def centroid(pgon):
4     """
```

```
    Вычисляет центроид и площадь многоугольника.
    Вход
      pgon: список объектов Point
    Выход
      A: площадь многоугольника
      C: центроид многоугольника
    """
    numvert = len(pgon)
    A = 0
    xmean = 0
    ymean = 0
    for i in range(numvert-1):
        ai = pgon[i].x*pgon[i+1].y - pgon[i+1].x*pgon[i].y
        A += ai
        xmean += (pgon[i+1].x+pgon[i].x) * ai
        ymean += (pgon[i+1].y+pgon[i].y) * ai
    A = A/2.0
    C = Point(xmean / (6*A), ymean / (6*A))
    return A, C

# TEST
if __name__ == "__main__":
    points = [ [0,10], [5,0], [10,10], [15,0], [20,10],
               [25,0], [30,20], [40,20], [45,0], [50,50],
               [40,40], [30,50], [25,20], [20,50], [15,10],
               [10,50], [8, 8], [4,50], [0,10] ]
    polygon = [ Point(p[0], p[1]) for p in points ]
    print centroid(polygon)
```

2.5. Определение положения точки относительно прямой

Результат вычисления площади многоугольника может оказаться положительным или отрицательным в зависимости от порядка точек. Это очень интересная особенность, позволяющая использовать знак площади для определения того, по какую сторону от прямой лежит точка. Для этого нужно вычислить площадь треугольника, образованного данной точкой и двумя точками на прямой. Пусть a, b, c – три вершины треугольника. По выведенной выше формуле получаем площадь треугольника в виде

$$A(abc) = \frac{1}{2}(x_b y_a - x_a y_b + x_c y_b - x_b y_c + x_a y_c - x_c y_a).$$

Включив в скобки член $x_b y_b - x_b y_b$, мы можем переписать эту формулу в более простом виде:

$$\begin{aligned}\text{sideplr}(abc) &= 2A(abc) \\ &= (x_a - x_b)(y_c - y_b) - (x_c - x_b)(y_a - y_b).\end{aligned}$$

Если точка *a* лежит слева от вектора, направленного из *b* в *c*, то вычисление по этой формуле дает отрицательное значение. Например, в конфигурациях точек *a*, *b*, *c* на рис. 2.3A–C площадь треугольника *abc* (обратите внимание на порядок точек) вычисляется так, что площади трапеций, расположенных ниже треугольника (например, трапеции под отрезком *cb* на рис. 2.3A), всегда вычитаются из полной площади (например, суммы площадей трапеций под отрезками *ba* и *ac* на рис. 2.3A). Это приводит к отрицательному значению. С другой стороны, если *a* лежит справа от вектора *bc* (см. рис. 2.3D–F), то значение будет положительно. Код, определяющий положение точки относительно прямой, приведен в листинге 2.5.

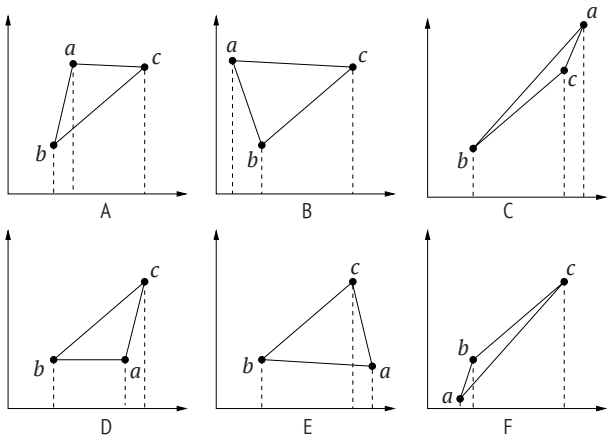

Рис. 2.3 ❖ Положение точки *a* относительно прямой *bc*

Листинг 2.5 ❖ Определение взаимного расположения точки и прямой (sideplr.py)

```
from point import *

def sideplr(p, p1, p2):
    """
    Вычисляет, по какую сторону от вектора p1p2 находится точка p.
    Вход
      p: точка
      p1, p2: начало и конец вектора
    Выход
      -1: p слева p1p2
      0: p на прямой p1p2
      1: p справа p1p2
    """
    return int((p.x-p1.x)*(p2.y-p1.y)-(p2.x-p1.x)*(p.y-p1.y))

if __name__ == "__main__":
    p=Point(1,1)
    p1=Point(0,0)
    p2=Point(1,0)
    print "Положение точки %s относительно прямой %s->%s: %d"%(
```

```
21          p, p1, p2, sideplr(p, p1, p2))
22      print "Положение точки %s относительно прямой %s->%s: %d"%(
23          p, p2, p1, sideplr(p, p2, p1))
24      p = Point(0.5, 0)
25      print "Положение точки %s относительно прямой %s->%s: %d"%(
26          p, p1, p2, sideplr(p, p1, p2))
27      print "Положение точки %s относительно прямой %s->%s: %d"%(
28          p, p2, p1, sideplr(p, p2, p1))
```

В результате выполнения тестов будет напечатано:

```
Положение точки (1,1) относительно прямой (0,0)->(1,0): -1
Положение точки (1,1) относительно прямой (1,0)->(0,0): 1
Положение точки (0.5, 0.0) относительно прямой (0,0)->(1,0): 0
Положение точки (0.5, 0.0) относительно прямой (1,0)->(0,0): 0
```

Если все три точки лежат на одной прямой, то программа вернет 0. Таким образом, мы легко можем узнать, лежит ли точка на прямой или вне нее. Эта техника окажется очень полезной в следующем разделе, когда мы будем обсуждать пересечение отрезков.

2.6. Пересечение отрезков прямых

Сначала рассмотрим пересечение двух прямых: L_1, проходящей через точки (x_1, y_1) и (x_2, y_2), и L_2, проходящей через точки (x_3, y_3) и (x_4, y_4). Их угловые коэффициенты равны соответственно

$$\alpha_1 = \frac{y_2 - y_1}{x_2 - x_1} \quad \text{и} \quad \alpha_2 = \frac{y_4 - y_3}{x_4 - x_3}.$$

Вычислить их точку пересечения просто. Сначала запишем уравнения прямых в виде

$$y = \alpha_1(x - x_1) + y_1$$

и

$$y = \alpha_2(x - x_3) + y_3.$$

Отсюда легко вычислить абсциссу точки пересечения:

$$x = \frac{\alpha_1 x_1 - \alpha_2 x_3 + y_3 - y_1}{\alpha_1 - \alpha_2},$$

– и ее ординату:

$$y = \alpha_1(x - x_1) + y_1.$$

Понятно, что нужно рассмотреть несколько случаев. Если угловые коэффициенты двух прямых равны, то прямые не пересекаются. А что, если одна или обе прямые вертикальны, т. е. угловой коэффициент бесконечен? В этом

случае, если $x_2 - x_1$ и $x_4 - x_3$ одновременно равны нулю, мы имеем две параллельные прямые. Если же только одна из этих величин равна нулю, то абсцисса точки пересечения равна координате x вертикальной прямой. Например, если $x_1 = x_2$, то точка пересечения имеет координаты $x = x_1$ и $y = \alpha_2(x_1 - x_3) + y_3$.

Но если рассматривать только отрезки прямых, то все описанное выше может оказаться ненужным, потому что отрезки могут и не пересекаться. Чтобы проверить, так ли это, нужно рассмотреть концы отрезков. Если оба конца одного отрезка лежат по одну сторону от другого, то отрезки не пересекаются. Для проверки расположения точки относительно отрезка мы можем воспользоваться алгоритмом `sideplr` из предыдущего раздела.

Прежде чем приводить формальный алгоритм вычисления точки пересечения двух отрезков, определим структуру данных для эффективного хранения информации об отрезке (листинг 2.6). Нам понадобится номер стороны (`e`) и координаты концов. Мы сохраняем исходное значение левого конца отрезка (`lp0` в строке 23), потому что в дальнейшем левый конец будет изменяться при вычислении нескольких точек пересечения, сдвигающихся слева направо. Наконец, в атрибуте `status` мы будем хранить признак, показывающий, является ли левый конец отрезка исходным левым концом (по умолчанию) или внутренней точкой, появившейся в результате пересечения с другими отрезками из-за наложения многоугольников, а в атрибуте `c` – свойства отрезка. Часть описанных возможностей понадобится нам в следующем разделе.

Для представления концов отрезков в классе `Segment` используется ранее разработанный класс `Point` из листинга 2.1. Мы также определяем операции сравнения отрезков, переопределив встроенные функции Python, в т. ч. `__eq__` (равно) и `__lt__` (меньше). Мы считаем, что отрезок s_1 меньше отрезка s_2, если s_1 целиком расположен ниже s_2. Мы также включили функцию для проверки того, является ли точка одним из концов отрезка.

Листинг 2.6 ❖ Структура данных для представления отрезков (linesegment.py)

```
1 from point import *
2 from sideplr import *
3
4 ## Два статуса левого конца
5 ENDPOINT = 0 ## изначальный левый конец
6 INTERIOR = 1 ## внутренняя точка отрезка
7
8 class Segment:
9     """
10    Класс для представления отрезков прямых.
11    """
12    def __init__(self, e, p0, p1, c=None):
13        """
14        Конструктор класса Segment.
15        Вход
16          e: ИД отрезка, целое число
17          p0, p1: концы отрезка, объекты Point
18        """
```

```
        if p0>=p1:
            p0,p1 = p1,p0             # p0 всегда левый конец
            self.edge = e             # ИД, задается для всех сторон
            self.lp = p0              # левый конец
            self.lp0 = p0             # первоначальный левый конец
            self.rp = p1              # правый конец
            self.status = ENDPOINT    # статус отрезка
            self.c = c                # c: ИД свойства
    def __eq__(self, other):
        if isinstance(other, Segment):
            return (self.lp==other.lp and self.rp==other.rp)\
                or (self.lp==other.rp and self.rp==other.lp)
        return NotImplemented
    def __ne__(self, other):
        result = self.__eq__(other)
        if result is NotImplemented:
            return result
        return not result
    def __lt__(self, other):
        if isinstance(other, Segment):
            if self.lp and other.lp:
                lr = sideplr(self.lp, other.lp, other.rp)
                if lr == 0:
                    lrr = sideplr(self.rp, other.lp, other.rp)
                    if other.lp.x < other.rp.x:
                        return lrr > 0
                    else:
                        return lrr < 0
                else:
                    if other.lp.x > other.rp.x:
                        return lr < 0
                    else:
                        return lr > 0
        return NotImplemented
    def __gt__(self, other):
        result = self.__lt__(other)
        if result is NotImplemented:
            return result
        return not result
    def __repr__(self):
        return "{0}".format(self.edge)
    def contains(self, p):
        """
        Возвращает True, если точка p является концом отрезка
        """
        if self.lp == p:
            return -1
        elif self.rp == p:
            return 1
        else:
            return 0
```

В листинге 2.7 приведен пример вычисления точки пересечения отрезков. Сначала мы с помощью функции `test_intersect` определяем, пересекаются ли отрезки (строка 60), для чего нужно проверить расположение концов одного отрезка относительно другого, вызвав функцию `sideplr` (например, строка 41). Если оба конца одного отрезка лежат по одну сторону от другого, то отрезки не пересекаются. В противном случае в функции `getIntersectionPoint` используются уравнения, выведенные в начале этого раздела, чтобы вычислить точку пересечения. Эта функция предполагает, что два входных отрезка пересекаются (так что факт пересечения нужно проверить заранее). Тестовые отрезки (строки 57 и 58) пересекаются в точке (1.5, 2.5).

Листинг 2.7 ❖ Вычисление точки пересечения отрезков (intersection.py)

```
 1 from linesegment import *
 2 from sideplr import *
 3
 4 def getIntersectionPoint(s1, s2):
 5     """
 6     Вычисляет точку перечения отрезков s1 и s2.
 7     Предполагается, что s1 and s2 пересекаются.
 8     Факт пересечения необходимо проверить до вызова этой функции.
 9     """
10     x1 = float(s1.lp0.x)
11     y1 = float(s1.lp0.y)
12     x2 = float(s1.rp.x)
13     y2 = float(s1.rp.y)
14     x3 = float(s2.lp0.x)
15     y3 = float(s2.lp0.y)
16     x4 = float(s2.rp.x)
17     y4 = float(s2.rp.y)
18     if s1.lp < s2.lp:
19         x1,x2,y1,y2,x3,x4,y3,y4=x3,x4,y3,y4,x1,x2,y1,y2
20     if x1 != x2:
21         alpha1 = (y2-y1)/(x2-x1)
22     if x3 != x4:
23         alpha2 = (y4-y3)/(x4-x3)
24     if x1 == x2:                    # s1 расположен вертикально
25         y = alpha2*(x1-x3)+y3
26         return Point([x1, y])
27     if x3==x4:                      # s2 расположен вертикально
28         y = alpha1*(x3-x1)+y1
29         return Point([x3, y])
30     if alpha1 == alpha2:            # прямые параллельны
31       return None
32     # нужно вычислить
33     x = (alpha1*x1-alpha2*x3+y3-y1)/(alpha1-alpha2)
34     y = alpha1*(x-x1) + y1
35     return Point(x, y)
36
37 def test_intersect(s1, s2):
38     if s1==None or s2==None:
39         return False
40     # проверка: концы s2 лежат по одну сторону от s1
```

```
    lsign = sideplr(s2.lp0, s1.lp0, s1.rp)
    rsign = sideplr(s2.rp, s1.lp0, s1.rp)
    if lsign*rsign > 0:
        return False
    # проверка: концы s1 лежат по одну сторону от s2
    lsign = sideplr(s1.lp0, s2.lp0, s2.rp)
    rsign = sideplr(s1.rp, s2.lp0, s2.rp)
    if lsign*rsign > 0:
        return False
    return True

if __name__ == "__main__":
    p1 = Point(1, 2)
    p2 = Point(3, 4)
    p3 = Point(2, 1)
    p4 = Point(1, 4)
    s1 = Segment(0, p1, p2)
    s2 = Segment(1, p3, p4)
    s3 = Segment(2, p1, p2)
    if test_intersect(s1, s2):
        print getIntersectionPoint(s1, s2)
        print s1==s2
        print s1==s3
```

2.7. Операция «точка внутри многоугольника»

Выяснение того, находится ли точка внутри многоугольника, – одна из важнейших операций при использовании ГИС. Например, щелкая мышью в точке на цифровой карте мира, мы ожидаем быстро получить информацию о регионе, в котором эта точка находится. В случае простого (несамопересекающегося) выпуклого многоугольника точка находится внутри, если она лежит по одну и ту же сторону от всех сторон многоугольника. Этот подход выглядит вычислительно сложным, поскольку требуется много операций умножения, а кроме того, он не работает для невыпуклых многоугольников (рис. 2.4).

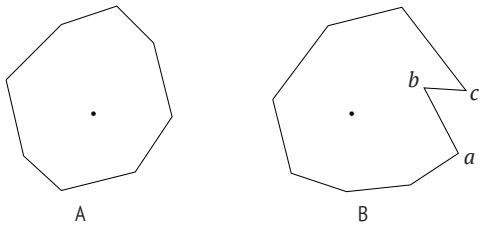

Рис. 2.4 ❖ Определение расположения точки относительно многоугольника путем выяснения того, по какую сторону точка расположена относительно его сторон. Предполагается, что задана последовательность вершин (в порядке обхода по часовой стрелке или против часовой стрелки): (A) точка находится по одну сторону от всех отрезков; (B) точка находится с противоположных сторон отрезков *ab* и *bc*

2.7.1. Алгоритм чет-нечет

Алгоритм чет-нечет – популярный метод, известный также под названиями алгоритм бросания лучей (ray-casting) или алгоритм числа пересечений. В этом алгоритме проверка пересечения выполняется путем проведения полупрямой (бросания луча) из точки. Если луч пересекает границу многоугольника в четном числе точек, то точка находится внутри многоугольника, в противном случае снаружи. Для удобства будем рисовать луч горизонтально (рис. 2.5). В целом процесс бесхитростный, но мы рассчитываем избежать фактического вычисления точек пересечения, потому что эта операция занимает много времени, особенно если производится многократно.

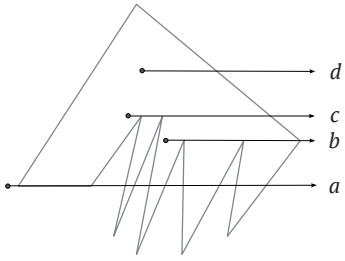

Рис. 2.5 ❖ Алгоритм «точка внутри многоугольника». Начальная точка луча *a* находится вне многоугольника, а начальные точки лучей *b*, *c* и *d* – внутри многоугольника

На рис. 2.6 изображены различные случаи, показывающие, когда вычислять точку пересечения необходимо, а когда без этого можно обойтись. Мы видим горизонтальный луч, начинающийся в точке *A*, и несколько отрезков, обозначенных буквами от *a* до *e* и *b′*. Мы пытаемся проверить, пересекается ли отрезок с лучом, не вычисляя саму точку пересечения. Отрезки *a*, *d* и *e* не пересекаются с лучом, потому что абсциссы обоих концов лежат по одну сторону от начала луча (*a*), или ординаты обоих концов лежат по одну сторону луча (*d* и *e*). Отрезок *c* мы засчитываем как пересечение, потому что он действительно пересекает луч, но вычислять точку пересечения не нужно, т. к. мы заведомо знаем, что этот отрезок пересекает луч: ординаты концов *c* лежат по разные стороны от луча, а их абсциссы расположены справа от точки *A*. Для отрезков *b* и *b′* невозможно сходу сказать, пересекают они луч или нет, так что придется вычислять точку пересечения. По счастью, нам нужно вычислить только абсциссу точки пересечения, чтобы решить, пересекается отрезок с лучом или нет. В случае отрезка *b* абсцисса точки пересечения находится слева от точки *A*, поэтому отрезок не пересекает луч. В случае отрезка *b′* точка пересечения находится справа от точки *A*, поэтому отрезок пересекает луч.

Имеется важное исключение, часто встречающееся в реальных приложениях. Что, если луч проходит через оба конца отрезка? А если луч проходит через один конец отрезка? В таких случаях мы можем автоматически прибавить к соответствующим концам очень маленькое значение, так что тео-

ретически они окажутся «над» лучом. В результате мы можем быть уверены, что либо отрезок пересекает луч, либо нет – третьего не дано. В реализации этого алгоритма мы считаем, что конец отрезка расположен «выше» луча, если его ордината «больше или равна» ординате луча. В противном случае считается, что отрезок расположен ниже луча. Например, считается, что луч *b* на рис. 2.5 пересекает границу многоугольника пять раз (а не один), а луч *a* пересекает границу восемь раз.

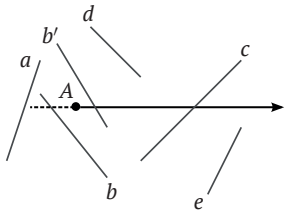

Рис. 2.6 ❖ Различные случаи при вычислении точки пересечения

Функция `pip_cross` в листинге 2.8 возвращает признак, показывающий, находится ли точка внутри многоугольника, используя алгоритм чет-нечет. В переменных `p1` и `p2` мы храним соседние вершины многоугольника. Переменные `yflag1` и `yflag2` показывают положение ординат двух вершин относительно луча (выше/на или ниже), а переменные `xflag1` и `xflag2` – положение абсцисс (слева/на или справа). В строке 25 проверяется, лежат ли точки `p1` и `p2` по разные стороны от луча. Если да, то далее в строке 28 проверяется, лежат ли они по разные стороны от начальной точки луча. Если обе точки находятся справа от начальной, то мы засчитываем одно пересечение. В противном случае придется вычислить абсциссу точки пересечения (строка 34) и решить, пересекается ли отрезок с лучом (строка 35).

Листинг 2.8 ❖ Алгоритм чет-нечет проверки расположения точки относительно многоугольника (point_in_polygon.py)

```
 1 import math
 2 from point import *
 3
 4 def pip_cross(point, pgon):
 5     """
 6     Определяет, находится ли точка внутри многоугольника. В основе кода
 7     лежит программа на C из книги Haines "Graphics Gems IV" (1994).
 8     Вход
 9       pgon: список вершин многоугольника
10       point: точка
11     Выход
12       Возвращает булево значение True или False и сколько раз луч
13       пересекает границу многоугольника
14     """
15     numvert = len(pgon)
16     tx=point.x
```

```
    ty=point.y
    p1 = pgon[numvert-1]
    p2 = pgon[0]
    yflag1 = (p1.y >= ty)              # p1 на одном уровне с point или выше нее
    crossing = 0
    inside_flag = 0
    for j in range(numvert-1):
        yflag2 = (p2.y >= ty)          # p2 на одном уровне с point или выше нее
        if yflag1 != yflag2:           # по разные стороны от луча
            xflag1 = (p1.x >= tx)      # слева от p1
            xflag2 = (p2.x >= tx)      # слева от p2
            if xflag1 == xflag2:       # обе точки справа
                if xflag1:
                    crossing += 1
                    inside_flag = not inside_flag
            else:
                m = p2.x - float((p2.y-ty))*\
                    (p1.x-p2.x)/(p1.y-p2.y)
                if m >= tx:
                    crossing += 1
                    inside_flag = not inside_flag
        yflag1 = yflag2
        p1 = p2
        p2 = pgon[j+1]
    return inside_flag, crossing

if __name__ == "__main__":
    points = [ [0,10], [5,0], [10,10], [15,0], [20,10],
               [25,0], [30,20], [40,20], [45,0], [50,50],
               [40,40], [30,50], [25,20], [20,50], [15,10],
               [10,50], [8, 8], [4,50], [0,10] ]
    ppgon = [Point(p[0], p[1]) for p in points ]
    inout = lambda pip: "ВНУТРИ" if pip is True else "СНАРУЖИ"
    point = Point(10, 30)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
    point = Point(10, 20)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
    point = Point(20, 40)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
    point = Point(5, 40)
    print "Точка %s находится %s"%(
        point, inout(pip_cross(point, ppgon)[0]))
```

Для тестирования программы мы взяли многоугольник и точки, показанные на рис. 2.7. Программа напечатала следующие результаты:

```
Точка (10,30) находится ВНУТРИ
Точка (10,20) находится ВНУТРИ
Точка (20,40) находится ВНУТРИ
Точка (5,40) находится СНАРУЖИ
```

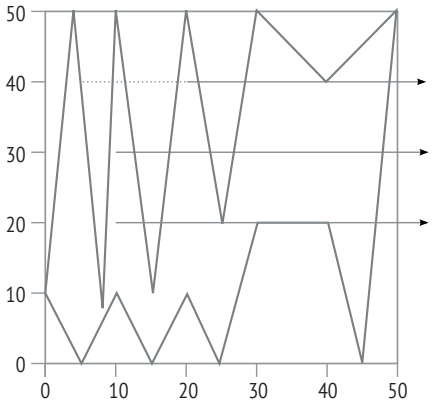

Рис. 2.7 ❖ Тесты для алгоритма чет-нечет

2.7.2. Алгоритм на основе числа оборотов

В обоих случаях, показанных на рис. 2.4, мы видим, что стороны многоугольника совершают оборот вокруг внутренней точки. Если же точка находится вне многоугольника, то стороны не оборачиваются вокруг нее. Алгоритм на основе числа оборотов пользуется этим свойством, чтобы определить, находится точка внутри или снаружи многоугольника. В общем случае, если даны два n-мерных вектора $X = (x_1, x_2, \ldots, x_n)$ и $Y = (y_1, y_2, \ldots, y_n)$, то имеем $X \cdot Y = |X||Y|\cos\theta$, где θ – угол между векторами, $X \cdot Y$ – скалярное произведение двух векторов, равное $\sum_{i=1}^{n} x_i y_i$, а $|X|$ и $|Y|$ – нормы (или длины) X и Y соответственно. Если задан отрезок между точками $v_i = (x_i, y_i)$ и $v_{i+1} = (x_{i+1}, y_{i+1})$ и точка $p = (x, y)$, то угол между p и отрезком обозначается θ_i и вычисляется по формуле

$$\theta_i = \arccos\left[\frac{\overrightarrow{v_i p} \cdot \overrightarrow{v_{i+1} p}}{|\overrightarrow{v_i p}||\overrightarrow{v_{i+1} p}|}\right]$$
$$= \arccos\left[\frac{(x - x_i)(x - x_{i+1}) + (y - y_i)(y - y_{i+1})}{\sqrt{(x - x_i)^2 + (y - y_i)^2}\sqrt{(x - x_{i+1})^2 + (y - y_{i+1})^2}}\right],$$

где $\overrightarrow{v_i p}$ – вектор из точки v_i в точку p, а $|\overrightarrow{v_i p}|$ – длина этого вектора. Предположим теперь, что многоугольник содержит n вершин и $v1 = v_n$, т. е. многоугольник замкнутый.

Тогда сумма углов между точкой и всеми n сторонами многоугольника, называемая индексом точки относительно кривой или числом оборотов, равна

$$wn = \frac{1}{2\pi}\sum_{i=1}^{n-1}\theta_i.$$

Точка находится внутри многоугольника, если ее индекс не равен нулю.

Описанный выше алгоритм на основе числа оборотов вычислительно неэффективен из-за использования тригонометрических функций. Эту проблему можно решить, оценивая число оборотов сторон многоугольника во-

круг точки по их направлениям. Если сторона поднимается и пересекает уровень *Y* точки, то число оборотов увеличивается на 1, а если опускается, то уменьшается на 1. Как показано на рис. 2.8, левые концы обоих горизонтальных отрезков будут считаться находящимися внутри многоугольника, потому что отрезок *a* пересекает одна восходящая сторона, а отрезок *b* – две восходящие стороны. По существу, этот подход оказывается таким же, как основанный на числе пересечений, и эффективность у них одинаковая. Код обоих вариантов метода на основе числа оборотов приведен в листинге 2.9; функция `pip_wn` реализует традиционный алгоритм, а `pip_wn1` – вариант с подсчетом числа пересечений. Тестовые данные, включенные в листинг, отражают ситуацию, сходную с рис. 2.8.

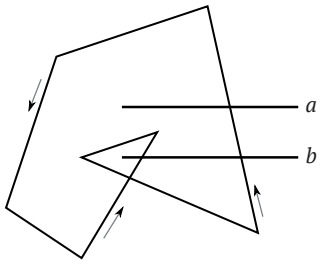

Рис. 2.8 ❖ Проверка принадлежности точки многоугольнику методом на основе числа оборотов. Стрелками показаны направления сторон многоугольника – против часовой стрелки

Листинг 2.9 ❖ Алгоритм на основе числа оборотов (point_in_polygon_winding.py)

```
import math
from point import *

def is_left(p, p1, p2):
    """
    Проверяет, лежит ли точка слева от отрезка, соединяющего
    p1 и p2
    Выход
       0 точка лежит на отрезке
      >0 p лежит слева от отрезка
      <0 p лежит справа от отрезка
    """
    return (p2.x-p1.x)*(p.y-p1.y) - (p.x-p1.x)*(p2.y-p1.y)

def pip_wn(pgon, point):
    """
    Определяет, находится ли точка внутри многоугольника, применяя алгоритм
    на основе числа оборотов с использованием тригонометрических функций.
    В основе коде лежит программа на C из книги Haines  "Graphics Gems IV"
   (1994).
    Вход
      pgon: список вершин многоугольника
      point: точка
```

```
    Выход
      Возвращает булево значение True или False и сколько раз луч
      пересекает границу многоугольника
    """
    if pgon[0] != pgon[-1]:
        pgon.append(pgon[0])
    n = len(pgon)
    xp = point.x
    yp = point.y
    wn = 0
    for i in range(n-1):
        xi = pgon[i].x
        yi = pgon[i].y
        xi1 = pgon[i+1].x
        yi1 = pgon[i+1].y
        thi = (xp-xi)*(xp-xi1) + (yp-yi)*(yp-yi1)
        norm = (math.sqrt((xp-xi)**2+(yp-yi)**2)
               * math.sqrt((xp-xi1)**2+(yp-yi1)**2))
        if thi != 0:
            thi = thi/norm
        thi = math.acos(thi)
        wn += thi
    wn /= 2*math.pi
    wn = int(wn)
    return wn is not 0, wn

def pip_wn1(pgon, point):
    """
    Определяет, находится ли точка внутри многоугольника, применяя алгоритм
    на основе числа оборотов без использования тригонометрических функций.
    В основе коде лежит программа на C из книги Haines  "Graphics Gems IV"
   (1994).
    Вход
      pgon: список вершин многоугольника
      point: точка
    Выход
      Возвращает булево значение True или False и сколько раз луч
      пересекает границу многоугольника
    """
    wn = 0
    n = len(pgon)
    for i in range(n-1):
        if pgon[i].y <= point.y:
            if pgon[i+1].y > point.y:
                if is_left(point, pgon[i], pgon[i+1])>0:
                    wn += 1
        else:
            if pgon[i+1].y <= point.y:
                if is_left(point, pgon[i], pgon[i+1])<0:
                    wn -= 1
    return wn is not 0, wn

```

```
76 if __name__ == "__main__":
77     pgon = [ [2,3], [7,4], [6,6], [4,2], [11,5],
78              [5,11], [2,3] ]
79     point = Point(6, 4)
80     ppgon = [Point(p[0], p[1]) for p in pgon ]
81     print pip_wn(ppgon, point)
82     print pip_wn1(ppgon, point)
```

В алгоритме на основе числа оборотов есть очевидная проблема: начальная (левая) точка луча *b* на рис. 2.8 будет считаться находящейся внутри многоугольника, потому что ее индекс больше нуля. Алгоритм чет-нечет сообщил бы, что эта точка находится снаружи, потому что луч пересекает границу многоугольника дважды. По поводу того, считать ли эту точку внутренней или внешней, было много споров. Но мы можем разрубить этот гордиев узел, потребовав, чтобы прямоугольники были простыми, т. е. чтобы никакая сторона не пересекала никакую другую сторону в точках, отличных от вершин. При таком предположении мы имеем на рисунке два многоугольника, меньший содержит точку, а больший – нет.

2.8. Картографические проекции

Под картографической проекцией понимается процесс преобразования из трехмерной сферической системы координат, в которой положение точки определяется широтой и долготой, в декартову систему координат на плоскости. Такое преобразование часто бывает необходимым, потому что представлять точки земной поверхности на плоском листе бумаге удобнее, чем на сфере. Существует много способов выполнить картографическую проекцию. Мы опишем только два метода, представляющих различные подходы к преобразованию.

Проекция Робинсона (рис. 2.9) – одна из самых распространенных в картографии, и не только. Она не сохраняет ни площади, ни локальные формы, но показывает мир, так чтобы были отчетливо видны приполярные области. Проекция Робинсона отличается от многих других проекций тем, что не основана всецело на математических формулах. Средний меридиан представлен прямой линией, длина которой составляет 0.5072 длины экватора. Параллели изображаются прямыми, расстояние между которыми одинаково

Рис. 2.9 ❖ Проекция Робинсона

в области между 38° с.ш. и 38° ю.ш., но уменьшается при приближении к полюсам. Полюсы в проекции Робинсона изображаются отрезками прямых длиной, равной 0.5322 длины экватора. На каждой параллели меридианы размещены через равные промежутки.

Точнее, начало координат в проекции Робинсона находится в точке пересечения экватора и среднего меридиана. Параметры в табл. 2.1 вычислены Робинсоном. Для каждой из 19 широт в диапазоне от 0° до 90° с шагом 5° Робинсон дает длины параллели и расстояние на карте от параллели до экватора. Поскольку меридианы на каждой параллели размещены с одинаковыми промежутками, легко вычислить, где на параллели находится отметка долготы относительно среднего меридиана, т. е. какова координата *X* этой точки. Однако точная длина известна лишь для отдельных параллелей (столбец 2 в табл. 2.1). И значения координаты *Y* известны только для этих параллелей (заметим, что параллели размещены с постоянным шагом в области между 38° с.ш. и 38° ю.ш.). Для любой другой широты находить длину (*A*) и расстояние до экватора (*B*) нужно путем интерполяции.

Таблица 2.1. Длины параллелей и их расстояния до экватора в проекции Робинсона

Широта (φ)	Длина (*A*)	Расстояние до экватора (*B*)
00	1.0000	0.0000
05	0.9986	0.0620
10	0.9954	0.1240
15	0.9900	0.1860
20	0.9822	0.2480
25	0.9730	0.3100
30	0.9600	0.3720
35	0.9427	0.4340
40	0.9216	0.4958
45	0.8962	0.5571
50	0.8679	0.6176
55	0.8350	0.6769
60	0.7986	0.7346
65	0.7597	0.7903
70	0.7186	0.8435
75	0.6732	0.8936
80	0.6213	0.9394
85	0.5722	0.9761
90	0.5322	1.0000

Однако интерполяция – это игра в угадайку, потому что Робинсон не оставил упоминаний о том, какой метод интерполяции был использован в оригинальной работе. Многие ученые предлагали различные способы аппроксимации проекции Робинсона. Один полезный метод интерполяции состоит в том, чтобы использовать полиномиальную функцию, так чтобы кривая проходила через все заданные в табл. 2.1 точки (широту и *A* или *B*).

Ниже описан алгоритм Невилла для нахождения такой функции. В общем случае предположим, что существует полином $p(x) = \sum_{i=0}^{n} a_i x_i$, который аппроксимирует $n + 1$ пару входных данных $\{(x_i, y_i)\}$, так что $p(x_i) = y_i$, $0 \leqslant i \leqslant n$. В табл. 2.2 показано, как алгоритм Невилла используется для вычисления значения в точке x, если известно пять пар (x_i, y_i) (первые два столбца). Обозначим $p_{i,j}(x)$ – результат интерполяции в точке x полиномом степени $j - i$ (степенью полинома называется максимальный показатель степени x в выражающей его формуле). На итерации 0 (столбец 3) каждое значение равно соответствующему значению y: мы имеем полином степени 0 $p_{i,i} = x^0 y_i = y_i$. Начиная с итерации 1 (столбец 4) каждое значение в столбце является результатом линейной интерполяции двух значений – над ним и под ним – в предыдущем столбце. Например:

$$p_{1,2}(x) = \frac{x_2 - x}{x_2 - x_1} p_{1,1}(x) - \frac{x_1 - x}{x_2 - x_1} p_{2,2}(x)$$
$$= \frac{x_2 - x}{x_2 - x_1} y_1 - \frac{x_1 - x}{x_2 - x_1} y_2.$$

Совершая итерации, мы вычисляем все полиномы первой степени (показаны в третьем столбце), а затем переходим к вычислению следующего столбца и так далее, пока не останется вычислить единственное значение, определяемое входными парами.

Таблица 2.2. Алгоритм Невилла

x_0	y_0	$p_{0,0}(x)$				
			$p_{0,1}(x)$			
x_1	y_1	$p_{1,1}(x)$		$p_{0,2}(x)$		
			$p_{1,2}(x)$		$p_{0,3}(x)$	
x_2	y_2	$p_{2,2}(x)$		$p_{1,3}(x)$		$p_{0,4}(x)$
			$p_{2,3}(x)$		$p_{1,4}(x)$	
x_3	y_3	$p_{3,3}(x)$		$p_{2,4}(x)$		
			$p_{3,4}(x)$			
x_4	y_4	$p_{4,4}(x)$				

Итоговую формулу алгоритма Невилла можно записать в виде определителя квадратной матрицы:

$$p_{i,j}(x) = \frac{1}{x_j - x_i} \begin{vmatrix} p_{i,j-1}(x) & x_i - x \\ p_{i+1,j}(x) & x_j - x \end{vmatrix},$$

или, эквивалентно,

$$p_{i,j}(x) = \frac{(x_j - x) p_{i,j-1}(x) + (x - x_i) p_{i+1,j}(x)}{x_j - x_i}.$$

В листинге 2.10 приведена реализация алгоритма Невилла. В этом коде нам не нужно заводить отдельный двумерный массив для хранения значений

p_{ij}. Если внимательно присмотреться к табл. 2.2, то мы увидим, что на k-й итерации нам нужно сохранить всего $n - k$ значений, так что списка размера n будет вполне достаточно (строка 12). Важно, что после обновления $p[i]$ новое значение не влияет на вычисления на той же итерации (строка 20).

Листинг 2.10 ❖ Алгоритм Невилла (neville.py)

```
 1 def neville(datax, datay, x):
 2     """
 3     Находим интерполированное значение с помощью алгоритма Невилла.
 4     Вход
 5       datax: входные x в списке длины n
 6       datay: входные y в списке длины n
 7       x: точка, в которой ищется интерполированное значение
 8     Выход
 9       p[0]: полином степени n
10     """
11     n = len(datax)
12     p = n*[0]
13     for k in range(n):
14         for i in range(n-k):
15             if k == 0:
16                 p[i] = datay[i]
17             else:
18                 p[i] = ((x-datax[i+k])*p[i]+ \
19                         (datax[i]-x)*p[i+1])/ \
20                         (datax[i]-datax[i+k])
21     return p[0]
```

Теперь мы готовы написать программу для вычисления проекции Робинсона (листинг 2.11). В самом начале мы инициализируем три списка значениями в табл. 2.1. В качестве среднего используется нулевой меридиан, мы еще вернемся к этому вопросу при обсуждении проекции Мольвейде. Функция `transform1` преобразует широту и долготу в систему координат проекции Робинсона. Поскольку все широты в таблице положительны, мы заменяем отрицательные широты (если они возникнут в процессе вычислений) абсолютными величинами (строка 62). В конце мы произведем обратное преобразование, если необходимо. В строке 65 проверяется, что входная широта не больше 90°. В строке 67 мы находим индекс наибольшей широты, которая меньше или равна входной широте. Это делает функция `find_le`, в которой используется эффективный метод двоичного поиска – это возможно, поскольку список `latitudes` отсортирован. Описание функции можно найти в официальной документации по Python[1].

Найдя в таблице индекс широты, непосредственно предшествующей входной широте (т. е. наибольшей из тех, что меньше нее), мы должны определить широты и значения A или B, которые будут использоваться для интерполяции. Использовать все имеющиеся в таблице значения просто, но рискованно. Дело в том, что степень интерполирующего полинома равна $n - 1$,

[1] https://docs.python.org/2/library/bisect.html.

где n – количество входных пар. Если мы будем использовать все значения в таблице, то получим полином степени 18. Скорее всего, это приведет к переподгонке, когда мы получим точные значения в самих заданных точках, но плохую аппроксимацию в интервалах между ними. Поэтому обычно мы берем для интерполяции два значения слева от входной широты и два значения справа. Например, если входная широта равна 12.5, то мы проведем интерполяцию по широтам 5, 10, 15 и 20, а функция `find_le` вернет 2 – индекс широты 10. Задача усложняется, когда входная широта находится близко к началу или к концу таблицы, потому что в этом случае нам не хватит индексов с одной стороны. Например, если входная широта равна 2.5, то нам придется использовать только одну широту слева, т. е. для интерполяции доступно лишь три широты: 0, 5 и 10.

Зная левый индекс, найденный функцией `find_le`, мы можем найти остальные с помощью функции `get_interpolation_range`. Эта функция позволяет задать желаемое количество индексов по обе стороны. По умолчанию с каждой стороны используется два индекса. В строке 68 мы получаем диапазон широт для интерполяции значения B, или координаты Y. Но чтобы интерполировать расстояние от параллели до экватора, мы должны учитывать, что параллели между 38° с.ш. и 38° ю.ш. расположены через равные промежутки. В таком случае мы можем задать только одну табличную широту с каждой стороны от входной. Из предыдущего обсуждения понятно, что в случае, когда имеется всего два известных значения, алгоритм Невилла выполняет линейную интерполяцию, а именно это нам и нужно для равноотстоящих параллелей. Это делается в строке 72.

Мы дважды вызываем алгоритм Невилла: для интерполяции расстояния от входной параллели до экватора (строка 70) и длины параллели (строка 74). В оставшейся части программы производится инициализация с учетом требований проекции Робинсона: длина среднего меридиана равна 0.5072 длины экватора (строка 75) и меридианы пересекают параллели в равноотстоящих точках (строки 76 и 77).

Листинг 2.11 ❖ Преобразование точек в проекцию Робинсона (transform1.py)

```
 1 import bisect
 2 from neville import *
 3 from numpy import fabs
 4
 5 latitudes=[0, 5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,
 6            65, 70, 75, 80, 85, 90]
 7
 8 # длины параллелей на каждой широте в списке latitudes
 9 A=[1.0000, 0.9986, 0.9954, 0.9900, 0.9822, 0.9730, 0.9600,
10    0.9427, 0.9216, 0.8962, 0.8679, 0.8350, 0.7986, 0.7597,
11    0.7186, 0.6732, 0.6213, 0.5722, 0.5322]
12
13 # расстояние от каждой параллели до экватора
14 # эти значения нужно умножать на 0.5072
15 B=[0.0000, 0.0620, 0.1240, 0.1860, 0.2480, 0.3100, 0.3720,
16    0.4340, 0.4958, 0.5571, 0.6176, 0.6769, 0.7346, 0.7903,
```

```
   0.8435, 0.8936, 0.9394, 0.9761, 1.0000]

def find_le(a, x):
    """Находит самый правый элемент, меньший или равный x"""
    i = bisect.bisect_right(a, x)
    if i:
        return i-1
    raise ValueError

def get_interpolation_range(sidelen, n, i):
    """
    Находит диапазон индексов для интерполяции
    проекции Робинсона
    Вход
      sidelen: количество элементов по обе стороны от i,
               включая i в число левых
      n: общее число элементов
      i: индекс наибольшего элемента, меньшего значения
    Выход
      ileft: индекс левого значения (включая)
      iright: индекс правого значения (не включая)
    """
    if i<sidelen:
        ileft = max([0, i-sidelen+1])
    else:
        ileft = i-sidelen+1
        if i>=n-sidelen:
            iright = min(n, i+sidelen+1)
        else:
            iright = i+sidelen+1
    return ileft, iright

def transform1(lon, lat):
    """
    Возвращает результат преобразования lon и lat
    в координаты на проекции Робинсона.
    Вход
      lon: долгота
      lat: широта
    Выход
      x: координата x (начало координат в 0,0)
      y: координата y (начало координат в 0,0)
    """
    n = len(latitudes)
    south = False
    if lat<0:
        south = True
        lat = fabs(lat)
    if lat>90:
        return
    i = find_le(latitudes, lat)
    ileft, iright = get_interpolation_range(2, n, i)
    y = neville(latitudes[ileft:iright],
```

```
70                B[ileft:iright], lat)
71    if lat<=38:
72        ileft, iright = get_interpolation_range(1, n, i)
73    x = neville(latitudes[ileft:iright],
74                A[ileft:iright], lat)
75    y = 0.5072*y/2.0
76    dx = x/360.0
77    x = dx*lon
78    if south:
79        y = -1.0 * y
80    return x, y, ileft, i, iright
```

Для тестирования алгоритма проекции Робинсона мы будем использовать данные двух видов. Программа в листинге 2.12 поможет сгенерировать набор точек для этой цели. Сначала мы хотим нарисовать сетку меридианов и параллелей. Каждая из этих линий состоит из множества точек. Для хранения данной информации мы используем простой список, каждый элемент которого сам является списком из трех значений: идентификатор линии, а также долгота и широта точки. Точки, образующие каждую линию, нужно хранить последовательно, чтобы обеспечить гладкое изображение. Мы используем вложенный цикл, начинающийся в строке 6, для создания параллелей с шагом 10° и вложенный цикл, начинающийся в строке 11, для создания меридианов с таким же шагом. Для каждого меридиана мы берем по одной точке на каждой параллели, кроме параллелей севернее 80° с.ш. и южнее 80° ю.ш., где выборка более плотная (с шагом 1°), чтобы кривые получились гладкими.

Второй набор данных – это очертания материков в масштабе 1:110 млн[1], которые мы хотим нанести на сетку. В исходном наборе данных очертания хранятся в формате файла фигур (shapefile). С помощью модуля OGR мы читаем координаты и преобразуем их в простую структуру данных. Имя файла фигур передается на вход функции (строка 3), а затем используется в строке 24, где модуль OGR открывает файл. Дополнительные сведения о модуле OGR и смежных вопросах см. в приложении В. По ходу дела мы запоминаем количество строк в сетке параллелей и меридианов (`numgraticule`) и общее количество строк (`numline`), позже они понадобятся для рисования проекции карты.

Листинг 2.12 ❖ Подготовка данных для карты мира (worldmap.py)

```
 1 from osgeo import ogr
 2
 3 def prep_projection_data(fname):
 4     points=[]
 5     linenum = 0
 6     for lat in range(-90, 91, 10):
 7         for lon in range(-180, 181, 10):
 8             points.append([linenum, lon, lat])
 9     linenum += 1
10
11     for lon in range(-180, 181, 10):
```

[1] http://www.naturalearthdata.com/downloads/110m-physical-vectors/.

```
12          for lat in range(-90, -80, 1):
13              points.append([linenum, lon, lat])
14          for lat in range(-80, 80, 10):
15              points.append([linenum, lon, lat])
16          for lat in range(80, 91, 1):
17              points.append([linenum, lon, lat])
18          linenum += 1
19
20      numgraticule = linenum
21
22      driveName = "ESRI Shapefile"
23      driver = ogr.GetDriverByName(driveName)
24      vector = driver.Open(fname, 0)
25      layer = vector.GetLayer(0)
26
27      for i in range(layer.GetFeatureCount()):
28          f = layer.GetFeature(i)
29          geom = f.GetGeometryRef()
30          for i in range(geom.GetPointCount()):
31              p = geom.GetPoint(i)
32              points.append([linenum, p[0], p[1]])
33          linenum += 1
34
35      numline = max([p[0] for p in points]) + 1
36
37      return points, numgraticule, numline
```

Последний шаг – преобразовать данные о широте и долготе в координаты на проекции Робинсона. Это делает программа в листинге 2.13. Здесь мы пользуемся мощным модулем построения графиков и визуализации Matplotlib. Конечно, карту можно нарисовать в любом пакете ГИС, но мы остаемся привержены инструментам с открытым исходным кодом конкретно на языке Python. Дополнительные сведения об этом модуле можно найти в приложении А.

Сначала мы получаем данные в исходном формате (строка 7), затем преобразуем их в систему координат проекции Робинсона (строка 11), сохраняя структуру данных: ИД линии, координаты *X* и *Y* (строка 12). Приступая к отображению данных, мы сначала рисуем рамку, в которой будет размещаться карта (строка 24). В цикле `for` мы перебираем все строки данных (строка 14), задаем цвет (строка 15), так чтобы сетка параллелей и меридианов отображалась светло-серым, а очертания материков – темно-серым. Заметим, что в Python цвет можно задать разными способами, и здесь мы в одном случае используем словесное название «lightgrey», а в другом – задание цвета в формате HTML: яркости красной, зеленой и синей составляющей (каждая задается двумя 16-ричными цифрами) в строке, начинающейся знаком «#». В строках 19 и 20 мы получаем из списков точек соответственно координаты *X* и *Y*, которые затем используем для нанесения на график двумерной линии с помощью библиотеки Matplotlib (строка 21). Заметим, что в этот момент линии еще не отображаются, а только добавляются на график, который будет показан позже командой `show`.

Оставшаяся часть кода – процедуры, необходимые для правильного изображения карты. Особенно важна строка 23, в которой говорится, что оси должны масштабироваться, поскольку на проекции они должны быть разной длины. В строке 24 мы получаем текущие оси и вносим изменения, т. к. не хотим, чтобы оси были видны на карте. Результат всех этих действий показан на рис. 2.9, который сгенерирован закомментированной строкой 28, сохраняющей изображение в инкапсулированном формате Postscript, пригодном для публикации.

Листинг 2.13 ❖ Тестирование проекции Робинсона на данных карты мира (test_projection.py)

```
 1 from osgeo import ogr
 2 import matplotlib.pyplot as plt
 3 from transform1 import *
 4 from worldmap import *
 5
 6 fname = '../data/ne_110m_coastline.shp'
 7 pp, numgraticule, numline = prep_projection_data(fname)
 8
 9 points=[]
10 for p in pp:
11     p1 = transform1(p[1], p[2])
12     points.append([p[0], p1[0], p1[1]])
13
14 for i in range(numline):
15     if i<numgraticule:
16         col = 'lightgrey'
17     else:
18         col = '#5a5a5a'
19     ptsx = [p[1] for p in points if p[0]==i]
20     ptsy = [p[2] for p in points if p[0]==i]
21     plt.plot(ptsx, ptsy, color=col)
22
23 plt.axis('scaled')
23 frame = plt.gca()
25 frame.axes.get_xaxis().set_visible(False)
26 frame.axes.get_yaxis().set_visible(False)
27 frame.set_frame_on(False)
28 #plt.savefig(robinson.eps',bbox_inches='tight',pad_inches=0)
29 frame.set_frame_on(True)
30 plt.show()
```

Проекция Мольвейде – равновеликая проекция, сохраняющая площадь. Она описывается следующими уравнениями преобразования:

$$x = \frac{\sqrt{8}}{\pi} R(\lambda - \lambda_0)\cos\theta;$$

$$y = \sqrt{2}R\sin\theta,$$

где R – радиус глобуса, на который проецируется Земля (обычно принимается равным 1), λ – преобразуемая долгота, λ_0 – долгота среднего меридиана, а θ – угловой параметр, который должен удовлетворять условию

$$2\theta + \sin 2\theta = \pi \sin\varphi,$$

где φ – преобразуемая широта. Фактическое значение θ для каждой широты φ оценивается с помощью какого-нибудь итеративного алгоритма, например по методу Ньютона–Рафсона:

$$\Delta\theta' = -\frac{\theta' + \sin\theta' - \pi\sin\varphi}{1 + \cos\theta'}.$$

Чтобы воспользоваться этим итеративным методом, мы вначале присваиваем θ' значение φ и получаем первое значение $\Delta\theta'$. Затем в качестве нового значения θ' выбираем $\theta' + \Delta\theta'$ и повторяем процесс. Так продолжается, пока $\Delta\theta'$ не станет очень малым. В этот момент последнее значение θ' используется для вычисления θ по формуле

$$\theta = \theta'/2.$$

Этот алгоритм реализован в функции `opt_theta` в листинге 2.14; итерации прекращаются, когда $\Delta\theta'$ оказывается меньше 0.0000001. Отметим, что обычно углы выражаются в градусах, но в тригонометрических функциях Python в качестве единицы измерения применяются радианы, поэтому необходимо не забывать о преобразовании одного в другое. Для этой цели служат функции `degrees` и `radians` из библиотеки NumPy (см. введение в NumPy в приложении А). Например, при применении этой функции к широте –30° печатаются следующие результаты:

```
>>> opt_theta(-30, True)
theta = -30.0
delta = -16.8015452415
theta = -46.8015452415
delta = -0.849241867829
theta = -47.6507871093
delta = -0.00275402735705
theta = -47.6535411367
delta = -2.92287909643e-08
-0.41585559653479859
```

Листинг 2.14 ❖ Преобразование точек в проекцию Мольвейде (transform2.py)

```
from numpy import pi, cos, sin, radians, degrees, sqrt

def opt_theta(lat, verbose=False):
    """
    Находим оптимальное значение тета методом Ньютона-Рафсона.
    Вход
      lat: значение широты
      verbose: True, если нужна промежуточная печать
    Выход
```

```
10        theta
11      """
12      lat1 = radians(lat)
13      theta = lat1
14      while True:
15          dtheta = -(theta+sin(theta)-
16                     pi*sin(lat1))/(1.0+cos(theta))
17          if verbose:
18              print "theta =", degrees(theta)
19              print "delta =", degrees(dtheta)
20          if int(1000000*dtheta) == 0:
21              break
22          theta = theta+dtheta
23      return theta/2.0
24
25  def transform2(lon, lat, lon0=0, R=1.0):
26      """
27      Возвращает результат преобразования lon и lat
28      в проекцию Мольвейде.
29      Вход
30        lon: долгота
31        lat: широта
32        lon0: средний меридиан
33        R: радиус глобуса
34      Выход
35        x: координат x (начало координат в точке 0,0)
36        y: координат y (начало координат в точке 0,0)
37        """
38      lon1 = lon-lon0
39      if lon0 <> 0:
40          if lon1>180:
41              lon1 = -((180+lon0)+(lon1-180))
42          elif lon1<-180:
43              lon1 = (180-lon0)-(lon1+180)
44      theta = opt_theta(lat)
45      x = sqrt(8.0)/pi*R*lon1*cos(theta)
46      x = radians(x)
47      y = sqrt(2.0)*R*sin(theta)
48      return x, y
```

Преобразование координат в проекцию Мольвейде производится просто и реализовано в функции `transform2` (строка 25). Во входных аргументах передается, в частности, долгота среднего меридиана (`lon0`). Это, однако, требует дополнительной работы, потому что мы должны гарантировать, что по обе стороны среднего меридиана имеется 180°, хотя входные долготы берутся из исходных данных. Необходимые действия выполняются в блоке кода, начинающемся в строке 39.

Для тестирования проекции Мольвейде нужно внести всего два небольших изменения в код в листинге 2.13: импортировать функцию `transform2` и заменить в строке 11 `transform1` на `transform2`. На рис. 2.10 показаны сетка параллелей и меридианов и очертания материков в проекции Мольвейде.

Рис. 2.10 ❖ Проекция Мольвейде

У проекции Мольвейде есть интересная особенность – меридианы, отстоящие на 90° к востоку и западу от среднего меридиана, преобразуются в идеальные окружности. Поэтому мы можем спроецировать всю поверхность Земли на два круга и расположить их бок о бок. Это часто используется в атласах мира, чтобы разместить восточное и западное полушария на одной странице. На рис. 2.11 показаны пары кругов, получающиеся при разном выборе среднего меридиана.

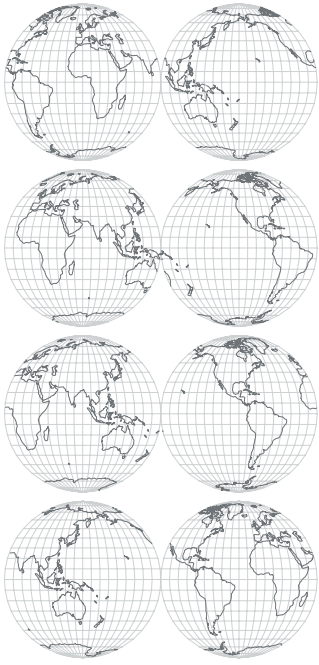

Рис. 2.11 ❖ Проекция Мольвейде карты мира на два круга.
Средние меридианы проходят соответственно по долготе 0°, 60°, 120° и 180°

Что, если мы не станем модифицировать долготы в функции `transform2` в листинге 2.14? Все математические формулы по-прежнему будут работать, но проекция окажется неправильной (рис. 2.12). В упражнениях мы увидим,

какая проблема не позволяет использовать в качестве среднего меридиана долготу, отличную от 0°. Однако это проблема данных, а не формулы и не рассмотренной выше реализации алгоритма.

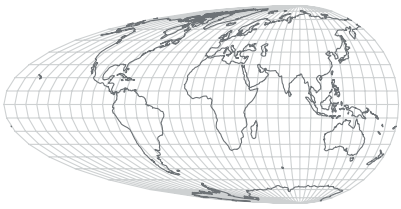

Рис. 2.12 ❖ Проекция Мольвейде со средним меридианом 90° без корректировки долгот

2.9. Примечания

Формулу гаверсинуса для вычисления расстояния по большой окружности можно найти в книге (1984). Формулу расстояния от точки до прямой см. в книге Deza and Deza (2010, стр. 86).

Существует много алгоритмов определения расположения точки относительно многоугольника (Haines, 1994; Huang and Shih, 1997), но в большинстве своем они основаны на теореме Жордана о простых кривых на плоскости (Jordan, 1887). Эта теорема утверждает, что простой многоугольник делит плоскость на две части: внутреннюю и внешнюю, – при этом сам многоугольник является их общей границей. Если не вдаваться в тонкости, то «простой» означает «несамопересекающийся». В этом случае интуитивно понятно, что означает «точка находится внутри многоугольника». Однако семантика нахождения внутри многоугольника – предмет споров, и алгоритмы, основанные на числе пересечений и числе оборотов, могут возвращать разные результаты.

Временная сложность всех алгоритмов, представленных в этой главе, составляет $O(N)$, где N – число сторон многоугольника (Huang and Shih, 1997). Хотя кажется, что это не очень плохо, полное время, необходимое для проверки нахождения точки внутри многоугольника, может быстро расти при увеличении числа сторон. Чтобы уменьшить его, применяются различные способы индексирования, описанные в последующих главах.

Для многоугольников специального вида, в частности монотонных, звездчатых или выпуклых, процедуру можно упростить и сделать более эффективной. Например, для выпуклого многоугольника существует алгоритм со сложностью $O(\log N)$, поскольку никакой луч не может пересекать более двух сторон (O'Rourke, 1998). Для выпуклых (и вообще монотонных) многоугольников можно спроектировать специальный алгоритм, ускоряющий проверку. Например, можно разбить многоугольник на два множества ломаных и каждое множество отсортировать по координатам Y. Тогда для проверки пересечения стороны с лучом можно будет применить двоичный поиск. У. Рэндольф

Фрэнклин (W. Randolph Franklin)[1] предложил также алгоритм для выпуклых многоугольников, состоящий из четырех шагов. На первом шаге находим уравнения прямых, содержащих стороны. На втором шаге записываем эти уравнения в виде $d = ax + by + c$. Точки, принадлежащие прямой, обращают правую часть в нуль. На третьем шаге нормируем каждое уравнение таким образом, что если подставить в него точку, лежащую внутри многоугольника, то результат будет положительным, а для точек вне многоугольника – отрицательным. На четвертом шаге подставляем координаты точки в каждое уравнение. Точка находится внутри многоугольника тогда и только тогда, когда все уравнения дают положительное значение.

Поскольку проекция Робинсона (Robinson, 1974) не основана на математических формулах[2], многие ученые пытались воспроизвести его оригинальную работу (Richardson, 1989; Snyder, 1990; Ipbuker, 2004). Предполагалось также, что Робинсон использовал в своей работе схему Эйткена (Richardson, 1989), похожую на алгоритм Невилла, приведенный в этой книге, но считающуюся устаревшей (Press et al., 2002, стр. 111). Математические детали многих проекций, в т. ч. проекции Мольвейде, см. в книге Snyder (1987).

2.10. Упражнения

1. Напишите на Python программу для вычисления манхэттенского расстояния между двумя точками.
2. Обсуждая вычисление площади многоугольника, мы ничего не сказали о многоугольниках, содержащих дырки. Остается ли приведенная формула площади справедливой для таких многоугольников? Если нет, напишите на Python программу, учитывающую эту возможность. Рекомендуем заглянуть в раздел приложения В.1.4, где обсуждается, как проецировать и отображать сложные многоугольники.
3. Вручную подготовьте набор многоугольников и протестируйте на нем алгоритмы принадлежности точки многоугольнику, рассмотренные в этой главе. Многоугольники могут быть выпуклыми или невыпуклыми, иметь разную форму и быть самопересекающимися.
4. Еще один способ протестировать алгоритмы – взять что-нибудь «реальное», т. е. реальный набор данных. В приложении В описана библиотека GDAL/OGR, которая умеет преобразовывать файлы данных существующих ГИС в простые структуры данных, используемые в этой главе, и тестировать алгоритмы принадлежности точки многоугольнику.

---

[1] PNPOLY – Point Inclusion in Polygon Test (http://www.ecse.rpi.edu/Homepages/wrf/Research/Short_Notes/pnpoly.html).

[2] В статье в газете New York Times (Wilford, 1988) приводится высказывание самого Робинсона: «Я выбрал своего рода артистический подход. Я представил себе формы и размеры, наиболее приятные глазу. И настраивал параметры, пока не достиг состояния, когда изменение любого из них ничего не улучшало. Тогда я вывел математическую формулу, дающую такой эффект. Большинство картографов начинают с математики».

5. Мы упомянули, что версия алгоритма на основе числа оборотов с тригонометрическими функциями работает долго. А так ли это? И снова воспользуйтесь функциями из библиотеки GDAL/OGR, рассматриваемой в приложении B.
6. Мы решили проблему среднего меридиана в коде, реализующем проекцию Мольвейде. Но если вы действительно захотите воспользоваться средним меридианом, долгота которого отлична от 0°, то многие береговые линии будут неправильно вытянуты по горизонтали. Дело в том, что наши данные привязаны к сетке, в которой долготы изменяются от –180° (к востоку) до 180° (к западу). Кроме того, если долгота среднего меридиана не кратна 10°, то мы будем иметь пару меридианов, проецируемых в другой интервал. Найдите способ решить эту проблему, так чтобы можно было правильно проецировать данные при любом среднем меридиане.
7. Программа на Python для проекции Робинсона не содержит кода для обработки других средних меридианов. Если вы справились с упражнением 6, то теперь самое время включить возможность задания среднего меридиана пользователем в программу для проекции Робинсона.
8. Теперь, разобравшись с проекцией Робинсона, мы можем поэкспериментировать с другими проекциями. Например, сможете ли вы применить аналогичный подход (как в табл. 2.1) для проецирования карты мира на треугольную сетку, где южный полюс представляется отрезком той же длины, что средний меридиан, а северный полюс – точкой? Разумеется, есть и другие формы сетки, которые было бы интересно исследовать.
9. В большинстве проекций масштаб в разных точках различен. Теперь, когда мы знаем, как получить данные для сетки, и умеем вычислять расстояния в разных системах координат (сферической и декартовой), можно систематически изучить вопрос о поведении коэффициента масштабирования на картографической проекции. Напишите соответствующую программу на Python и отобразите результаты на карте. Каковы закономерности искажения масштаба на проекциях Робинсона и Мольвейде?
10. На рис. 2.11 мы показали ряд карт в проекции Мольвейде, не приведя кода. Для построения этого рисунка использовались те же данные об очертании материков, что для других карт, и функция `transform2` из листинга 2.14. Напишите на Python программу, генерирующую карты на этом рисунке.